

CESAR PEREIDA GARCÍA

Side-Channel Analysis and Cryptography Engineering

Getting OpenSSL Closer to Constant-Time

CESAR PEREIDA GARCÍA

Side-Channel Analysis
and Cryptography Engineering
Getting OpenSSL Closer to Constant-Time

ACADEMIC DISSERTATION

To be presented, with the permission of
the Faculty of Information Technology and Communication Sciences
of Tampere University,
for public discussion at Tampere University,
on 11 February 2022, at 12 o'clock.

ACADEMIC DISSERTATION

Tampere University,
Faculty of Information Technology and Communication Sciences
Finland

*Responsible
supervisor
and Custos*

Associate Professor
Billy Bob Brumley
Tampere University
Finland

Pre-examiners

Senior Lecturer Daniel Page
University of Bristol
United Kingdom

Dr. Clémentine Maurice
French National Centre for
Scientific Research (CNRS)
France

Opponent

Associate Professor Yuval Yarom
University of Adelaide
Australia

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

Copyright ©2022 author

Cover design: Roihu Inc.

ISBN 978-952-03-2288-5 (print)

ISBN 978-952-03-2289-2 (pdf)

ISSN 2489-9860 (print)

ISSN 2490-0028 (pdf)

<http://urn.fi/URN:ISBN:978-952-03-2289-2>

PunaMusta Oy – Yliopistopaino
Joensuu 2022

Dedicado a mi *Amá* y a mi *Apá*.

PREFACE

This dissertation and the related research work was possible thanks to the support provided by many people.

First and foremost, I am indebted to Prof. Billy Bob Brumley, the finest mentor and supervisor I could have asked for. This dissertation would not be possible without your support inside and outside of the office, your patience while I was slowly learning the ropes of the trade, the motivation and resilience you instill in the research group, the expertise you provide on several topics beyond academy—you have my deepest respect, please accept my sincerest thanks.

I wish to extend my gratitude to Prof. Daniel Page and Dr. Clémentine Maurice for evaluating this dissertation, and providing me with suggestions to further improve it. Furthermore, my gratitude goes to Prof. Yuval Yarom for agreeing on being my opponent during the public defense.

I want to express my gratitude to the following organizations that generously supported my research work over the years: The Doctoral School at Tampere University of Technology, Academy of Finland (grant #303814), the European Cooperation in Science and Technology (COST) Actions IC1306 and IC1403, the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No 804476), the Nokia Foundation, the Industrial Research Fund at Tampere University of Technology, and Huawei Technologies Oy.

I would like to thank my co-authors with whom I’ve had the joy of collaborating.

I also would like to thank my fellow colleagues in the NISEC research group at Tampere University. In addition to the research work we did, you taught me how to navigate in a new university, and in a new city.

Special thanks go to Nicola, Ale, and Sohaib, with whom I’ve shared many

lunches, discussed ideas, shared the long nights of Finland, travelled to conferences and research visits, played board games and videogames, watched movies, and experienced many other adventures.

In the same vein, I am grateful to all the people that took me in, and supported me during this journey from one way or another, including those that shared their time, their knowledge, a laugh, and a dance with me.

None of this would be possible without the support, love, and patience from my wife, my parents, my siblings, and my extended family in Finland—thanks for everything.

Tampere, January 10, 2022,

Cesar Pereida García

ABSTRACT

As side-channel attacks reached general purpose PCs and started to be more practical for attackers to exploit, OpenSSL adopted in 2005 a flagging mechanism to protect against SCA. The opt-in mechanism allows to flag secret values, such as keys, with the `BN_FLG_CONSTTIME` flag. Whenever a flag is checked and detected, the library changes its execution flow to SCA-secure functions that are slower but safer, protecting these secret values from being leaked. This mechanism favors performance over security, it is error-prone, and is obscure for most library developers, increasing the potential for side-channel vulnerabilities. This dissertation presents an extensive side-channel analysis of OpenSSL and criticizes its fragile flagging mechanism. This analysis reveals several flaws affecting the library resulting in multiple side-channel attacks, improved cache-timing attack techniques, and a new side channel vector. The first part of this dissertation introduces the main topic and the necessary related work, including the microarchitecture, the cache hierarchy, and attack techniques; then it presents a brief troubled history of side-channel attacks and defenses in OpenSSL, setting the stage for the related publications. This dissertation includes seven original publications contributing to the area of side-channel analysis, microarchitecture timing attacks, and applied cryptography. From an SCA perspective, the results identify several vulnerabilities and flaws enabling protocol-level attacks on RSA, DSA, and ECDSA, in addition to full SCA of the SM2 cryptosystem. With respect to microarchitecture timing attacks, the dissertation presents a new side-channel vector due to port contention in the CPU execution units. And finally, on the applied cryptography front, OpenSSL now enjoys a revamped code base securing several cryptosystems against SCA, favoring a secure-by-default protection against side-channel attacks, instead of the insecure opt-in flagging mechanism provided by the fragile `BN_FLG_CONSTTIME` flag.

CONTENTS

Preface	v
Abstract	vii
Abbreviations	xii
Original publications	xvii
Author's contribution	xix
1 Introduction	21
1.1 Main Contributions	23
1.2 Scope	27
1.3 Outline	30
2 Background	31
2.1 The Microarchitecture	31
2.1.1 Pipelining	33
2.1.2 The Cache Hierarchy	36
2.2 Cache-Timing Attacks and Techniques	41
2.3 A Brief History of SCA against OpenSSL	49
3 Results	61
3.1 Make Sure DSA Signatures are closer to Constant-Time	61
3.2 A Cache-Timing Attack on Constant-Time NIST P-256	63
3.3 Single-Trace Cache-Timing Attack on RSA Key Generation	65
3.4 A Game of Whack-A-Mole	67
3.5 Towards an SCA-secure OpenSSL	69

3.6	Port Contention Side-Channel Attack	72
3.7	Side-Channel Attacks Enabled by Cryptographic Key Formats	75
3.8	Summary of Mitigations on OpenSSL	78
4	Conclusions	83
	References	91
	Publication I	119
	Publication II	133
	Publication III	153
	Publication IV	185
	Publication V	209
	Publication VI	225
	Publication VII	245

List of Figures

2.1	Simplified microarchitecture containing a variety of components.	32
2.2	Example of a simple pipeline with stalls.	34
2.3	Memory hierarchy showing different memory levels color coded according to their access speed. Blue to the left is the fastest, and red to the right is the slowest.	36
2.4	Top: 64-byte cache line structure; Bottom: memory address structure.	37
2.5	An <i>m-way</i> set-associative cache: the index selects the cache set, the tag uniquely identifies the data within the cache set, and the offset determines the data location in the data block.	39

2.6	The EVICT+TIME technique: (a) attacker times operation to obtain a baseline; (b) attacker partially evicts the cache, e.g., a cache-set; (c) attacker times and compares.	43
2.7	The PRIME+PROBE technique: (a) attacker primes some cache-sets by loading its own data; (b) attacker waits for the victim to evict its data; (c) attacker probes own data by loading and timing it; (d) example trace probing 32 cache sets during DSA modular exponentiation revealing multiplier look-ups.	44
2.8	The FLUSH+RELOAD technique: (a) attacker flushes a memory address shared with the victim out of the cache; (b) attacker waits for some time; (c) attacker reloads memory address and times it; (d) example trace probing 2 cache-lines accessed by the victim revealing a sequence of operations.	45
2.9	Timeline of (mostly) microarchitectural side-channel attacks and countermeasures on asymmetric primitives in OpenSSL.	49
3.1	The PORTSMASH technique: (a) attacker issues instructions saturating a set of ports; (b) attacker times the instruction execution completion; (c) attacker continues timing and observes variations due to victim; (d) example trace of attacker spying on the <i>wNAF</i> scalar multiplication revealing add operations (peaks).	73
3.2	Updated timeline including attacks and countermeasures from Publications I–VII.	81

List of Tables

1.1	Relationship between research questions and publications.	24
-----	---	----

ABBREVIATIONS

AES	Advanced Encryption Standard
ANSI	American National Standards Institute
ASLR	Address Space Layout Randomization
AVX2	Advance Vector Extensions 2
BEEA	Binary Extended Euclidean Algorithm
BN	BIGNUM structure in OpenSSL
BPU	Branch Prediction Unit
CISC	Complex Instruction Set Computer
CPU	Central Processor Unit
CRT	Chinese Remainder Theorem
CVE	Common Vulnerability and Exposures
dcache	data cache
DH	Diffie-Hellman key exchange
DoS	Denial-of-Service
DPA	Differential Power Analysis
DRAM	Dynamic Random-Access Memory
DSA	Digital Signature Algorithm
EC	Elliptic Curve
ECC	Elliptic Curve Cryptography
ECDH	Elliptic Curve Diffie-Hellman key exchange
ECDSA	Elliptic Curve Digital Signature Algorithm

ECIES	Elliptic Curve Integrated Encryption Scheme
EM	Electro Magnetic
EOL	End of Life
FLT	Fermat's Little Theorem
FOSS	Free and Open-Source Software
GCD	Greate common divisor
GOST	Soviet and Russian government cryptographic standards
HSM	Hardware Security Module
HT	Hyper-Threading
icache	instruction cache
ISA	Instruction Set Architecture
L1	first-level cache
L2	second-level cache
LLC	last-level cache
lru	Least Recently Used
LSB	Least Significant Bits
LSD	Least Significant Digit
LTS	Long Term Support
MSBLOB	Microsoft's Private Key BLOB format
NAF	Non-Adjacent Form
NIST	National Institute of Standards and Technology
OID	Object Identifier
OS	Operating System
PC	Personal Computer
PEM	Privacy-Enhanced Mail
PKCS	Public Key Cryptography Standards
PVK	Private Key format

RISC	Reduced Instruction Set Computer
RSA	Rivest-Shamir-Adleman public-key cryptosystem
SBPA	Simple Branch Prediction Analysis
SCA	Side-Channel Analysis
SECG	Standards for Efficient Cryptography Group
SGX	Software Guard Extensions
SM2	Chinese government elliptic curve cryptographic standards
SMT	Simultaneous Multithreading
SSE2	Streaming SIMD Extensions 2
SSH	Secure Shell Protocol
SSL	Secure Sockets Layer
TLS	Transport Layer Security
TSX	Transactional Synchronization Extensions
wNAF	windowed Non-Adjacent Form

ORIGINAL PUBLICATIONS

This dissertation is based on the following original publications.

- Publication I C. Pereida García, B. B. Brumley and Y. Yarom. “Make Sure DSA Signing Exponentiations Really are Constant-Time”. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. Ed. by E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers and S. Halevi. ACM, 2016, 1639–1650. DOI: 10.1145/2976749.2978420.
- Publication II C. Pereida García and B. B. Brumley. Constant-Time Callees with Variable-Time Callers. *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. Ed. by E. Kirda and T. Ristenpart. USENIX Association, 2017, 83–98. ISBN: 978-1-931971-40-9. URL: <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/garcia>.
- Publication III A. C. Aldaya, C. Pereida García, L. M. Alvarez Tapia and B. B. Brumley. Cache-Timing Attacks on RSA Key Generation. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019.4 (2019), 213–242. DOI: 10.13154/tches.v2019.i4.213-242.
- Publication IV I. Gridin, C. Pereida García, N. Tuveri and B. B. Brumley. Triggerflow: Regression Testing by Advanced Execution Path Inspection. *Detection of Intrusions and Malware, and Vulnerability Assessment - 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19-20, 2019, Pro-*

ceedings. Ed. by R. Perdisci, C. Maurice, G. Giacinto and M. Almgren. Vol. 11543. Lecture Notes in Computer Science. Springer, 2019, 330–350. DOI: 10.1007/978-3-030-22038-9_16.

- Publication V N. Tuveri, S. ul Hassan, C. Pereida García and B. B. Brumley. Side-Channel Analysis of SM2: A Late-Stage Featurization Case Study. *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. ACM, 2018, 147–160. DOI: 10.1145/3274694.3274725.
- Publication VI A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. Pereida García and N. Tuveri. Port Contention for Fun and Profit. *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 2019, 870–887. DOI: 10.1109/SP.2019.00066.
- Publication VII C. Pereida García, S. ul Hassan, N. Tuveri, I. Gridin, A. C. Aldaya and B. B. Brumley. Certified Side Channels. *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. Ed. by S. Capkun and F. Roesner. USENIX Association, 2020, 2021–2038. URL: <https://www.usenix.org/conference/usenixsecurity20/presentation/garcia>.

AUTHOR'S CONTRIBUTION

- Publication I The current author is responsible for implementing the cache-timing attack, implementing the performance degradation attack, adapting the tools to attack at the protocol level, implementing the countermeasures, and the related writing.
- Publication II The current author is responsible for implementing the cache-timing attack, implementing the improved performance degradation attack, adapting the tooling to attack at the protocol level, implementing the countermeasures, and the related writing.
- Publication III The current author is responsible for implementing the cache-timing attack, developing signal processing techniques, testing the new methodology, implementing countermeasures, and the related writing.
- Publication IV The current author is responsible for testing and validating the new tool with respect to existing cache-timing attacks in the literature, as well as the related writing.
- Publication V The current author is responsible for the cache-timing attack portion of the work, implementing some of the countermeasures, and the related writing.
- Publication VI The current author is partially responsible for discovering the new side-channel vector, for the procurement portion of the new side-channel attack, and the related writing.
- Publication VII The current author is partially responsible for discovering and analyzing the vulnerabilities, implementing the cache-

timing attack, implementing some of the countermeasures,
and the related writing.

1 INTRODUCTION

In the current digital world where people and devices are always connected, the confidentiality, integrity, and privacy of data among users and devices is of utmost importance, especially as the generation, storage, and transportation of personal identifiable information and confidential information increases. Cryptography engineering is an essential building block necessary to achieve these security goals in both software and hardware, however, often these goals are in conflict with other requirements such as size and performance, resulting in the complex task of implementing secure and fast cryptosystems with a reasonable code size. Moreover, it is common to find a disconnection between cryptography and software engineering, causing problems when cryptosystems are converted from the mathematical model in paper to the practical implementation in hardware and software. These problems arise since the security models from theoretical constructions do not translate to the practical implementations, where attackers have more power and thus additional threats must be considered to achieve full security.

It is not difficult to find examples of theoretically secure cryptosystems and protocols that fail to provide security guarantees when implemented in a system. Perhaps the most infamous example of this is the *HeartBleed* vulnerability affecting OpenSSL, the most widely used cryptographic library on the Internet. RFC 6520 [120] describes the *Heartbeat* extension as a keep-alive mechanism for TLS and DTLS, however, the implementation in OpenSSL failed to check the payload request and response memory bounds, thus a server using this library responded to any client with as much data as requested, potentially including the server's own private keys if they happened to be in memory. In addition to the severity of *HeartBleed* affecting millions of servers running OpenSSL, this issue unearthed a bigger problem affecting open-source security software such as OpenSSL. The industry and research communities realized that a considerable

part of the world infrastructure is secured by a handful of open-source software projects that are maintained by few researchers and developers during their free time. At the time of *HeartBleed*, many of these security-critical projects, including OpenSSL, were running with little to no funding, and worryingly under-staffed, making clear why this and other serious security vulnerabilities slipped under the radar for such a long time. Soon after the *HeartBleed* disaster, OpenSSL received much needed funding, leading to a better code base, and an improved testing infrastructure.

Fast-forward to 2016, this dissertation uses a post-*HeartBleed* OpenSSL as a research tool to investigate more subtle but equally important software defects affecting the security of an otherwise functionally correct cryptographic library. These software defects are well beyond the considerations of the mathematical models for many protocols and cryptosystems, however this dissertation demonstrates that they are present in OpenSSL, and leak confidential information to an experienced attacker. The discovery, analysis, exploitation, and remediation of these data leaks in practical implementations of cryptosystems are part of the research field called Side-Channel Analysis (SCA). Several branches exist within SCA based on the channel from which the information is being leaked, e.g., EM radiation, power consumption, execution time, microarchitecture, etc. The majority of this work focuses on cache-timing attacks, whose goal is to exploit the cache behavior to leak confidential information during the execution of a variety of cryptographic algorithms—in this case applied to OpenSSL—to ultimately recover secret keys.

OpenSSL uses a flagging mechanism to try to protect not only from cache-timing attacks but from SCA in general. The mechanism allows to flag variables holding secret values so that only SCA-secure functions operate on them, thus avoiding information leakage. The flagging mechanism was introduced more than a decade ago [105] with some level of success—only a handful of attacks were published during the decade after its initial introduction [3, 4, 5, 23, 30, 32, 144]. However, no proper testing infrastructure was added to this mechanism, thus its effectiveness was implicit but not certain. This dissertation takes a closer look at this mechanism and its effectiveness to protect against cache-timing attacks, uncovering several flaws and software defects enabling SCA, in addition to an increased technical debt as a result of a security mechanism

focused on performance instead of security.

1.1 Main Contributions

OpenSSL is arguably the most widely used cryptographic library in the world, and it has influenced several newer implementations. The impact of this research work goes beyond academic results as it affects and improves the practical security of real-world deployments using OpenSSL, enjoying a more robust security against SCA and cache-timing attacks.

The bulk of research work leading to this dissertation contributes to the field of SCA and cache-timing attacks on both fronts, offense and defense. On the offense, the work discovers and exploits new vulnerabilities, improves the exploitation techniques, and finds new attack vectors. On the defense side, the work improves the code quality of OpenSSL, and applies much needed countermeasures not only to OpenSSL, but also to other cryptographic libraries such as LibreSSL and BoringSSL against side-channel attacks.

This dissertation consists of seven novel and original scientific publications published across several highly ranked system security and applied cryptography venues. The publications offer new contributions and new results advancing both SCA research and the practical security of OpenSSL. Table 1.1 shows the relationship between the publications and the research questions they answer to. As can be seen, all of the publications relate to multiple research questions since the publications complement each other, in other words, the publications are tightly connected to each other. This demonstrates this work is the result of a continuous and iterative process, a common approach followed in this research field.

In general, this dissertation answers the following research questions:

Research Question 1. *How secure are public-key cryptographic primitives against Side-Channel Analysis as implemented in OpenSSL?*

Research Question 2. *How can side-channel vulnerabilities be prevented in OpenSSL?*

Research Question 3. *How can side-channel vulnerabilities be efficiently detected in OpenSSL?*

Research Question 4. *In addition to the caches, do other microarchitec-*

	RQ1	RQ2	RQ3	RQ4
Publication I	✓	✓		
Publication II	✓	✓		
Publication III	✓		✓	
Publication IV	✓		✓	
Publication V	✓	✓		
Publication VI	✓	✓		✓
Publication VII	✓	✓	✓	

Table 1.1 Relationship between research questions and publications.

ture components leak confidential information during cryptographic operations?

The next paragraphs offer a summary for each publication.

Publication I. In the realm of SCA, cache-timing attacks have seen important improvements in terms of methodology and results, however, many of these cache-timing attacks are not completely practical. Most of the previous works demonstrate attacks on any of the following scenarios: 1) own implementation of the target cryptosystem; 2) well-established implementation but executed in isolation; or 3) well-established implementation where the attacker’s code is embedded into the target cryptosystem. While relevant, these scenarios allow an attacker to benefit from increased signal-to-noise ratio, thus improving the quality of the signal, allowing them to better observe and process the data leakage. Unfortunately, these scenarios do not represent the typical obstacles that an attacker would encounter in a practical scenario. As the distance—in the software stack—between the attacker and the target increases, so does the difficulty of performing a cache-timing attack, as the attacker has less control of the environment. Publication I answers **RQ1** by presenting the first end-to-end practical cache-timing attack against a cryptographic primitive executing as part of a protocol— DSA signature generation as part of TLS and OpenSSH protocols. The cache-timing attack is paired with a performance degradation attack in order to slowdown the overall execution of DSA. This allows to target the TLS and SSH protocols linked against OpenSSL during DSA digital signature generation, ultimately enabling full key recovery. The root-cause analysis reveals that the `BN_FLG_CONSTTIME` flag is the culprit, as it is not correctly set,

leading to the execution of an SCA-vulnerable modular exponentiation function during DSA computation. Additionally, this work is relevant as it reflects on the possibility to prevent SCA of OpenSSL, thus leading to **RQ2**. Moreover, this work represents the first instance (of many to come) of a side-channel attack attributed to the error-prone and insecure-by-default approach supported by the `BN_FLG_CONSTTIME` flag mechanism in OpenSSL.

Publication II. The fixed-point scalar multiplication operation is a common SCA target due to the complexity of implementing it in a constant-time manner. However, even if this is achieved, it is not the only operation that can potentially leak secret information during signature generation. Publication II answers **RQ1** by demonstrating the first practical cache-timing attack against the modular inversion operation used during ECDSA computation on the popular NIST P-256 curve featuring a specialized constant-time scalar multiplication implementation. This work improves the performance degradation attack used in Publication I, and combines it with an LLC cache-timing attack against a highly input-dependent modular inversion operation in OpenSSL. Similar to the previous case, the root-cause analysis reveals the vulnerability is enabled by yet another software defect in the fragile `BN_FLG_CONSTTIME` flag. This work continues the discussion of the failing flag mechanism to protect against SCA.

Publication III. Key generation is a cryptographic operation commonly neglected outside of power and EM analysis since, historically, keys have been generated and procured in secure and controlled environments such as Hardware Security Modules (HSM). Moreover, key generation is a one-time operation, therefore SCA with a single trace was not deemed feasible, especially on software. Publication III answers **RQ1** by challenging this assumption and demonstrating the first single-trace cache-timing attack on RSA key generation with a modest success rate. The attack targets the input-dependent binary GCD algorithm used during a coprimality test, leaking partial information of the secret prime values p and q . This work answers **RQ3** by proposing a new methodology to identify the usage of input-dependent functions with secret inputs—an undesirable but recurring situation in OpenSSL and other cryptographic libraries.

Publication IV. SCA not only requires a deep understanding of cryptography but also requires wide set of skills such as microarchitecture, operating systems, algorithms, implementation details, and good cryptography engineering practices. This allows to analyze, detect, and remediate flaws leaking secret information on a given implementation. Conveying all this knowledge to a tool in order to automatically detect side-channel leakage is a challenging task, hence automatic discovery of side-channel leakage with little to no input from the user is unfeasible. Publications I–III and VII suggest that many side-channel attacks can be attributed to a small set of implementation flaws, thus by identifying these flaws it is possible to detect similar leakage in other scenarios and cryptographic libraries, as well as detect regressions reintroducing side-channel vulnerabilities previously fixed. Publication IV answers **RQ3** by expanding and implementing the methodology proposed in Publication III as a stand-alone dynamic analysis tool capable of selectively tracking function calls by using small code annotations. The tool serves not as an automated side-channel detection tool, but as a supporting tool for developers with access to the cryptographic source code. The tool can be integrated to the development pipeline using Continuous Integration (CI) to identify new flaws and detect code regressions.

Publication V. SM2 is a suite of elliptic curve cryptosystems originating from Chinese standards, equivalent and similar to those proposed by NIST. Given the similarity to existing elliptic curve primitives in OpenSSL, one could think that their implementation would be as secure as existing primitives in the library—unfortunately this is not the case. Publication V shows that SCA-secure implementation of modern cryptography is a complex task requiring in-depth analysis to avoid common pitfalls found in software engineering, in addition to advanced flaws leaking secret information. This work presents a security evaluation of SM2, with respect to SCA, as implemented in OpenSSL just before the release of version 1.1.1. This evaluation uncovers several software flaws in the code, enabling a variety of side-channel attacks including power, timing, and cache-timing attacks. This work answers **RQ1**, and it takes a strong stance against the repetitive and common issues with the `BN_FLG_CONSTTIME` and provides several countermeasures, answering **RQ2**.

Publication VI. Up to this point, previous works exploit software flaws in OpenSSL using the cache as a side-channel, however, caches are not the only microarchitecture component leaking confidential information due to time variation during execution. In fact, caches are only some of the many microarchitecture components known to leak information. This rapidly increasing field within SCA is termed microarchitecture attacks. Publication VI answers **RQ1** and **RQ4** by introducing a new microarchitecture side-channel attack exploiting port contention inside the execution units within the microprocessor. This attack is possible on microarchitectures supporting simultaneous multi-threading (SMT). Additionally, this work demonstrates that the state-of-the-art Intel SGX technology is still vulnerable to the new side-channel attack since the source of leakage is oblivious to this security feature.

Publication VII. Previous Publications I–VI demonstrate side-channel attacks against OpenSSL low-level SCA-vulnerable functions from a higher level protocol such as TLS and SSH. Unsurprisingly, this is arguably the most common use case for the affected cryptosystems, therefore resulting in the deepest research impact. However, side-channel attacks can also be exploited from other protocols in less known scenarios. Publication VII answers **RQ1**, **RQ2**, and **RQ3** by using the tool developed in Publication IV to analyze the execution flow followed by OpenSSL and mbedTLS during private key parsing and private key conversion for diverse key formats containing valid but uncommon key parameters. The analysis finds multiple software flaws and weaknesses affecting RSA, DSA, and ECDSA keys, and demonstrate different side-channel attacks. Similar to Publication V, this work receives a full SCA treatment, resulting in traditional timing, cache-timing, and EM attacks.

1.2 Scope

The previous description in Section 1.1 gives a high-level overview of the topics relevant to this dissertation. However, SCA is a vast and rapidly expanding research field, and this dissertation does not aim at covering every single topic and aspect related to this field. To that end, the following paragraphs delimit the scope of this work by describing related topics that are not covered in this

dissertation but that the reader may find interesting.

Other Cryptographic Libraries. Several cryptographic libraries exist on the Internet, each of them developed with a different use case in mind. While some of them try to be general purpose libraries, other cover very specific use cases. This dissertation focuses on OpenSSL due to its widespread usage on the Internet, the impact it has had on other libraries, and its continuous support and (mis)use of the `BN_FLG_CONSTTIME` flag. Although Chapter 3 briefly mentions vulnerabilities impacting other cryptographic libraries such as BoringSSL and LibreSSL, their security analysis is out of the scope of this dissertation.

SCA of Symmetric Key Cryptosystems. This dissertation and related publications focus on cache-timing attacks performed on asymmetric key cryptosystems such as DSA, RSA, and ECDSA. However, symmetric key cryptosystems are also prone to implementation flaws leading to leakage of confidential information. The best and most common example of this is the Advanced Encryption Standard (AES) cryptosystem in its T-table implementation. Bernstein [24] demonstrated that this particular AES implementation, introduced more than two decades ago, is vulnerable against cache-timing attacks. Since then it is commonly used as a target benchmark to introduce new side-channel attacks and vectors [54, 59, 71, 103, 126]. Moreover, state-of-the-art cache-timing attack techniques discussed in Section 2.2 started as techniques against symmetric key cryptosystems, and later found their application as techniques to attack asymmetric key cryptosystems—such as the ones presented in Publications I–VII. Nowadays, AES has a greatly reduced side-channel attack surface due to the adoption of hardware accelerated AES instructions, such as Intel’s Advanced Encryption Standard New Instructions (AES-NI) [56], and ARMv8 Cryptographic Extension [20]. In both cases, symmetric and asymmetric key cryptosystems, protection against SCA is a mandatory requirement for a primitive to be deemed secure.

Lattices. Lattices are mathematical structures used in cryptography and cryptanalysis, and among their many uses, they allow to find secret keys by finding a small solution to undetermined systems of equations. Lattices are a recurring theme in Publications I–III, V–VII, where these mathematical structures are used during the last part of the attacks to fully recover secret keys

by combining public information with the information leaked through the side-channel. These structured are complex and the related publications use lattices as black boxes—i.e., they find a solution (private keys) when given many small partial solutions to the problem (leaked information) and a generous amount of time. For this reason, this dissertation does not discuss the theory behind lattices neither their application to cryptanalysis nor post-quantum cryptosystems. For interested readers, Howgrave-Graham and Smart [65] present the first, and perhaps most reader-friendly, work on lattices as a cryptanalysis tool applied to DSA, which later Nguyen and Shparlinski [101] adapted to ECDSA.

Automated Detection of Side-Channel Leakage. Cryptography does not exist in a vacuum, it is typically used as part of a bigger and more elaborated security protocol. Nevertheless, it can be implemented in practically any programming language, leading to a situation where the security of the implementation not only depends on the implementation itself, but also on the underlying hardware and software stacks. In other words, the success of SCA depends on a wide variety of software and hardware details and factors including the target cryptosystem, the implementation, the programming language used to implement the cryptographic primitive, the protocol using the implementation, the compilation options used to compile the code, the microprocessor executing the code, and the side-channel being exploited, just to name a few. Convoluted function hierarchies leading to deep stack of function calls in cryptographic libraries make automated identification, analysis, and quantification of side-channel leakage points and vulnerabilities an incredibly complex task. Publication IV describes a tool that supports the development process of cryptography engineers to find repeating instances of known side-channel vulnerabilities and general security vulnerabilities in their projects. This tool does not try to find new side-channel leakage points nor quantifies leakage, instead, it helps developers to find code paths and functions that should or should not be reached, thus it has a more general applicability beyond SCA. For this reason, this dissertation does not discuss the theory and practice behind automated detection and quantification of side-channel leakage, but instead, interested readers may find useful the following references using different approaches to achieve these goals, e.g., programming languages [21, 27, 36, 124], black box testing [111], static program analysis [15, 113, 142], and dynamic program anal-

ysis [84, 138, 141].

Trusted Execution Environments (TEEs). TEEs are secure areas of execution within the main processor commonly implemented on hardware, but can also be implemented solely on software. Their main goal is to protect the data being processed by only allowing trusted applications to operate in this data, thus effectively isolating the data in the secure world from the insecure world. It is inevitable to talk about SCA without mentioning TEEs and their effect on the field of microarchitectural attacks. In fact, recent work [116] argues that TEEs fuel research on microarchitectural attacks, and this is somewhat reflected on Publication VI. Despite this close relationship between SCA, cache-timing attacks, microarchitecture attacks, and TEEs, this topic is not covered in this dissertation. The following references are recommended as they provide an overview of TEEs from different perspectives [37, 116, 148].

1.3 Outline

This compilation-based dissertation is divided in two parts. In the first part, Chapter 1 introduces the topic, gives an overview of the main contributions, defines the scope, and the outline. Chapter 2 presents background information on the microarchitecture, the caches, the cache-timing attack techniques, and a brief history of side-channel attacks on OpenSSL. Chapter 3 presents the results from Publications I–VII, discusses the vulnerabilities found, the microarchitecture attack techniques used, and the countermeasures developed to improve the security of OpenSSL. Chapter 4 discusses the practical implications of the research work, the real-world impact, the future of microarchitectural attacks and OpenSSL, as well as summarizes the results and concludes the dissertation. The second part includes a compilation of original Publications I–VII as published in the respective journals and conference proceedings.

2 BACKGROUND

This chapter presents a brief introduction to several topics revolving around the central theme of the dissertation. Section 2.1 introduces the microarchitecture and further describes the most relevant components for this dissertation, namely, the pipeline and the cache hierarchy. Section 2.2 offers an overview of cache-timing attacks and the most widely used techniques to perform them. Section 2.3 recapitulates the history of side-channel attacks against OpenSSL to set the scene leading to Publications I–VII and the results in Chapter 3.

2.1 The Microarchitecture

An instruction set architecture (ISA), commonly known as architecture, is an abstract model of a CPU. This model specifies the functionality of the programming interface of a CPU, e.g., instruction set, supported data types, memory management, registers, etc. The ISA effectively abstracts away irrelevant details of the microarchitecture with respect to functionality. Architectures are commonly classified based on the complexity of their instruction set, being Complex Instruction Set Computer (CISC) and Reduced Instruction Set Computer (RISC) the two most prominent architecture families due to the widespread usage of x86 and ARM processors, respectively.

On the other hand, a microarchitecture is an actual implementation of a particular ISA in a processor. The microarchitecture description contains all the implementation details of said microarchitecture, however, often many details are not documented or they are not publicly available, thus demanding reverse engineering efforts from the research community [1, 67, 72, 93, 146]. Generally speaking, different microarchitectures can implement the same ISA, however, the performance observed within a CPU family implementing the same ISA can vary significantly due to differences in microarchitecture components, features,

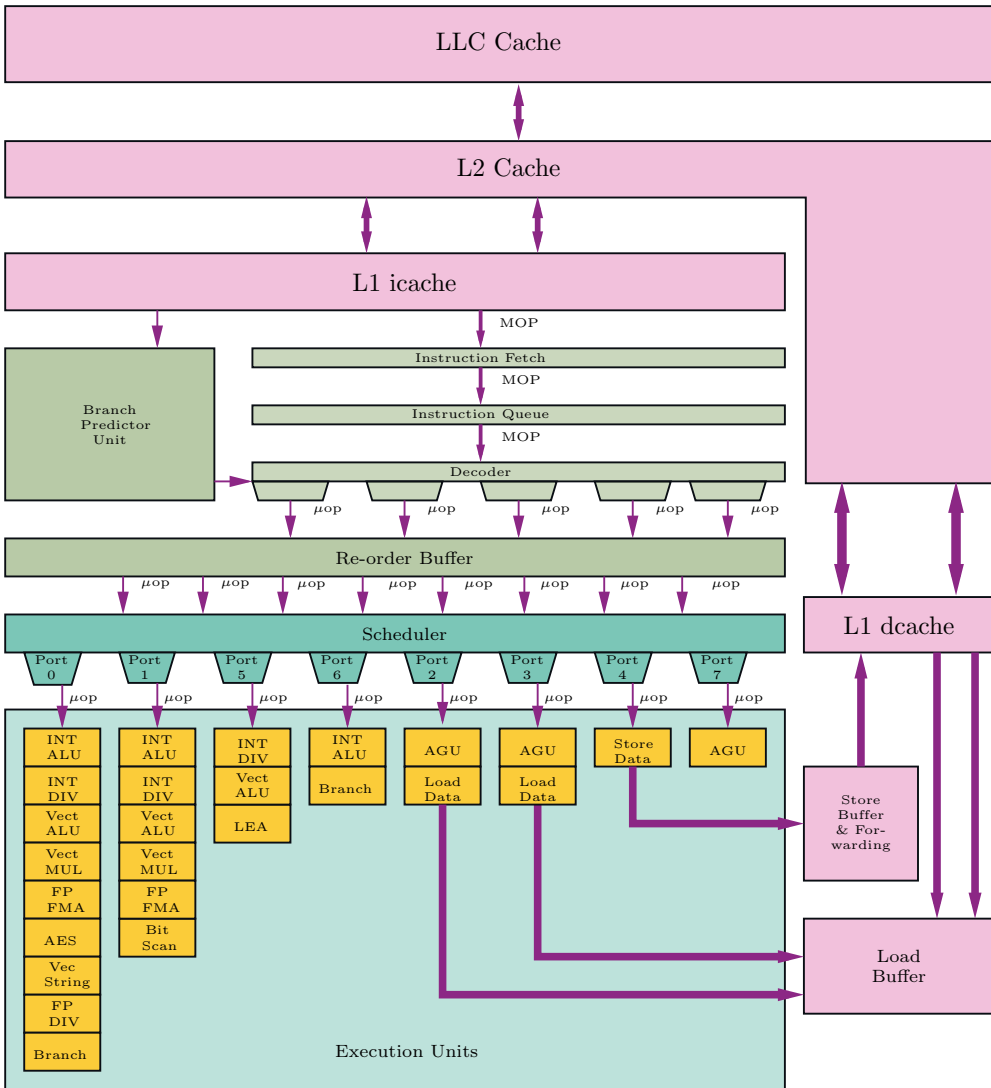


Figure 2.1 Simplified microarchitecture containing a variety of components.

and optimizations.

The microarchitecture is composed of several microarchitecture components implementing a wide range of techniques such as pipelining and out-of-order execution, to increase the overall performance of the microprocessor. Figure 2.1 shows a simplified diagram of Intel Skylake and newer microarchitectures containing components such as execution units, caches, buffers, predictors, fetchers, decoders, and schedulers.

A typical programmer does not care and does not directly interact with these low level microarchitecture components, as they are abstracted away by the ISA, which guarantees functionality as long as the programmer adheres to it. In other words, a program is potentially portable if it adheres to the ISA and it does not depend on any specific microarchitecture components or features. Moreover, by combining increasingly complex microarchitectures and smart compilers, microprocessors achieve an incredible level of optimization without the programmers even knowing about it.

2.1.1 Pipelining

In contrast to specific designs, modern general purpose microprocessors use pipelining to process instructions in a more efficient way. Pipelining means to break down instruction processing into multiple stages, thus, allowing multiple stages to complete at the same time, increasing the overall performance of the microprocessor. As effective as pipelining is, it does not solve all the performance problems in the processor. Pipelining suffers from scalability issues, that is, it cannot increase indefinitely. As in any other system, the pipeline overall performance is dictated by the slowest of its stages, and a common cause for slow down in the pipeline are waiting periods—also known as stalls—caused by a variety of reasons, including: memory delays, data dependencies, control dependencies, and resource contention.

- *Memory delays* occur when data is not readily available in the current stage, causing the pipeline to stall until data arrives. Retrieving memory can take up to several hundred CPU cycles depending on where the data is first available in the memory subsystem.
- *Data dependencies* happen when the current pipeline stage has to wait for the result of the previous stage in order to proceed.
- *Control dependencies* are due to changes in the control flow, caused for example by a branch misprediction. The pipeline must stall until the branch is resolved and the address of the next instruction is known.
- *Resource contention* occurs when an instruction tries to use a microarchitecture component that is currently in use by another instruction, there-

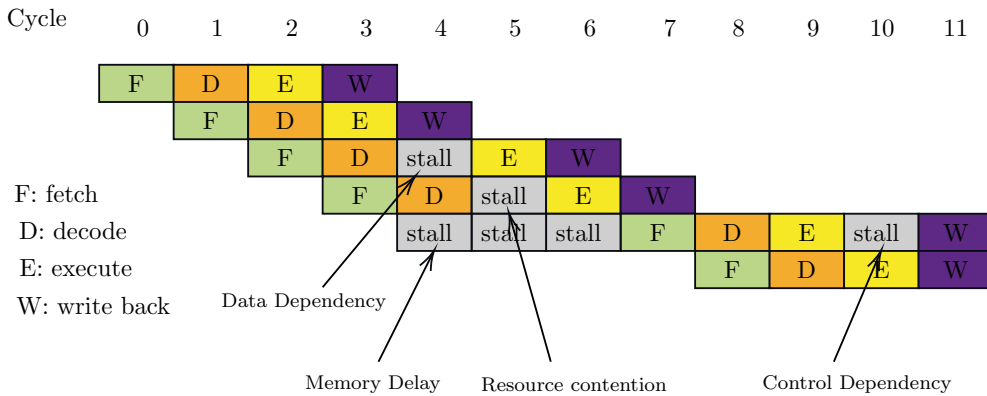


Figure 2.2 Example of a simple pipeline with stalls.

fore having to wait until the resource is freed.

Figure 2.2 shows a simplified four-stage pipeline breaking the process into fetch, decode, execute, and write back stages. Generally speaking, the stages operate in parallel but depend on each other, i.e., the output of a stage is the input for the following stage. This allows instructions to proceed to the next stage as soon as the current stage is done, however as mentioned previously, stalls occur from time to time. This simplified example hides complexity introduced by additional microarchitecture techniques and components that further increase performance. Relevant techniques that aim to improve the performance by better utilizing the resources and reducing the number and length of pipeline stalls include: out-of-order execution, branch prediction, simultaneous multithreading [69], and caching.

Out-of-order execution. In contrast to Figure 2.2 where it is assumed that instructions are executed in-order, i.e., instructions are executed in the order of the program, modern microprocessors perform out-of-order execution. Out-of-order execution operates based on data flow, allowing a program to execute instructions if the data and operands are available, even if previously issued instructions are still waiting for a resource. As long as each executing instructions depends on previous results, this technique permits to continue program execution while some of the instructions are waiting for stalls to be resolved, thus increasing overall program execution performance.

Branch prediction. Another common technique improving the CPU performance is branch prediction. Branch prediction allows a microprocessor to combine local and global history information to speculate the path that will be followed in a conditional branch, thus preemptively executing future instructions based on this prediction. Over the years, branch predictors have been implemented as static branch predictors [47], hybrid branch predictors [38], one-level branch predictors [115], two-level branch predictors [114], and recently more complex branch predictors using machine learning techniques [75, 76, 149]. State-of-the-art branch predictors are accurate. During a typical workflow, the steady-state accuracy, i.e., the peak performance of the branch predictor when is warm, is measured as less than 5 mispredictions per kilo-instruction [121, 134]. However, in the case of a misprediction, the pipeline is penalized with a longer stall, as the pipeline must be partly or fully flushed and the correct instructions must be fetched and executed from the resolved branch.

Simultaneous Multithreading. To greatly improve resource utilization, some microprocessors support simultaneous multithreading (SMT)—known as Hyper-Threading (HT) on Intel microprocessors—as part of their microarchitecture. As the name indicates, SMT allows to execute multiple threads belonging to different processes at the same time. SMT makes a single physical processor core to appear as multiple cores by copying the architecture state for each logical core [91]. In reality only a single set of physical resources exist, but the logical partition is transparent to user programs, thus different programs execute simultaneously by sharing and competing for execution resources. Chapter 3 describes a new technique that exploits resource contention as a side channel in processors supporting SMT.

Caching. Exploiting the locality of reference by either a single program or multiple programs, caching allows the microprocessor to store data and instructions that are frequently used in a small memory component that is quickly accessible. Compared to main memory, caches are physically closer to the CPU, hence they are faster to access, making them an excellent place to store data and instructions that are used throughout the execution of a program. This dissertation focuses on the caching mechanism as a side channel, enabling cache-

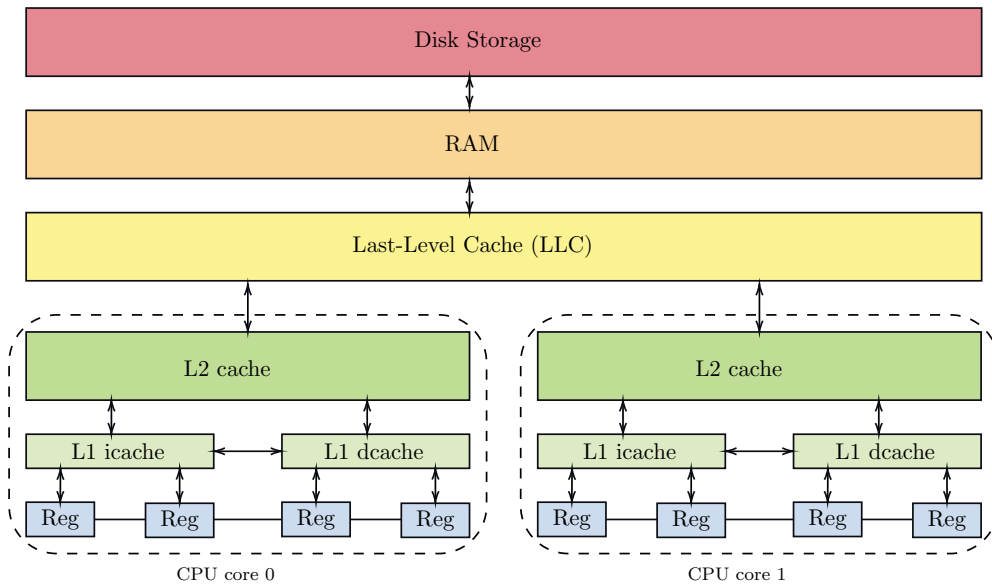


Figure 2.3 Memory hierarchy showing different memory levels color coded according to their access speed. Blue to the left is the fastest, and red to the right is the slowest.

timing attacks, therefore the rest of this chapter provides a detailed description of the caches and how they work.

2.1.2 The Cache Hierarchy

Microprocessors contain a set of registers capable of holding data during execution. These registers are able to match the microprocessor’s processing speed, however, due to the limited size and number of registers, combined with a slow data retrieving speed from the main memory, an intermediate hierarchy of caches was introduced to the memory subsystem in the microprocessor to alleviate this bottleneck.

Caches are fast memory components with an increasing speed, and inversely a decreasing size, as they get physically closer to the CPU. Figure 2.3 shows the typical memory hierarchy from fastest to slowest memory components: the CPU registers, L1 instruction and data caches, L2 cache, LLC, main memory, and disk storage. The first-level cache, also known as L1 cache, is comprised of two separate small caches with the fastest speed in the cache: an L1 data cache

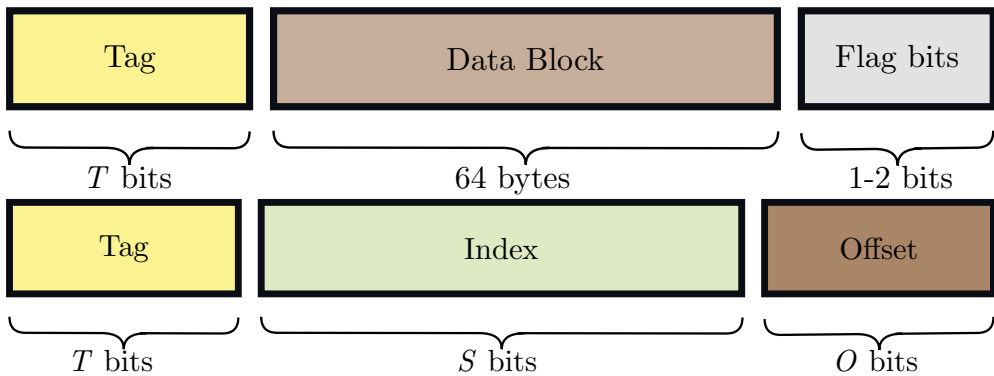


Figure 2.4 Top: 64-byte cache line structure; Bottom: memory address structure.

(dcache) and an L1 instruction cache (icache). The L1 cache is followed by a second-level cache (L2) of intermediate speed, which contains both data and instructions. Finally, the last-level cache (LLC) also contains both data and instructions, and it is slower but bigger than the previous levels. A set of L1 and L2 caches is commonly found per each CPU core, while the LLC is shared among all the CPU cores.

During program execution, caches exploit the locality of reference to improve the performance. At a high level, the locality of reference is divided in two: (i) *temporal locality* dictates that the same set of data and resources will be used within a short time interval; and (ii) *spatial locality* dictates that data stored relatively close to recently used data, will also be used. Thus, based on these observations, caches not only store recently accessed data, but also data close to it.

Typically, a cache is divided into cache lines of 64 bytes each. Each cache line holds an aligned block of adjacent bytes from memory, and when the CPU needs data from main memory, it first looks for it in the cache. If the CPU finds the data in the cache, it is said to be a *cache hit*. On the other hand, if the data is not found in the cache, it is said to be a *cache miss* and the search continues to the next level of memory hierarchy.

Caches read and write data from and to main memory using memory addresses, therefore each cache line is tagged based on the data address where it resides in main memory. Figure 2.4 shows the structures of a cache line (top) and a memory address (bottom). The cache line is composed of the tag, the

data block, and the flag bits. The tag allows to efficiently search for data when reading or writing in the cache, by simply finding for a matching tag instead of having to compare data blocks. The data block contains the actual data read or to be written to main memory, while the flag bits are bits allowing to mark cache lines as *invalid* or *dirty*—this last one used to indicate that the content of a cache line has changed since it was read from main memory, thus it needs to be written back to memory. Note that the cache line size is equivalent to the data block, and is the amount of useful data it holds, i.e., the size does not include the bits used for the tag, and the flags.

On the other hand, the memory address includes a tag, an index, and an offset. For a cache with 64-byte cache lines and 2^S cache sets, the low $\log_2(64) = 6 = O$ bits determine the offset within the cache line where the data resides. The next S bits in the memory address determine the cache set index in which the data can be found, leaving the top $T = 64 - S - O$ bits to form the tag, uniquely identifying a cache line within a cache set.

Cache Associativity. Caches can be commonly implemented in three different ways: direct mapped caches, fully associative caches, and set-associative caches. Most modern microprocessors implement set-associative caches since they offer a good balance between cache complexity and CPU performance. Set-associative caches are often referred to as *m-way* set-associative caches, as each data block can be stored in only one out of m cache lines in a given cache set. Depending on the implementation, caches use either virtual addresses, physical addresses, or a combination of both to compute the index and the tag. Figure 2.5 shows the typical workflow of an *m-way* set-associative cache using physical addresses. Given a memory address, the cache controller checks the existence of that data in the cache by taking the cache set index bits. Then, once the cache set has been located, the tag portion is checked simultaneously against all *m-ways*. If the data exists in the cache, the tag matches exactly one entry, and a *cache hit* is recorded. The data is obtained from the data block at the specific offset. If no tag is matched, a *cache miss* is reported and the data search continues to the next levels in the memory hierarchy. Eventually, the cache will be full, requiring data to be removed from the cache in order to make space for new entries. The processor determines the cache lines to be removed and kept using a cache replacement policy.

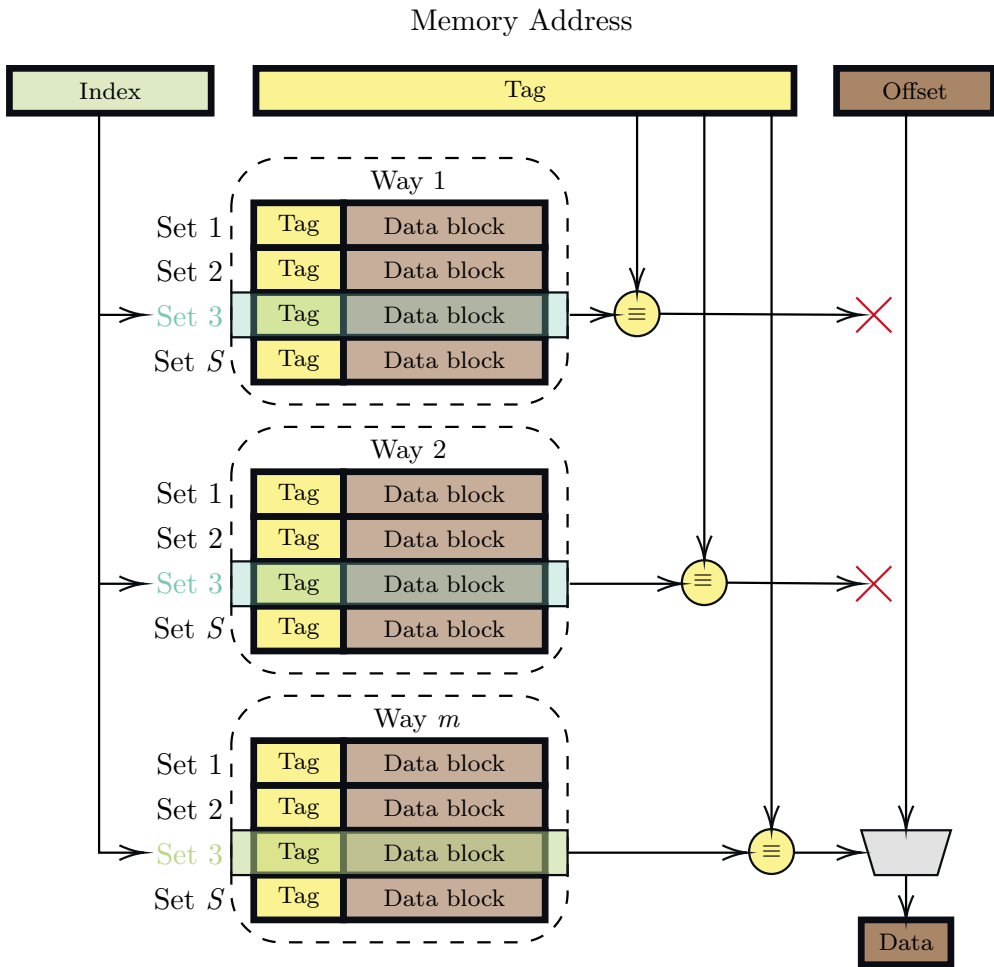


Figure 2.5 An m -way set-associative cache: the index selects the cache set, the tag uniquely identifies the data within the cache set, and the offset determines the data location in the data block.

Cache Replacement Policies. Whenever the cache is full and a new data block needs to be loaded, e.g., due to a cache miss, a cache replacement policy helps the processor determine which cache line to *evict* in order to make room for the new data block. Historically, the most common cache replacement policy used in Intel CPUs is an approximation to the Least Recently Used (LRU). In the LRU policy, each cache line has an associated age field used to select and evict the oldest cache line. This policy works particularly well for workloads where data blocks are often reused within a time period, however, a downside of

this policy is the need for extra metadata such as the aging field, in addition to increased implementation complexity. An alternative policy commonly used by the ARM Cortex-A processor family [85, 125, 126, 127] is the pseudo-random policy which does not make any assumptions about access patterns, and instead it evicts a random cache line when needed, hence requiring a simpler logic during implementation. Over the years many other policies have been proposed [74, 110], however the actual implementations are not disclosed by CPU vendors as they are considered intellectual property, thus leaving researchers with the task of understanding and reverse engineering them [1, 132, 133].

Inclusiveness. The cache hierarchy in modern Intel CPUs is *inclusive*, that is, all data stored in the smaller caches (L1 and L2) is also stored in the LLC. Similarly, if a cache line is *evicted* out of a cache, coherency must be enforced by evicting data out of all cache levels. As mentioned previously, the LLC is shared among all the CPU cores. This is done by splitting the LLC into smaller cache slices that are interconnected [70], allowing all the CPU cores to access all the slices.

In contrast to *inclusive* caches, modern AMD CPUs can be either *exclusive* or *non-inclusive*. While an *exclusive* cache hierarchy allows data to be stored in exactly one cache at a given time, the *non-inclusive* cache hierarchy permits data to exist in one or several cache levels at a given time, without strictly enforcing data coherency.

2.2 Cache-Timing Attacks and Techniques

In addition to being transparent to the programmer, the microarchitecture hides information in its internal state, which can be observed through the execution time of a program utilizing its resources. Attacks exploiting the hidden state of the microarchitecture to leak confidential information are termed microarchitectural attacks. Cache-timing attacks are a subset of microarchitectural attacks abusing cache contention in different cache levels to exfiltrate confidential information exposed by the cache internal state during the execution of cryptographic operations. Historically, cryptography has been the preferred target since it is the most cost-effective—recovering a single key gives, potentially, access to past, present, and future secrets.

In contrast to traditional EM and power side-channel attacks, requiring specialized hardware tools and close proximity, cache-timing attacks are exploited remotely from software, thus these do not require physical access to the target. Typically, the threat scenario for cache-timing attacks—and many other microarchitectural attacks—is an attacker with unprivileged code execution on the target machine while being co-located with the victim. Co-location with the victim can be either in the same physical CPU core, i.e., cross-thread attacks in systems supporting SMT, or in a different CPU core within the microprocessor, i.e., cross-core attacks.

In order to successfully perform a cache-timing attack on a given cryptographic implementation, an attacker not only requires detailed knowledge of the inner workings of the cryptosystem, but it also must carefully choose the cache-timing attack technique to use. Choosing the correct technique makes the difference between a successful and a failed attack, therefore declaring an implementation secure or vulnerable with respect to cache-timing attacks.

Cache-timing attack techniques are a variety of software techniques exploiting the cache internal state exposed through timing differences introduced by the cache hierarchy, and they are a building block for cache-timing attacks. These techniques cleverly execute unprivileged system instructions, abusing the cache replacement policies and other cache features to break the isolation mechanisms preventing unrelated programs and processes from communicating with each other, thus creating a covert channel. A natural extension to covert

channels is to use this communication channel by a malicious process to spy or exfiltrate information from a victim process, i.e., the victim process unknowingly communicates confidential information through its cache footprint, while the malicious process actively listens for this communication.

Cache-timing attack techniques rely on the spy process modifying and monitoring the cache state during the execution of a cryptographic operation by the victim process. If the cache footprint of the victim’s cryptographic process is dependent on confidential information such as the private key, the spy process can potentially recover the key. The following paragraphs present a non-exhaustive list of the most notable, state-of-the-art, cache-timing attack techniques developed over the past decades. These techniques enable a variety of cache-timing attacks at different cache levels in the cache hierarchy.

EVICT+TIME. First described by Osvik, Shamir and Tromer [103] in 2006, the authors used this technique to demonstrate a successful attack against the commonly used AES T-table as implemented in OpenSSL 0.9.8. The goal of an attacker using this technique is to modify the cache state prior to a cryptographic operation by evicting specific cache-sets, then the attacker measures the execution time of said operation to detect differences in execution time caused by cache-set evictions. After repeating the process several times, an attacker is able to recover private key bits by using algebraic methods on the timing information. The timing difference is due to cache-hits and cache-misses, induced by successfully evicting content from cache-sets which are mapped to cache-sets used by the cryptographic operation. Figure 2.6 illustrates the attack flow of this technique. The authors note that this technique assumes the attacker has the ability to distinguish the beginning and the end of an encryption, as well as the ability to trigger operations at will. EVICT+TIME has a cache-set granularity, and greatly suffers from noise caused by other processes running in parallel, as well as cache thrashing.

Related work using EVICT+TIME includes attacks against OpenSSL AES implementation executing in ARM microprocessors [85, 126], as well as defeating address space layout randomization (ASLR) [49, 66].

PRIME+PROBE. In the same previous work, Osvik, Shamir and Tromer [103] propose a second attack technique. The goal of an attacker using this technique

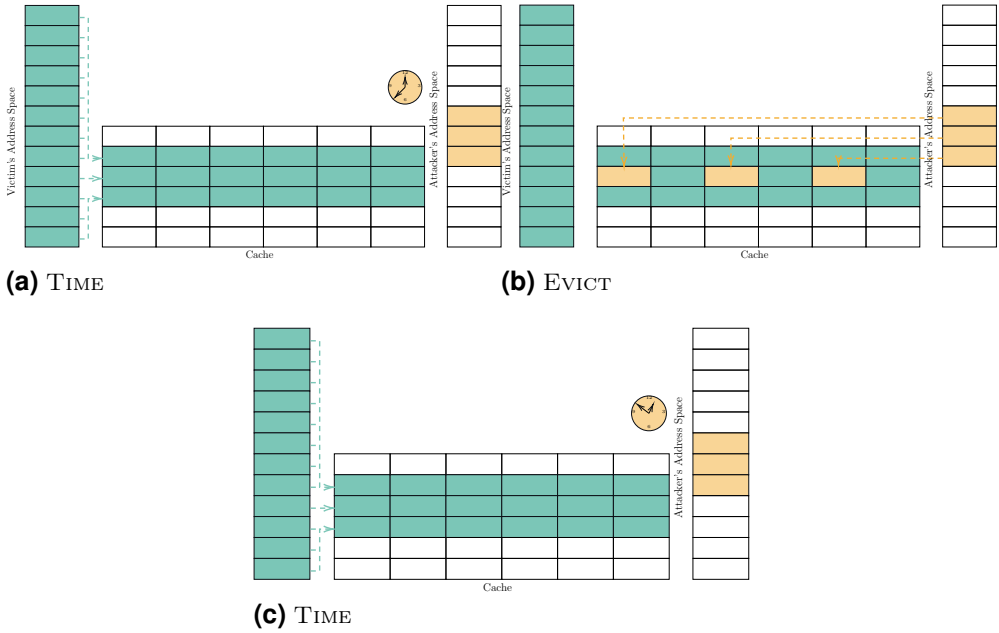


Figure 2.6 The EVICT+TIME technique: (a) attacker times operation to obtain a baseline; (b) attacker partially evicts the cache, e.g., a cache-set; (c) attacker times and compares.

is to continuously bring the cache to a known state by filling it with its own data (prime), then timing the loading of its own data (probe) while the cryptographic operation takes place. Similar to the previous technique, both processes will likely share some cache sets, thus the attacker will obtain information about the cryptographic operation being executed due to timing differences during the loading of its own data. In this technique the time measurement is done per cache set, thus produces several measurements for a number of cache sets during a single cryptographic operation, i.e., the attacker gets continuous snapshots of several cache sets during the execution of a cryptographic operation. Figure 2.7 shows the typical attack flow and an example trace. The example trace shows the victim accesses contiguous cache sets corresponding to table look-ups during the cryptographic operation.

While still requiring a shared and an inclusive cache, PRIME+PROBE is a generic and powerful technique that is ISA-independent and adapts to different replacement policies. It can be used to attack several cryptographic primi-

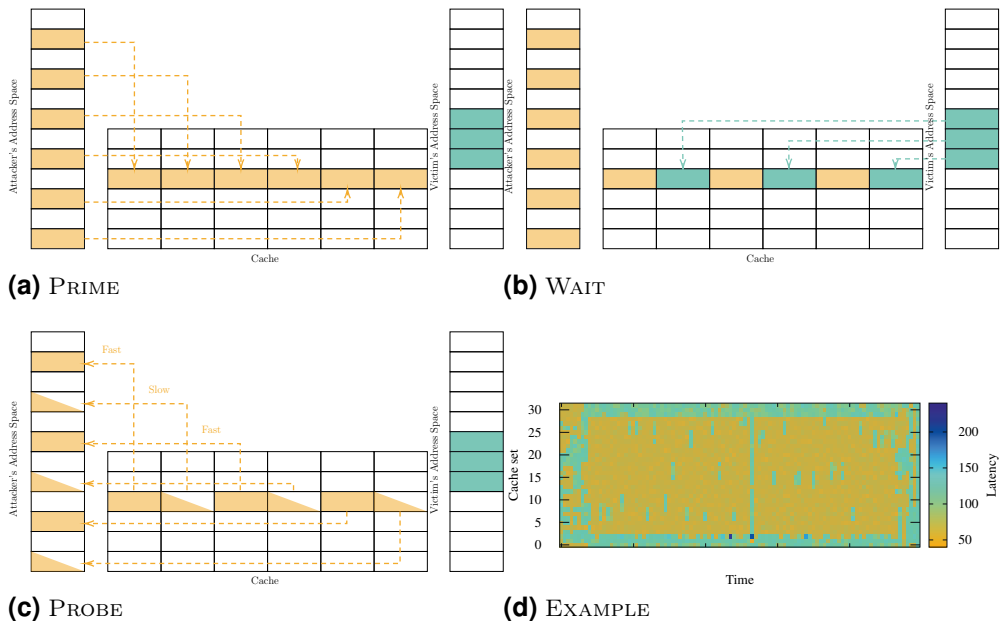


Figure 2.7 The PRIME+PROBE technique: (a) attacker primes some cache-sets by loading its own data; (b) attacker waits for the victim to evict its data; (c) attacker probes own data by loading and timing it; (d) example trace probing 32 cache sets during DSA modular exponentiation revealing multiplier look-ups.

tives in a wide variety of attack scenarios, thus making it, arguably, the most versatile of the cache-timing attack techniques. PRIME+PROBE has been applied to a wide range of cryptosystems in different scenarios, well beyond the original attack against the AES T-table implementation in OpenSSL. Some of these include attacks against different cryptographic primitives, targeting the L1 icache and dcache [3, 4, 30, 103], and the LLC [89, 92, 93], on scenarios such as the browser [102], cloud [68, 71, 112, 150], SGX [117], and ARM-based mobile devices [85].

FLUSH+RELOAD. First described by [59] and later popularized by [145], this technique targets the LLC and can detect memory access at a cache line level, thus giving a finer granularity, higher accuracy, and higher signal-to-noise ratio when compared to the previous techniques. As the name implies, to accurately achieve data eviction from the cache, this technique uses the `clflush` instruction introduced by Intel with the Streaming SIMD Extensions 2 (SSE2).

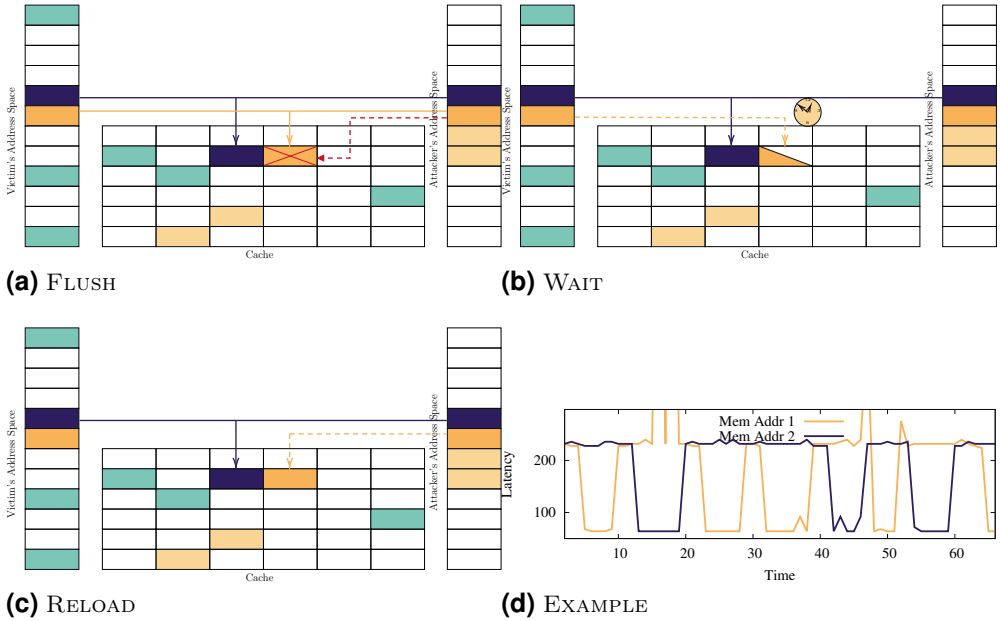


Figure 2.8 The FLUSH+RELOAD technique: (a) attacker flushes a memory address shared with the victim out of the cache; (b) attacker waits for some time; (c) attacker reloads memory address and times it; (d) example trace probing 2 cache-lines accessed by the victim revealing a sequence of operations.

Due to the inclusive caches in Intel CPUs, this means that the target data gets evicted from all the cache levels, including the LLC.

The goal of an attacker using the FLUSH+RELOAD technique is to continuously evict data out of a specific cache line using the `clflush` command, then measuring the time it takes to reload that same data after a small waiting time. The attacker can identify if the victim process accessed the shared data during the time between the flush and the reload operations just by looking at the timing information. If the victim process accessed the data, it will be available in the cache, resulting in a cache hit on the next reload operation, taking a short amount of time. On the other hand, if the victim process did not access the shared data, then it is not available in the cache, resulting in a cache miss on the next reload operation, therefore taking a longer time as the data must be brought to the cache from a more distant memory, i.e., RAM, or main memory. Figure 2.8 illustrates the attack flow for the FLUSH+RELOAD technique, as well as an example trace obtained after probing two memory addresses.

The authors describe three requirements when using the FLUSH+RELOAD technique:

- Access to the `clflush` instruction. In Intel CPUs, `clflush` is an unprivileged instruction, thus an attacker does not require special permission to execute this instruction while applying this technique. However, other vendor CPUs such as ARM may require special permissions to execute an instruction equivalent to `clflush`, although some results suggest that this instruction is widely used by developers and therefore some microprocessors used in mobile devices do not strictly enforce the permission [85].
- Shared data. The attacker and victim processes must share the data that is evicted out and reloaded into the cache. This is typically achieved through the use of shared libraries or page de-duplication [18, 135].
- *Inclusive* caches. The microprocessor’s cache must be *inclusive*, as this ensures the LLC contains copies of the data located in lower levels of cache and they must be synchronized. This has the added benefit that it allows to easily differentiate between a cache miss and a cache hit as the timing difference is longer. As mentioned previously, most of Intel CPUs have *inclusive* caches, making them vulnerable to FLUSH+RELOAD and other cross-core cache-timing attack techniques. Despite newer Intel CPUs shipping with *non-inclusive* caches, research suggest that their microprocessors are still vulnerable to these attack techniques [143]. On the other hand, AMD CPUs have *non-inclusive* caches, and it is widely accepted that they are not vulnerable to FLUSH+RELOAD, as originally reported in [145], however, [73, 86] demonstrated similar attack techniques for cross-processor attacks and cross-thread attacks, respectively.

Other Techniques. The power of PRIME+PROBE and FLUSH+RELOAD has led to the development of attack technique variants adapted to different technologies, constraints, and attack scenarios. Some relevant variants are the following.

In situations where the microarchitecture does not expose a `clflush` instruction, or the `clflush` equivalent instruction supported by the CPU requires

special privileges to execute, `EVICT+RELOAD` [55] can be used. This technique replaces the `clflush` (or equivalent) instruction with other efficient and accurate methods to evict the cache—similar to the `PRIME+PROBE` technique.

Similarly, in Intel CPUs supporting the Transactional Synchronization Extension (TSX) instructions the `PRIME+ABORT` technique [43, 52] can be used for noiseless traces without the need for a high-resolution timer. With this technique, an attacker loads cache-lines as TSX transactions, thus accurately detecting when the victim accesses these cache-lines due to the abortion alerts generated by the TSX mechanism.

Finally, the `FLUSH+FLUSH` technique based on `FLUSH+RELOAD` is particularly useful in scenarios where speed and stealthiness are required. This technique does not perform the reload step of the classic `FLUSH+RELOAD`, but instead, it exploits the timing difference of the `clflush`. The execution time of `clflush` depends on data availability in the cache, completely avoiding memory accesses, and therefore, this technique can bypass detection techniques relying on cache-hit and cache-miss counters reported by the CPU to privileged processes [54].

Cache Eviction. Typically, an attacker using `EVICT+TIME`, `PRIME+PROBE`, or any other cache-timing attack technique requiring memory eviction from the cache, faces two main obstacles: (i) finding eviction sets, i.e., memory addresses that map to the same cache sets used by the victim process during the cryptographic operation; and (ii) finding the correct eviction strategy, i.e., the access sequence for the eviction sets resulting in the successful eviction of the victim’s cryptographic data out of the cache. Data is stored in a given cache set based on a combination of physical memory address bits. In L1 caches, physical and virtual addresses share the same LSBs used to determine the cache set in which data is stored. Given the small size of L1 caches, it is feasible to access all cache sets to achieve eviction without requiring a particular eviction set or eviction strategy. However, targeting the bigger LLC complicates eviction as additional factors must be taken into consideration. First of all, in some cases physical and virtual memory addresses do not completely share the bits determining the cache set where data is stored, therefore creating an eviction set is a non-trivial problem. Additionally, starting with the Sandy Bridge architecture, Intel introduced cache slices to the LLC, obscuring the mapping of physical ad-

dresses into cache sets in cache slices. As mentioned in Section 2.1.2, to defeat this issue, recent work focuses on reverse-engineering this undocumented feature using timing measurements [89] and performance counters [72, 93]. Finally, also briefly mentioned in Section 2.1.2, CPU vendors have improved their cache replacement policies, complicating the task of evicting data reliably out of the cache. Nevertheless, finding new eviction strategies and methods to optimize these strategies is an ongoing research work [29, 53, 132, 133].

Performance Degradation. An additional challenge when performing a cache-timing attack is to convert the leakage signal into usable information, e.g., a bit pattern, a sequence of operations, a sequence of table look-ups, etc. An attacker must try to acquire the best possible signal before applying any signal processing methods in order to reduce the number of errors, and consequently reduce the total computational load necessary to fully recover keys. To that end, performance degradation techniques were developed as a set of techniques used to slow down the execution of a process. Unlike physical side-channel attacks where the attacker can control the oscilloscope to increase or decrease the probing frequency according to its specifications, in cache-timing attacks the attacker has less control of the probing frequency, as it depends on several microarchitecture features, and the maximum probing frequency is bound by the CPU frequency.

Historically, performance degradation attacks were categorized as denial-of-service (DOS) attacks, where the goal was to annoy the victim process by reducing the quantity or quality of the process [51, 61], but with no real benefit for the attacker. However, Allan et al. [14] demonstrated that performance degradation attacks can be used to amplify the side-channel leakage, thus greatly improving the success rate of cache-timing attacks.

In the context of cache-timing attacks, a performance degradation attack is typically paired with a cache-timing attack technique such as FLUSH+RELOAD, both targeting the same cryptographic operation but different memory addresses. While the cache-timing attack technique probes data or instructions in memory, the performance degradation attack slows down the execution of the cryptographic operation enough to successfully probe data or instructions in memory that otherwise would be missed due to a rapid execution.

attack target as its widely used. Any research resulting in new features or vulnerabilities represent a deep real-world impact. This is especially true for SCA and side-channel attacks targeting actual implementations of several different cryptographic primitives—of which OpenSSL has plenty.

The rest of this section recapitulates part of the troubled history of OpenSSL with respect to SCA, applied to asymmetric cryptography, namely RSA, DSA, and ECC. Figure 2.9 shows a timeline of the most relevant SCA attacks and countermeasures adopted in OpenSSL throughout the years. The list of events is non-exhaustive and it includes only those related to the `BN_FLG_CONSTTIME` flag, one of the main themes of this dissertation. The following summary is essential to understand the research work presented on Chapter 3. It includes events starting in 2003, and stretches all the way to the start of the research work leading to this dissertation in 2016.

2003. The theory and practice of side-channel attacks reached a new level when a timing attack was successfully demonstrated against general purpose PCs and servers instead of the smaller and specialized smart cards—a more common target at that time. Against common belief, D. Brumley and Boneh [33] showed that remote timing attacks are practical by demonstrating an attack against an Apache server powered by OpenSSL. This attack targeted timing variations in RSA decryption during TLS handshakes, thus effectively attacking from the protocol level.

Through this work, the authors demonstrated that SCA represented a real threat to the security of a growing number of web servers, and they not only proved that it was possible to recover secret keys from a general purpose server but also that it was possible to do it in a completely remote scenario and without specialized equipment. The authors noted that RSA decryption followed the CRT method, a method that supposedly prevented side-channel attacks. The CRT method itself used the *sliding window* modular exponentiation function, which in turn used two distinct algorithms, i.e., textbook vs. Karatsuba, to compute a multi-precision multiplication. The choice of algorithm depended on the size of the input values, and each algorithm had a different running time, thus leaking input size information. Additionally, the Montgomery reduction function used for `BIGNUM` modular reduction also performed an extra reduction step based on the size of the input value. By combining these observations, the

authors managed to recover enough bits of the RSA prime factors, ultimately leading to full key recovery. The authors concluded that RSA blinding is an effective countermeasure against this vulnerability. In fact, this countermeasure was already implemented in OpenSSL but not active by default due to the performance impact on RSA. As a result of this work RSA blinding was set by default² in OpenSSL with the option to turn this off by using the new flag `RSA_FLAG_NO_BLINDING`.

As a side-note, despite not being a cache-timing attack, this work is relevant to this dissertation since it represents the first published instance where a side-channel attack is enabled by following a performance-first and insecure-by-default approach in OpenSSL, a recurring issue and central theme of this dissertation.

2005. Percival [105] not only described a new cache-timing attack technique, but also showed its power by using this technique against OpenSSL RSA digital signing to steal private keys. As part of this work, the author describes the cache behavior and its possible use as a covert channel, then demonstrates the impact of the cache-timing attack by targeting OpenSSL. Interestingly, the target is the same *sliding window* modular exponentiation function previously exploited by [33]. However, this time the cache footprint difference between the square (`BN_sqr`) and multiplication (`BN_mul`) functions is what reveals enough bits of information to ultimately factor the RSA modulus and recover the private key. To respond to this work, OpenSSL introduced two important changes to its code base³: a brand new `BN_FLG_EXP_CONSTTIME` flag, and a *fixed window* modular exponentiation function secure against cache-timing attacks named `BN_mod_exp_mont_consttime`. On the one hand, the newly created `BN_mod_exp_mont_consttime` function combined fixed windows and a special memory layout for the pre-computation table to prevent data-dependent accesses, thus avoiding leaking confidential information. On the other hand, the `BN_FLG_EXP_CONSTTIME` allowed to mark security sensitive integer values, e.g., private keys, secret exponents, and nonce values, as secrets. By flagging the secret values OpenSSL changes its execution flow, therefore, instead of executing the vulnerable *sliding window* function, the execution flow changes to the new,

²Commit: 96c15b8aad15e0cb3d107ac281be215ce04241d8

³Commit: ecb1445ce2df51e5310bb58c67c57f7f83ed6a52

secure, and data-independent *fixed window* modular exponentiation function. In addition to RSA, the OpenSSL team made the necessary changes in DH and DSA to provide the same level of protection as to RSA for the modular exponentiation operation.

2007. Continuing the previous trend, Aciğmez, Gueron and Seifert [5] focused on the RSA implementation in OpenSSL highlighting new SCA issues by exploiting a microarchitecture component not targeted before, the Branch Prediction Unit (BPU). Using Simple Branch Prediction Analysis (SBPA), the authors uncovered an implementation flaw leaking information from the previous *fixed window* modular exponentiation function—theoretically immune to SCA. The flaw was present during the construction of the *window value*, which was done by scanning the exponent bit by bit, thus using SBPA, a malicious attacker could easily retrieve the complete exponent value. Additionally, this work highlighted for the first time the insecurity of using the Binary Extended Euclidean Algorithm (BEEA) function `BN_mod_inverse` for modular inversion of secret inputs. CRT-RSA used this function to compute four secret values: (i) the private exponent $d = e^{-1} \bmod (p-1)(q-1)$; (ii) the CRT parameter $q^{-1} \bmod p$; (iii) the Montgomery constant $-p^{-1} \bmod m$; (iv) and the blinding pair $(x, y = x^{-e} \bmod N)$. OpenSSL introduced more changes to its code base based on these results⁴: (1) a new and branch-free modular inversion function (`BN_mod_inverse_no_branch`) immune against SBPA; (2) a new and (almost) branch-free multi-precision division function (`BN_div_no_branch`) function; (3) the vulnerable *fixed window* modular exponentiation function was fixed; and (4) the `BN_FLG_EXP_CONSTTIME` flag was replaced by a newer `BN_FLG_CONSTTIME` flag. The flag update reflected the increasing need to support more SCA-secure operations within the library, i.e., modular exponentiations and modular inversions. Following a similar approach as before, a flag check was introduced at the top of the `BN_mod_inverse` function to determine which execution flow to follow. If the input values are marked with the new flag the execution flow jumps to the new `BN_mod_inverse_no_branch` function, otherwise, it continues with the old SCA-vulnerable `BN_mod_inverse` function.

Unfortunately, these changes strengthened the architecture decision to continue with performant but SCA-vulnerable functions by default, thinking that

⁴Commit: 7cdb81582cafdddce891f1da8d85ca372e5dabbc

only a handful of places in the code would require an SCA-secure execution flow.

2008. Five years after being first mentioned in [33], Aciicmez and Schindler [3] noticed that OpenSSL Montgomery multiplication function was never fixed, and it was still leaking information due to the extra reduction step happening based on the input values. The authors mounted an L1 instruction cache-timing attack by embedding a spy process into OpenSSL. This was done to reduce the cache noise produced by other processes, but successfully exploiting the instruction cache against RSA. Previous to this attack, cache-timing attacks were (mostly) performed by exploiting the L1 data cache, but the authors demonstrated the feasibility against the L1 instruction cache. The OpenSSL team adopted the proposed changes⁵ to finally remove the extra reduction operation in the Montgomery multiplication function.

2009. Up to this point, previous SCA was mostly focused on attacking RSA because, arguably, it was the most common and widely used cryptosystem for data encryption and digital signing. Moreover, given that RSA shares many low level operations with other cryptosystems, many of the side-channel countermeasures previously applied to RSA cascaded down to DH and DSA, allowing them to enjoy some level of protection against SCA. However, ECC and especially ECDSA, used a different set of algorithms to implement EC operations, thus requiring a separate analysis. ECC already was widely used by security applications relying on OpenSSL as the cryptography provider, thus the arena was set for practical SCA of OpenSSL ECC implementations. [30] demonstrated a practical cache-timing attack against ECDSA using the PRIME+PROBE technique on the L1 data cache. The vulnerable function was the multi-scalar multiplication algorithm falling back to the textbook scalar multiplication with a modified *wNAF* scalar encoding. This function is the ECC equivalent to the *sliding window* modular exponentiation, thus the leaked information was similar to that previously obtained by [105], but this time applied to ECDSA. Another novelty of this work was the usage of a standalone spy process, instead of the more common spy process embedded in the library code. The standalone spy process ran in parallel to the ECDSA computation in a PC

⁵Commit: 1a56614af2b015fdb79fa1b6df56820d08110523

supporting SMT. While the victim process computed ECDSA, the spy process was able to take snapshots of the L1 dcache state, effectively extracting the sequence of double and add operations used to compute the scalar multiplication. By having a standalone spy process, the noise captured by the spy process increased considerably compared to the embedded approach, thus the authors used statistical methods to reduce the noise.

A benefit of attacking ECDSA over RSA is that the former benefits from lattice methods [65], allowing an attacker to fully recover a private key by observing a small leakage equivalent to only a few bits over several signatures signed with the same private key. Among the proposed countermeasures to mitigate this attack was the scalar blinding approach, similar to the exponent blinding technique proposed and deployed earlier to protect the RSA cryptosystem. However no changes were introduced to OpenSSL as a result of this work.

2010. Aciçmez, B. B. Brumley and Grabher [4] followed similar methods as in the previous attack with two main differences: (i) the target cryptosystem was DSA; and the target cache was the L1 instruction cache. The authors use a standalone spy process running in parallel to DSA taking snapshots of the L1 instruction cache using the PRIME+PROBE cache-timing attack technique. An interesting observation from this work is that DSA used the *sliding window* exponentiation function to compute modular exponentiation instead of the more secure `BN_mod_exp_mont_consttime` function previously introduced to the code base and already in use by RSA [105]. This exponentiation function revealed the sequence of square and multiply operations performed during DSA signature generation. The sequence partially leaked secret nonce bits over several signatures through the L1 instruction cache. Same as in [32], the authors collected the leakage over several signatures, then later applying lattice methods in order to fully recover the private key. In addition to the successful attack on DSA, the authors analyzed multiple countermeasures, focusing on system level countermeasures. The analysis included cache flushing, cache disabling, and turning off SMT support. While the source of the leakage was clearly understood, no additional analysis was performed to understand the reason why DSA was still using the SCA-vulnerable *sliding window* exponentiation function with secret exponents despite the `BN_mod_exp_mont_consttime` function being

introduced 5 years previous to this work. Clearly, the SCA countermeasures were not uniformly applied to all the cryptosystems, and the complexity (and technical debt) of the `BN_FLG_CONSTTIME` flag was starting to show. OpenSSL did not introduce any new changes to its code base following this work.

2011. Analogous to [33], B. B. Brumley and Tuveri [32] demonstrated that remote timing attacks were still possible, but this time against ECC using curves over binary fields. Binary curves in OpenSSL used the faster and SCA-secure Montgomery ladder algorithm⁶ for scalar multiplication instead of the common *wNAF* method used by prime curves. The most interesting aspect of this attack is the feasibility of performing a timing attack even if the efficient and SCA-secure Montgomery ladder algorithm was used to compute the scalar multiplication operation. The authors noted that the bit length of the secret nonce was being leaked by the loop bounding the iteration count during scalar multiplication, i.e., the number of loop iterations was directly related to the bit length of the secret nonce, thus, by timing the signature generation computation the authors were able to distinguish bit length information for several secret nonces. Interestingly, the regularity of the Montgomery ladder implementation considerably improved the timing leakage, resulting in a clear signal, allowing a protocol level attack on TLS. The authors repeatedly captured timing signals generated during TLS connection establishment, ultimately allowing them to fully recover the server's private key after applying lattice methods to the partial bit leakage collected over many TLS connections. As a countermeasure to this issue, the authors proposed⁷ to fix the nonce bit length to match the bit length of the order of the group in which the ECDSA curve operates, thus fixing the number of iterations performed during scalar multiplication and closing the source of leakage.

2012. As ECC became mainstream and more widely deployed on the Internet, new research results started to permeate into OpenSSL. Käsper [78] presented faster implementations of multiple elliptic curves⁸. These fast implementations also performed scalar multiplication in an SCA-secure manner by combining

⁶Commit: 7793f30e09c104b209206608a20f2088b1b635fd

⁷Commit: 992bdde62d2eea57bb85935a0c1a0ef0ca59b3da

⁸Commit: 3e00b4c9db42818c621f609e70569c7d9ae85717

fixed window combing and secure cache table look-ups using software masking. These results were applied not only to NIST P-224, but also to NIST P-256, and NIST P-521, thus increasing considerably their performance and security in OpenSSL. However, not all of the elliptic curves enjoyed the improvements resulting from this work, therefore the academic research switched to attack the `secp256k1` curve as it still used the generic and SCA-vulnerable *wNAF* scalar multiplication function.

2014. Although the Montgomery ladder is considered an SCA-secure scalar multiplication function due to its regular behavior and balanced sequence of operations, Yarom and Benger [144] demonstrated that small details can make or break the practical security of a Montgomery ladder implementation. Using the new FLUSH+RELOAD cache-timing attack technique, the authors demonstrated that an implementation that was considered secure against SCA, was no longer secure, as previous threat models failed to consider new state-of-the-art techniques. The FLUSH+RELOAD technique opened new opportunities for SCA research, as it offers a finer granularity to the spy process when compared to the previous PRIME+PROBE technique, i.e., FLUSH+RELOAD offers a cache-line size (64 byte) granularity allowing to pinpoint executing, and thus cached, instructions by the victim process. This impressive level of granularity allowed them to detect the branch taken inside the main loop of the Montgomery ladder scalar multiplication function, thus revealing each bit of the secret scalar value in an otherwise balanced algorithm. The authors demonstrated the FLUSH+RELOAD attack against ECDSA over the binary curve NIST B-571, arguably due to bigger key sizes facilitating the probing of the spy process, and the attack overall. In contrast to previous ECDSA attacks, the full key recovery is achieved without the use of lattice methods, and instead, the computationally friendlier Baby-Step-Giant-Step (BSGS) algorithm [122] is used to recover the secret nonce and ultimately the private key. As part of their work, the authors disclosed these results to OpenSSL and provided a patch⁹ to improve the Montgomery ladder implemented for binary curves.

Following the previous work, Benger et al. [23] combined the FLUSH+RELOAD technique with lattice methods and applied it to ECDSA to recover private keys of the `secp256k1` curve. As mentioned previously, this curve became

⁹Commit: 2198be3483259de374f91e57d247d0fc667aef29

the new target for research as it used the SCA-vulnerable *wNAF* scalar multiplication function. The main contribution of this work is the improved usage of leaked information during the lattice phase, allowing an attacker to recover private keys after observing only 200 signatures, the lowest number so far. The work highlighted yet again the insecurity of the *wNAF* function, but due to limited practical usage of the `secp256k1` curve, no fixes were proposed nor adopted to improve this particular curve.

2015. Gueron and Krasnov [57] studied and implemented software optimizations for ECC using 256-bit primes, e.g., NIST P-256. This work prompted two important changes to OpenSSL for ECDSA: a brand new `EC_GFp_nistz256_method` implementation for the NIST P-256 curve taking advantage of the performance gain offered by Intel AVX2 instructions; and a modular inversion via `BN_mod_exp_mont_consttime` and Fermat's Little Theorem (FLT)¹⁰. The new changes pushed the NIST P-256 implementation to a new level in terms of speed and SCA-security with an overall speedup factor of 2.0x compared to the previous fastest implementation [78]. This new implementation had the unintended consequence of fragmenting the ECDSA implementations for different elliptic curves. On the one hand, NIST P-256 enjoyed the fastest specialized implementation in OpenSSL, the NIST P-224 and NIST P-521 curves still benefited from [78], and P-384 did not enjoy any optimizations. On the other hand, the introduction of FLT to compute the modular inversion meant that modular inversion was possible through the SCA-vulnerable function `BN_mod_inverse`, and the SCA-secure functions `BN_mod_inverse_no_branch` and `BN_mod_exp_mont_consttime`. The flexibility offered by OpenSSL with its diverse implementations and configurations started to become equally demanding for users and developers, increasing even more the maintenance complexity, and with more SCA-vulnerable options by default.

2016. The FLUSH+RELOAD technique proved to be useful for new cache-timing attacks as it provided finer granularity to accurately probe instructions used by the victim process, allowing to attack implementations that were previously considered secure such as in [144]. Based on the same concept as FLUSH+RELOAD, Allan et al. [14] developed a new side-channel amplification technique

¹⁰Commit: `8aed2a7548362e88e84a7feb795a3a97e8395008`

using a performance degradation approach. This technique allowed them to slow down the execution of the victim process, thus giving more time to the spy process to accurately detect cache hits and misses when probing the instructions of interest. The slowdown permitted the authors to collect bit leakage from the ECC point inversion operation during *wNAF* scalar multiplication for the `secp256k1` curve. The *wNAF* method was known to be vulnerable, however this work improved the side-channel information leaked from the implementation, leading to full key recovery after observing only 6 signatures.

Continuing the work on cache-timing attack techniques, Yarom, Genkin and Heninger [147] developed a new technique giving a finer granularity than the FLUSH+RELOAD technique. The *fixed window* modular exponentiation function introduced 10 years before as a result of [105] was not completely secure against side-channel attacks, it was intended to avoid secret-dependent memory accesses detectable at a cache line level, thus leaving a very small and subtle timing leakage. Prior to this work, for 2048-bit and 4096-bit keys in OpenSSL, the *fixed window* exponentiation function used a window size of 5, using 32 multipliers. Each multiplier was divided in 8-byte fragments, each fragment scattered across groups of four cache lines, therefore, during a multiplication, all the cache lines containing the multipliers were accessed and a mask was used to correctly select the fragments of the required multiplier. The leakage was due to timing variations introduced by cache-bank collisions when accessing multiplier fragments located at the same offset in each group of four cache lines. The timing variations leaked the multiplier accessed, which was dependent on the private exponent, potentially revealing the private key. Yarom, Genkin and Heninger [147] demonstrated that this minor timing leakage believed to be impossible to exploit was exploitable with their newly developed CACHEBLEED technique. They applied this technique to the *fixed window* exponentiation function in the context of RSA, and they managed to recover 3 bits for most of the 5-bit windows, leading to full private key recovery after applying a branch-and-prune algorithm based on work by [64, 67]. To alleviate this issue, the OpenSSL team changed the size of the memory accesses to 16 bytes (instead of 8 bytes), in combination with access to a variable offset for each group of four cache lines¹¹. The authors noted that this countermeasure is a band-aid to

¹¹Commit: `d6482a82bc2228327aa4ba98aeec9979542a31`

the algorithm implementation, as more granular and powerful attacks could be developed in the future, rendering the solution useless. Their recommendation is a new constant-time modular exponentiation function which is still yet to be implemented.

3 RESULTS

This chapter describes the work and results derived from Publications I–VII, and discusses the defects enabling the side-channel attacks performed for each publication by tracing back the history of OpenSSL introduced in Section 2.3 and the insecure-by-default `BN_FLG_CONSTTIME` flag. This dissertation criticizes the `BN_FLG_CONSTTIME` flag as an architecture decision taken by the OpenSSL team defaulting to an insecure behavior, and instead favoring performance over security. In hindsight this poor choice is obvious now, but the criticism is done in an attempt to ultimately lead OpenSSL to a secure-by-default approach across the library due to the importance of OpenSSL as a research tool and as a cryptography library powering a large portion of the Internet. Finally, despite the criticism, big parts of this dissertation would not be possible without the existence of the `BN_FLG_CONSTTIME` flag.

3.1 Make Sure DSA Signatures are closer to Constant-Time

Publication I [107] brings attention to the fragile flagging mechanism used as an SCA countermeasure. Looking in retrospective, the first warning signs of the costly architecture decision of maintaining the new `BN_FLG_CONSTTIME` flag go back to 2007 just after it was introduced¹. The flag maintenance involved correctly tracking all the places in the code base dealing with secret inputs where the flag needed to be set, and correctly setting the flag in any of the secret inputs before calling any of the functions supporting the flag check.

Publication I discovers that two independent and seemingly unrelated changes meant to mitigate two different side-channel attacks in OpenSSL instead intro-

¹Commit: `2ac061e487e402a1d5abde866322c47550fc9186`

duced a software defect. The software defect resulted in the improper setting of the `BN_FLG_CONSTTIME` flag in secret nonce values prior to the execution of modular exponentiation operations. Recall that as a result of [105], the `BN_mod_exp_mont_consttime` function was introduced as an SCA-secure operation during signature generation for RSA. This change permeated into other cryptosystems, including DSA, which now enjoyed side-channel protection against timing attacks and cache-timing attacks. However, this protection was only temporary due to the attack and the countermeasure adopted as a result of [32]. The DSA execution flow was slightly modified by the nonce padding fix—an essential change to avoid leaking the bit length of the nonce in order to prevent timing attacks on DSA. Yet, this countermeasure caused the flag to be lost since the padded nonce was created as a copy of the original nonce using the `BN_copy` function, which failed to propagate the flags set in the `BIGNUM` structure from source to destination. This flaw resulted in signatures being generated using the SCA-vulnerable *sliding window* modular exponentiation function². The `BN_copy` function completely ignores the flags set in the source `BIGNUM` value, and in fact, OpenSSL instead recommends using `BN_with_flags` as this function does propagate previously set flags from source to destination.

From an SCA perspective, Publication I combines the `FLUSH+RELOAD` technique [144] with a performance degradation attack [14], slowing down the execution of the *sliding window* modular exponentiation function enough to accurately trace the sequence of square and multiply operations performed by the algorithm with a spy process running in parallel to the DSA operation. This sequence of operations is then converted into partial bit knowledge of the nonce value, which is finally converted into the private key after applying lattice methods [65]. In terms of novelty, Publication I presents the first cache-timing attack at the protocol level, i.e., the attack recovers private keys used during DSA signature generation as part of the TLS and SSH protocols, both using OpenSSL as the cryptography provider. Prior to this work, cache-timing attacks were performed by either embedding the spy process directly in OpenSSL or targeting directly the cryptographic primitive running in isolation. These approaches reduce considerably the cache noise, translating into fewer errors captured by the spy process, requiring less signal post-processing and overall

²Commit: 44a287747f8594c0ab517526190c7d8af6f8572b

less signatures and traces to recover private keys. Despite the increased cache noise generated by the TLS and SSH protocols during the trace capture by the spy process, Publication I demonstrates that it is feasible to perform cache-timing attacks at the protocol level, and moreover, it shows that the accuracy and complexity of the cache-timing attack depends on the target protocol, i.e., results vary between TLS and OpenSSH even if the target function is identical. As a fix for this specific software defect, the flag is checked in the `BIGNUM` structure holding the padded nonce value prior to modular exponentiation, and is set if it is not present³, forcing DSA to use the constant-time modular exponentiation function. In addition to OpenSSL, BoringSSL and LibreSSL libraries suffered from the same software defect, as this issue was already present by the time these libraries were created as forks from OpenSSL.

This side-channel attack is the first of many subsequent works exploiting software defects and programming errors involving the `BN_FLG_CONSTTIME` flag, however using the experience gained from this first analysis, it was evident that the architectural decision of using the flag as an opt-in mechanism instead of adopting a secure-by-default approach was not optimal to protect against side-channel attacks. In fact, the initial fix for this issue in LibreSSL shortly introduced another side-channel vulnerability during nonce inversion in DSA⁴, which was quickly fixed. This transient defect supported and highlighted the issues with the flagging mechanism, i.e., it is extremely error-prone and easy to misuse. This observation sparked the idea leading to the next research work.

3.2 A Cache-Timing Attack on Constant-Time NIST P-256

After the work by Aciğmez, Gueron and Seifert [5], the old `BN_FLG_EXP_CONSTTIME` was substituted by the current `BN_FLG_CONSTTIME` which was applicable not only to the modular exponentiation function but also other functions such as modular inversion and division, therefore impacting other cryptosystems using these operations. The change was done by replacing all the instances of the old `BN_FLG_EXP_CONSTTIME` flag with the new `BN_FLG_CONSTTIME` flag,

³Commit: 399944622df7bd81af62e67ea967c470534090e2

⁴<https://github.com/libressl-portable/openbsd/pull/61>

in addition to setting the flag in several new places calling the `BN_div` and `BN_mod_inverse` functions across multiple cryptosystems. Unfortunately these changes were not equally applied to ECC-based cryptosystems such as ECDH and ECDSA, consequently leaving a gap with respect to side-channel protection, and enabling a new cache-timing attack. Publication II [106] exploits this gap in OpenSSL in the context of ECDSA instantiated with the widely used NIST P-256 curve using the SCA-secure *fixed window* scalar multiplication function implemented by Käsper [78]. The offending function this time was the input-dependent BEEA function, previously shown to be vulnerable against SBPA [5, 11, 17]. The secure-by-default approach introduced by the changes in [78] clashed with the insecure-by-default approach used by the flagging mechanism, resulting, yet again, in a flag not set in the nonce value. It made sense, since the execution flow followed by default the SCA-secure *fixed window* scalar multiplication function during the computation of the signature's first half, however, protection of the second half was forgotten. In other words, the flag was not set in the nonce prior to modular inversion computation, leading to the SCA-vulnerable `BN_mod_inverse` function. In order to exploit this vulnerability, Publication II improves the performance degradation attack described in [14] by using the performance counters exposed through the `perf` command in Linux. In simple terms, the new approach flushes memory lines from the target code out of the cache in a tight loop, and uses the `perf` command to count the number of cache-misses. This is done for each memory line used by both, the target caller function and the lower level callees, hence providing a quantitative picture of highly utilized memory lines during a normal execution. This resulted in a list of candidate memory lines to degrade that, potentially, provide the most degrading effect. The performance degradation experiments demonstrate that CPU slow down is not enough to successfully degrade a process, and instead, other metrics such as cache-misses and branch mispredictions are better indicators of a successful degradation attack.

Publication II combines the improved performance degradation attack with the well-known FLUSH+RELOAD technique to trace the sequence of operations performed during the modular inversion operation of the nonce value, obtaining a sequence of right-shifts and subtraction operations. Aldaya, Cabrera Sarmiento and Sánchez-Solano [11] used an algebraic algorithm to extract

bits of the nonce based on the sequence of right-shifts and subtraction operations, reducing the side-channel leakage requirements initially given in [5] for the BEEA. The approach followed in Publication II also uses this sequence of operations, but in contrast to [11], it empirically determines the number of bits leaked for every possible valid sequence of a given length, i.e., it performs 2^{26} BEEA computations, extracting tuples of sequence, bit length, and value for each computation. The results show that sequences of length 5 and 7 leak at least 3 bits of information with the lowest error rate, a desired goal as all sequences, and consequently all signatures, can be used in the lattice stage to recover the private key. Moreover, this work demonstrates this attack at the protocol level by attacking OpenSSL NIST P-256 ECDSA featuring constant-time scalar multiplication when used inside the TLS and SSH protocols.

To mitigate this attack, two solutions were adopted: (i) properly setting the flag before nonce inversion during ECDSA computation⁵, and (ii) use of FLT and the `BN_mod_exp_mont_consttime` function to compute the modular inversion operation⁶. Since the targeted OpenSSL version (1.0.1u and previous) reached end of life before this work was published, the mitigation proposed—setting the `BN_FLG_CONSTTIME`—was not accepted by OpenSSL. Hence, it was up to the affected vendors to choose the countermeasure to adopt.

3.3 Single-Trace Cache-Timing Attack on RSA Key Generation

Up to this point, the cache-timing attacks discussed in Publication I and Publication II focus on digital signature algorithms such as DSA and ECDSA due to two main reasons. On the one hand, the offending vulnerable algorithms are well-known to be used in DSA and ECDSA, with a troubled history of attacks and failed countermeasures in OpenSSL, suggesting that more vulnerabilities could potentially exist in the code. On the other hand, cryptosystems like RSA have been heavily scrutinized and analyzed by researchers working on SCA, resulting in few successful side-channel attacks on software. However, previously exploited algorithms such as `BN_mod_exp_mont_consttime` and

⁵<https://securitytracker.com/id/1037575>

⁶Commit: 8aed2a7548362e88e84a7feb795a3a97e8395008

`BN_mod_inverse` are a pillar for public-key cryptography, thus they found use in several algorithms across different cryptosystems throughout OpenSSL. Using a new and simple, but rather effective methodology, Publication III finds more instances of function calls to SCA-vulnerable functions leading to the first cache-timing attack against the RSA key generation procedure. The methodology and the newly developed tool follow these simple steps:

- The tool takes as input a list of known side-channel vulnerable functions, in this case the focus is in OpenSSL, thus includes functions such as `BN_mod_inverse`, `BN_gcd`, and `BN_mod_exp_mont`.
- The tool uses a debugger, such as `gdb`, and sets break points at multiple lines of code based on the previous functions. Note that these functions are SCA-vulnerable, therefore they should not be reached with secret inputs.
- Finally, the tool produces a report if any break point is reached during execution.

The tool reported several hits to SCA-vulnerable break points when it was applied to the RSA key generation procedure, with `BN_gcd` being the most relevant function as it leaks information from the secret values $p - 1$ and $q - 1$. More specifically, during RSA key generation several prime value candidates are generated, and the function `BN_gcd` is used to check the coprimality between each candidate value and the public exponent e , thus effectively computing `BN_gcd(e, p - 1)`. This ensures that the public exponent e is invertible, thus $d \equiv e^{-1} \pmod{(p - 1)(q - 1)}$.

Although being a similar leak as in Publication II, this cache-timing attack requires a quite different approach since private key recovery from small bit leakage is not possible in RSA. Moreover, the fact that key generation is a one-time operation further complicates the attack, as it requires nearly perfect traces. From an attacker's perspective, a favorable factor from the leakage is that the trace collected by the spy process contains a big portion of the `BN_gcd` execution exclusively from the secret values $p - 1$ and $q - 1$, this is due to the large bit length difference—more than 1000 bits—between e and the secret values. During a typical `BN_gcd` execution, $p - 1$ (and similarly $q - 1$) must be reduced to the size of e before the execution is interleaved between e and the

secret values, this means that the algorithm operates and modifies the secret value state for almost all of the algorithm execution. Despite this, the trace still contains several errors, but combining an error correction algorithm [63], and a lattice attack [100], Publication III successfully recovers the secret prime values generated during RSA key generation.

As a side note, simultaneous to Publication III, Weiser, Spreitzer and Bodner [137] independently found the same vulnerability, and demonstrated a controlled-channel page-fault attack on RSA key generation in a privileged threat model using Intel SGX enclaves. The authors disclosed their findings to the OpenSSL team and provided a patch, replacing the offending `BN_gcd` function with the `BN_mod_inverse_no_branch`⁷. However, this patch was not enough to completely close the leak and a subsequent patch was merged into the library⁸.

3.4 A Game of Whack-A-Mole

The previous methodology and the accompanying tool developed in Publication III, as well as 15 years of cache-timing attack history, unveiled a widespread problem affecting OpenSSL and other cryptographic libraries with respect to SCA. The problem is that a considerable percentage of side-channel attacks are attributable to a small known set of vulnerable functions used in a wide variety of cryptosystems and contexts. Focusing on this small set of vulnerable functions it is possible to greatly reduce the side-channel attack surface by tracking their usage in several places and entry points in code. Publication IV takes and greatly improves the methodology from Publication III, and packages it in a new tool called Triggerflow⁹. Triggerflow permits to track specific code paths during program execution through the use of code annotations, i.e., it generates a report and a complete map, through a call graph, of all the functions called leading to the Triggerflow annotation for any program running under it. The main goal of Triggerflow is to support developers by providing an accurate map of execution flows, as well as testing reachability of new code in order to detect vulnerable code paths enabling side-channel attacks. Moreover, the

⁷Commit: 8db7946ee879ce483f4c81141926e1357aa6b941

⁸Commit: 54f007af94b8924a46786b34665223c127c19081

⁹<https://gitlab.com/nisec/triggerflow>

tool is flexible, supports different types of annotations such as conditionals, groups, and ignore annotations; and it can be used in the Continuous Integration (CI) development pipeline to automatically generate reports, maps, and test new builds, allowing to detect code flaws and regressions. Publication IV demonstrated the effectiveness of Triggerflow in two ways: firstly by validating old and previously exploited SCA vulnerabilities in OpenSSL, such as the ones exploited in Publications I–III and VII; and secondly by finding a new side-channel leak and a software defect.

The side-channel leak was due to the use of the insecure modular inversion function `BN_mod_inverse` when converting an EC point from projective to affine coordinate representation. Leakage protection in the context of coordinate conversion was a known issue [90, 99] that could potentially be exploited with a side-channel attack. In fact, this same issue was later found and exploited in the `mbedtls` library by Aldaya, Pereida García and B. B. Brumley [13]. With the help of Triggerflow this leak was closed by unifying the finite field inversion across the EC module in OpenSSL¹⁰. In other words, instead of setting the `BN_FLG_CONSTTIME` flag in each function requiring modular inversions, a new function pointer was added to the EC structure containing pointers for the required finite field operations. This new pointer led to one of three different SCA-secure implementations based on the elliptic curve in use.

Triggerflow also helped to identify a deep rooted issue in the `BN_CTX` object used by OpenSSL. The `BN_CTX` object is a stack-like buffer storing `BIGNUM` variables that are shared among the top level functions and their callees and descendants, with the goal of minimizing memory allocations [31]. During a typical execution of a cryptosystem in OpenSSL, such as ECDSA, the lifetime of `BIGNUM` variables inside the `BN_CTX` object can vary greatly, and some of these variables are reused at different stages during the execution of the cryptosystem. Interestingly, Triggerflow detected that the `BN_FLG_CONSTTIME` flag is preserved in those `BIGNUM` variables in which the flag was previously set, resulting in different execution paths depending on the flag status of each `BIGNUM` variable used. Although it seemed that this software defect only led to the execution of SCA-secure code paths, it also introduced false negatives, unexpected function calls, performance penalties, and potential application crashes. The fix was

¹⁰Commit: 97dfa14e05f142d7c00a46514975a0a02cf14668

simply to unconditionally clear the `BN_FLG_CONSTTIME` flag every time a `BIGNUM` variable is retrieved from the `BN_CTX` object¹¹.

In the case of OpenSSL, the library supports a wide variety of platforms and building options, dictating the implementation to execute at runtime, therefore catering to a large user base. However, this also means it is difficult to test every possible combination for possible SCA-vulnerable code paths and software defects. Triggerflow is a remarkably useful tool for developers dealing with such projects as it can be automated to reliably test for new and existing code leading to regressions. Moreover, Triggerflow confirmed once again that the OpenSSL architecture decision of making the `BN_FLG_CONSTTIME` flag an opt-in instead of a default mechanism resulted in unexpected and fragile execution flow with deeper implications than those originally intended.

3.5 Towards an SCA-secure OpenSSL

Prior cache-timing attacks on OpenSSL, especially those described earlier in this dissertation, exploited software flaws accumulated over more than a decade, mostly involving the insecure-by-default approach requiring correct and exact usage of the `BN_FLG_CONSTTIME` flag. One would think that some lessons would be learned from this abundant list of examples, avoiding repeating these mistakes in current and future code in the process of integration into the library. Unfortunately, this is not the case when the complexities of the code base demand a specialised skill set and a deep knowledge of the library possessed by only few developers, and more importantly, out of consideration for average developers. The burden of keeping track of subtle details such as the `BN_FLG_CONSTTIME` flag, and correctly implementing these details to achieve SCA-secure implementations should be abstracted away from the developers. In addition to being a specialised task, SCA is an iterative process requiring years of expertise, and often is extremely error-prone. All of the previous points are demonstrated in OpenSSL, suggesting that lessons take a long time to be learned in big FOSS projects, leading to repeated software flaws and vulnerabilities, especially when code contributions are integrated in a rush into the code base.

Publication V analyzes the late-stage adoption of the SM2 cryptosystem into

¹¹Commit: 4803ad495293515ccdd2b490620384d9736c4956

OpenSSL just before the release of version 1.1.1. Despite SM2 being similar and sharing many low-level functions with other widely used EC cryptosystems, it failed in the exact same functions previously exploited using SCA. Additionally, this hurried integration contained traditional software bugs causing crashes and other security vulnerabilities.

The implementations for SM2 digital signature (SM2DSA) and public key encryption (SM2PKE) rely on the same building blocks as ECDSA and ECIES, namely, modular inversion and scalar multiplication functions. These functions require the `BN_FLG_CONSTTIME` flag to be set prior to their execution when dealing with secret inputs, otherwise the execution flow leads to the SCA-vulnerable versions of those algorithms. The rushed integration of the SM2 cryptosystem at a late stage prior to the major release of OpenSSL version 1.1.1 left a short window of time for public reviews, and Publication V filled this gap with a complete analysis and rework of OpenSSL internals not only improving the SCA security of SM2, but the overall security of the library. The analysis revealed software flaws enabling remote timing attacks, cache-timing attacks, and EM attacks.

The remote timing attack vulnerability discovered was facilitated by a lack of scalar padding before the scalar multiplication function, hence it was analogue to the research in [32, 33]. SM2DSA can be instantiated with a variety of elliptic curves, including prime curves and binary curves, so this attack was demonstrated in both types of curves. Binary curves use the Montgomery ladder algorithm for scalar multiplication whereas the recommended SM2 prime curve used the vulnerable *wNAF* scalar multiplication function. Despite the algorithm difference, the execution time of both algorithms was shown to be correlated to the length of the nonce used during signature generation, therefore leaking the top bits of the nonce on each signature. Additionally, two different cache-timing attacks were demonstrated against SM2DSA. On the one hand, and as mentioned before, the scalar multiplication algorithm used with the SM2 recommended curve was the SCA-vulnerable *wNAF* function, therefore the spy process clearly captured the sequence of ECC double and add operations, leaking information from the secret scalar. On the other hand, the second attack was the result of, yet again, a missed `BN_FLG_CONSTTIME` flag prior to modular inversion, hence taking the SCA-vulnerable code path to the `BN_mod_inverse`

function, similar to the vulnerabilities in Publications II and III. However, in contrast to (EC)DSA, SM2DSA does not invert the nonce value, but instead the long-term private key, therefore leaking bits directly from the private-key during each signature generated, and giving an attacker the opportunity to accurately trace the sequence of operations from the BEEA execution. Finally, the EM analysis confirmed that the *wNAF* function affecting SM2DSA, also affected SM2PKE but this time during the decryption part, thus leaking the sequence of ECC double and add operations from the long-term private key.

While the techniques and the attacks themselves were not novel, Publication V showed that not only old, but also new asymmetric cryptosystems are potentially vulnerable to SCA due to continuous support of the opt-in and insecure-by-default approach. Moreover, this work proved that OpenSSL developers must be careful when implementing new cryptographic primitives, as it is quite easy to shoot oneself in the foot due to the current mechanisms in use, demanding an unreasonable skill set in order to achieve an SCA-secure implementation.

On the mitigation side, Publication V contributes greatly towards a secure-by-default OpenSSL with several contributions to the EC module, thus not only improving SM2, but also other ECC cryptosystems. The contributions are the following:

- *Scalar multiplication.* The insecure *wNAF* function is no longer the default algorithm for scalar multiplication. Instead, an early exit is taken to the new Montgomery ladder [96] when either a fixed-point or a variable-point scalar multiplication must be calculated¹², the former commonly used in ECDSA, and the latter in ECDH and ECIES. The *wNAF* function is still in the code base, however is only used for multi-scalar point multiplication, a common use case for digital signature verification where no secrets are used as input values.
- *Scalar padding.* The padding is no longer dependant on the cryptosystem, but instead was pushed to the lower-level EC module, ensuring a constant number of ladder iterations¹³.
- *Coordinate blinding.* In addition to the highly regular Montgomery lad-

¹²Commit: 3712436071c04ed831594cf47073788417d1506b

¹³Commit: fe2d3975880e6a89702f18ec58881307bf862542

der, coordinate blinding [41] serves as a countermeasure against DPA attacks as well as attacks during coordinate conversion [13, 90, 99] by applying a map by a random value before the ladder execution¹⁴.

- *Modular inversion.* Similarly, the insecure BEEA function was phased out of EC cryptosystems, replacing it by the FLT method originally provided by Gueron and Krasnov [57]. This change finally provided a secure-by-default modular inversion at the EC layer¹⁵.

Arguably, the biggest contribution of Publication V is the change to the execution flow to avoid dependence on the `BN_FLG_CONSTTIME` flag for side-channel protection. While these changes did not completely eliminated the use of the flag in OpenSSL, it was a big step towards a secure-by-default execution flow in cryptosystems based on EC.

3.6 Port Contention Side-Channel Attack

So far, this dissertation dealt in its majority with SCA abusing the cache hierarchy using different cache-timing attack techniques, threatening a wide variety of cryptosystems as demonstrated in Publications I–V and VII. Yet, the techniques used as part of those publications, as well as the techniques previously described in Section 2.2, are only a subset of a growing number of techniques exploiting not only the cache hierarchy, but also other components in the microarchitecture. CPUs supporting SMT share many other components, in addition to the caches, among their executing threads. These components have the potential to be abused to break the isolation between independent processes, thus allowing them to communicate. Publication VI describes a new side-channel attack vector exploiting port contention on the execution units within a single physical CPU core, enabled by the SMT technology. Dubbed PORTSMASH, this new side-channel attack technique differs from cache-based attack techniques by targeting a non-persistent shared resource, making it stealthier than the cache-based counterpart.

Figure 3.1 shows PORTSMASH in action. At a high level, given a CPU with SMT support, two programs executing in two separate logical cores but within

¹⁴Commit: f667820c16a44245a4a898a568936c47a9b0ee6e

¹⁵Commit: 792546eb18c3088d7eca0c1eb86695bcae18d8

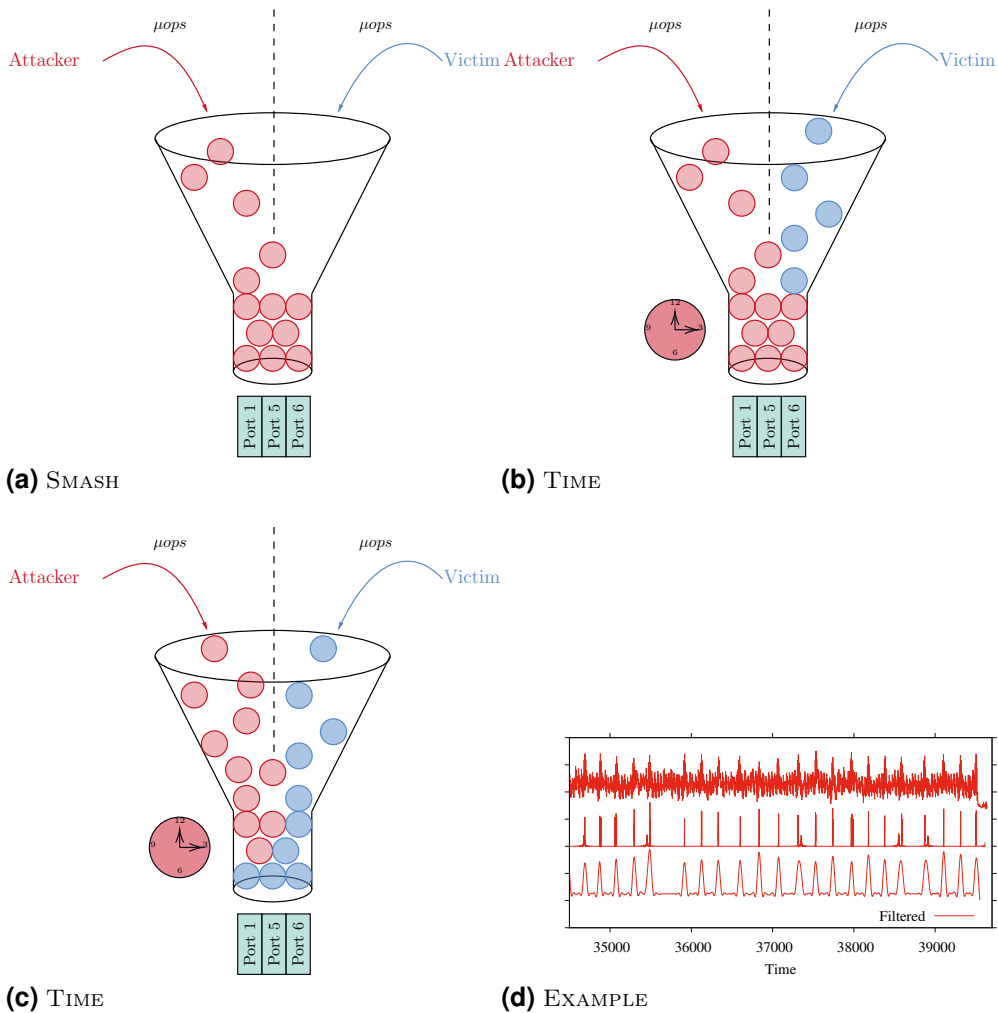


Figure 3.1 The PORTSMASH technique: (a) attacker issues instructions saturating a set of ports; (b) attacker times the instruction execution completion; (c) attacker continues timing and observes variations due to victim; (d) example trace of attacker spying on the *wNAF* scalar multiplication revealing add operations (peaks).

the same physical core, can potentially infer information from each other by looking at the port utilization. A spy process can saturate one or several ports by cleverly issuing instructions with a specific port footprint, then it measures the time it takes to execute said instructions. This is done in a loop, trying to detect port contention, which is observed through an increased latency. From an SCA perspective, if the victim's port utilization depends on a secret, this

can be potentially observed by the spy process using PORTSMASH.

As can be seen, PORTSMASH operates at a port level, and as such, it provides a very fine spatial granularity—finer than previous cache-timing attack techniques. Moreover, it is able to adapt to different scenarios requiring different port saturation according to the target footprint, and due to this flexibility, the attack requirements are minimal since no knowledge of the cache-hierarchy, cache lines, nor eviction sets are needed prior to the attack, making it considerably more portable than the cache-based techniques.

Publication VI first demonstrates PORTSMASH in action by using it to create a covert channel allowing communication between two unrelated programs, then it proceeds to demonstrate its efficacy by targeting and exploiting the vulnerable *wNAF* scalar multiplication function in the context of the NIST P-384 elliptic curve. Using PORTSMASH, a spy process creates contention in a specific set of ports while timing the latency of executing a group of instructions utilizing the set of ports previously chosen. This generates a raw signal trace containing the sequence of ECC double and add operations executed during scalar multiplication. After processing the signal it reveals the LSDs of the secret scalar value, then the LSDs of several traces are combined and lattice methods applied, resulting in full key recovery. Similar to Publications I–III, the attack is not only demonstrated against ECDSA in isolation, but also when running as part of the TLS protocol, showing that this technique is as capable as cache-based techniques. Additionally, this attack is also presented against an OpenSSL victim process running inside an Intel SGX enclave, suggesting that this technique can be transparently applied to this technology. As a side note, an observant reader may notice that previous Publication V claims to have fixed the SCA-vulnerable *wNAF* function. Indeed, the function is no longer in use in new OpenSSL releases, however, older releases still use the vulnerable function. Nevertheless, the attack on this curve on an old OpenSSL version is for demonstration purposes, to show the impact of this new side-channel vector.

PORTSMASH is possible due to the very nature of SMT, i.e., trying to maximize resource utilization by sharing as many resources among processes as possible, even if these processes have different security needs. As possible countermeasures against this new vector, Publication VI recommends two different approaches: (i) use code whose port footprint does not depend on secret values;

and (ii) remove the SMT attack surface. The simplest of the countermeasures is to disable SMT at the OS level, which was promoted and is already adopted by OpenBSD developers¹⁶. The port-independent code approach is in line with the countermeasures previously described and implemented as part of this dissertation, and for this specific case, the newest versions of OpenSSL no longer uses the *wNAF* scalar multiplication by default, but instead it uses the secret-independent Montgomery ladder algorithm [96].

3.7 Side-Channel Attacks Enabled by Cryptographic Key Formats

Previous SCA-vulnerabilities and software flaws affecting common functions used by different cryptosystems, as described in Publications I–VI, were exploited at the protocol level. This allowed to demonstrate the impact of the vulnerabilities, and to show that cryptography does not exist in isolation, instead, it is commonly used as a building block in a protocol or a system. The countermeasures applied to fix these flaws were introduced as close as possible to the lower level functions, with the expectation that current and future implementations would directly benefit from the SCA countermeasures in place. However, Publication VII exposes, one more time, that the insecure-by-default `BN_FLG_CONSTTIME` flag mechanism prevents new and existing implementations benefiting from these SCA countermeasures even when the newly discovered vulnerabilities are identical to others previously exploited and fixed, but in a slightly different context.

OpenSSL and many other cryptographic libraries offer command line tools allowing users to quickly perform a wide variety of cryptographic operations that are commonly needed when dealing with public and private keys. These command line tools are convenient as they abstract away complex operations such as reading files, parsing keys, sending the output to either a file or to the screen, from the actual cryptographic operations. Moreover, for the sake of simplicity, these command line tools try to be as generic as possible within reason, i.e., the user can input a cryptographic key and is up to the tool to

¹⁶<https://marc.info/?l=openbsd-cvs&m=152943660103446>

decide if the key is correct and can be used to perform the chosen operation, i.e., sign, verify, encrypt, or decrypt a message.

Publication VII analyzes a set of command line tools in OpenSSL and mbedTLS operating in private keys. The analysis uses Triggerflow from Publication IV, and finds new and repeated vulnerabilities facilitated by the insecure-by-default behavior adopted across OpenSSL, leading to SCA-vulnerable code paths that are either oblivious to the flagging mechanism, or fail to set the `BN_FLG_CONSTTIME` flag correctly. Focusing on OpenSSL, the vulnerabilities are dependent on two different features: key format, and key parameters.

Typically, public and private keys contain several key parameters required by the cryptosystem in order to perform a specific operation. These parameters are defined by standards such as the ANSI X9.62 standard [2] and the SEC1 standard [118], and often these standards declare the mandatory and the optional parameters. However, sometimes the standards only issue recommendations, leaving to the developer’s interpretation on how to deal with the parameters—such as the case of RSA and PKCS #1 (RFC 8017 [97]). From a software perspective, this flexibility increases the complexity of key parsing tremendously. Cryptographic library developers need to decide which mandatory and which optional parameters to support, while keeping in consideration interoperability across libraries, across versions, and across formats—with legacy formats sometimes supporting and containing a subset of parameters.

Publication VII discovers that in the case of OpenSSL, the SCA-security of ECDSA keys depends on some key parameters, which at the same time are dictated by the way the keys are provisioned. More specifically, keys containing the curve name or the curve OID, follow a specialized scalar multiplication implementation for the given curve if it is available, otherwise they default to the generic Montgomery ladder implementation. Keys containing explicit parameters follow the generic implementation that is SCA-secure thanks to the changes introduced in Publication V, but it is not as optimized as a specialized implementation. This is true even if the explicit parameters describe a named curve, that is, they are equivalent. Finally, keys containing explicit parameters but with a zero value in their co-factor parameter, default to the SCA-vulnerable *wNAF* implementation previously exploited in Publication VI. As countermeasures to fix these issues, Publication VII introduced changes to

match explicit parameters to named curves parameters¹⁷, checking their equivalence, and allowing them to enjoy optimized and specialized implementations when possible. In the case of the co-factor, instead of relying on the parameter value in the key, it is now computed at runtime¹⁸ for both named curves and curves with explicit parameters, thus avoiding the insecure *wNAF* scalar multiplication function.

Similarly, Publication VII detects that OpenSSL uses SCA-vulnerable functions when using, converting, or checking DSA and RSA private keys. In the case of DSA, OpenSSL supports private keys from Microsoft’s proprietary PVK and MSBLOB formats, but leaks private key material when using them or converting them to other widely used formats such as PEM (RFC 7468 [77]). The library expects the private and public keys to be readily available, but since these formats only contain private key material, the public part must be computed at runtime. Analogous to Publication I, the execution flow uses the SCA-vulnerable *sliding window* function to compute the public key because the private key is not marked with the `BN_FLG_CONSTTIME` flag, thus directly leaking private key information. In the case of RSA, OpenSSL provides a command line tool to check the validity of an RSA key, which involves recomputing the key parameters and comparing them to the input key. Unfortunately, the computation of many secret key parameters is done in an insecure manner. Resembling the flaws presented in Publication III, OpenSSL fails to set the `BN_FLG_CONSTTIME` flag in several `BIGNUM` values holding secret key parameters, leaking information on several functions:

- `BN_mod_exp_mont` and `BN_mod_inverse` leak information on p and q during primality testing.
- `BN_gcd` leaks information on $p - 1$ and $q - 1$ during private exponent d computation.
- `BN_mod_inverse` leaks information on q during CRT parameter i_q computation.

Despite the countermeasures introduced in Publications I–VI, this work finds new instances of missed flags, supporting previous claims on the hardness of

¹⁷Commit: bacaa618c26411d212015493d0eb82076a3e76a1

¹⁸Commit: b783beeadf6b80bc431e6f3230b5d5585c87ef87

using properly the flag. While RSA key checking and DSA key conversion from these legacy formats are perhaps not widely used operations, their security and resistance against SCA is equally important. Moreover, all of these are preventable flaws, in other words, they could have been avoided by using a secure-by-default approach. To fix these issues, the `BN_FLG_CONSTTIME` flag was properly set during DSA¹⁹ key conversion and RSA²⁰ key checking. Moreover, the highly input-dependent `BN_gcd` function affecting this and previous Publication III, was replaced by a new constant-time version proposed by Bernstein and Yang [26]. This new version provides SCA-security across OpenSSL to all the cryptosystems using this function, therefore getting OpenSSL a step closer to constant-time.

3.8 Summary of Mitigations on OpenSSL

This dissertation tries to present the research and results as a continuous, clear, concise, and logical sequence of events to facilitate its understanding. Looking back, it is easy to put all the pieces together, however, it was not always the case while performing the research work, especially when dealing with an ever evolving project such as OpenSSL, offering several versions—sometimes supporting very different cryptographic implementations.

At the time of writing OpenSSL version 3.0 is under beta release, bringing new features and several architecture changes compared to the current long term support (LTS) version 1.1.1. From a cryptography point of view, both of the current versions contain either the mitigations introduced as a direct result of Publications I–VII, or improved mitigations due to more recent research work. Figure 3.2 presents an updated timeline including the results of this chapter. In summary, the mitigations presented in this dissertation and adopted in OpenSSL can be classified by the affected cryptosystem as follows:

ECC. The SCA mitigations provided are the following:

- Curves without a specialized implementation such as NIST P-384, *Brain-pool* curves [95], SECG curves [119], and generic prime curves defined

¹⁹Commit: 724339ff44235149c4e8ddae614e1dda6863e23e

²⁰Commit: 311e903d8468e2a380d371609a10eda71de16c0e

using custom parameters—e.g., for *GOST* [44] or SM2²¹—now perform scalar multiplication using the Montgomery ladder instead of the SCA-vulnerable and input-dependent *wNAF* implementation.

- Scalar padding has been integrated into the Montgomery ladder function, therefore providing a constant number of ladder iterations.
- The modular inversion uses the FLT method instead of the `BN_mod_inverse_no_branch` function.
- Generic prime curves use coordinate blinding.
- The co-factor is manually computed when parsing a key, leading to the correct scalar multiplication function.
- Explicit key parameters are matched against a known list of named curves, enabling the use of specialized implementations even if a key was provisioned with explicit parameters.

DSA. The SCA mitigations are the following:

- The modular exponentiation uses the *fixed window* implementation after properly setting the `BN_FLG_CONSTTIME` flag during signature generation.
- The modular exponentiation uses the *fixed window* implementation after properly setting the `BN_FLG_CONSTTIME` flag during key conversion and general key use of PVK and MSBLOB keys.
- The modular inverse used the `BN_mod_inverse_no_branch` function after setting the missing flag, however subsequent contributions changed the execution flow, allowing DSA to enjoy the secure-by-default approach provided by the FLT algorithm.

RSA. The SCA mitigations are the following:

- Modular exponentiation uses the *fixed window* implementation after setting the missing `BN_FLG_CONSTTIME` flag during key generation and key checking.

²¹<https://datatracker.ietf.org/doc/html/draft-shen-sm2-ecdsa-02>

- Modular inverse uses the `BN_mod_inverse_no_branch` function after setting the missing `BN_FLG_CONSTTIME` flag during key generation and key checking.
- The `BN_gcd` function was replaced by the `BN_mod_inverse_no_branch` and appropriate flags during key generation²².
- The highly input-dependent `BN_gcd` function was replaced by a new constant-time and SCA-secure `BN_gcd` function. This is now the secure-by-default GCD implementation across the library, thus fixing the vulnerabilities during key generation and key checking.

²²This was independently and simultaneously discovered and fixed by [137]

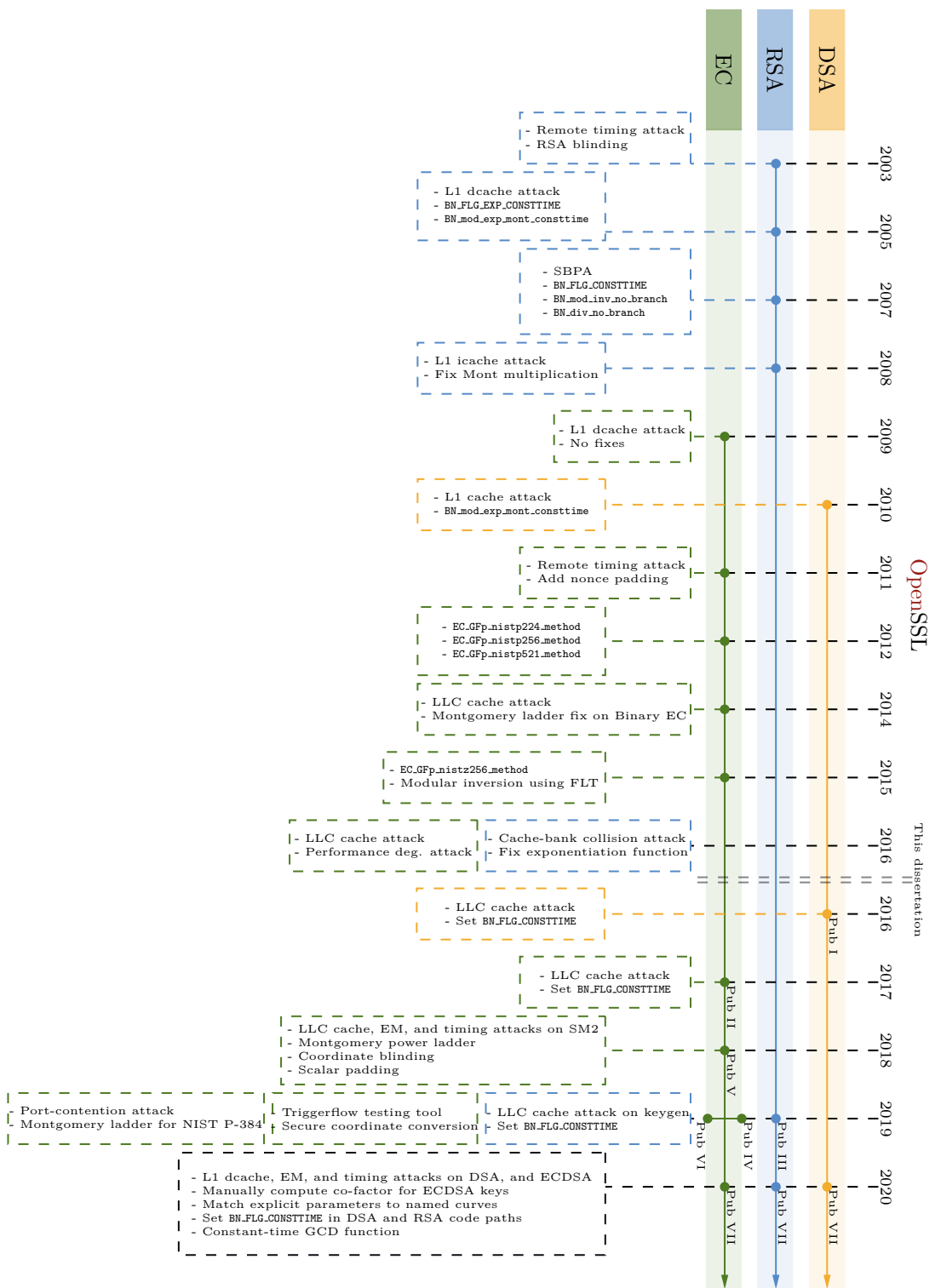


Figure 3.2 Updated timeline including attacks and countermeasures from Publications I–VII.

4 CONCLUSIONS

This dissertation takes a closer look and criticizes the well-intentioned, but flawed, defense mechanism against side-channel attacks in OpenSSL—the `BN_FLG_CONSTTIME` flag. The architecture decision to make this an opt-in mechanism instead of a secure-by-default mechanism in the name of performance, has led to several security flaws representing a serious threat to security-critical software relying on this library. This dissertation advances the field of side-channel analysis, cache-timing attacks and applied cryptography not only using state-of-the-art attack techniques, but improving them, proposing new attack techniques, and adopting better defense mechanisms in real-world cryptography.

The following paragraphs briefly discuss the future direction of the results presented in this dissertation, as well as topics related to OpenSSL, microarchitectural attacks, cache-timing attack defenses, and final words of advice. After this, the introductory part of this dissertation ends, and is followed by the seven publications previously discussed.

The future of the `BN_FLG_CONSTTIME` flag. Publications I–VII built a good knowledge base not only of cache-timing attacks techniques, but also of the most common errors and pitfalls enabling side-channel attacks on multiple public-key cryptosystems, many derived from the `BN_FLG_CONSTTIME` flag misuse. The work leading to this dissertation improved the overall security of OpenSSL as described in Section 3.8, therefore providing a better understanding of the cryptographic library as a research tool, and consequently easing the path for future researchers exploring this field.

At the time of writing this dissertation OpenSSL 3.0 is in beta release, and unfortunately, the `BN_FLG_CONSTTIME` flag is still a mechanism used in the library to provide SCA-security. However, as seen throughout this dissertation, it is a fragile and error-prone mechanism. The recent work by Braga, Fouque

and Sabt [28] follows a similar trend as Publication VII by performing a side-channel attack exploiting a well-known vulnerability but through a different protocol, enabled once more by a missing `BN_FLG_CONSTTIME` flag. The offending function is the SCA-vulnerable `BN_mod_exp_mont` modular exponentiation function used as part of the Secure Remote Password (SRP) protocol. While correcting vulnerabilities related to the `BN_FLG_CONSTTIME` is trivial, these results demonstrate that the `BN_FLG_CONSTTIME` flag is the gift that keeps giving, and it will continue to do so until OpenSSL finally decides to adopt a secure-by-default approach across the library—which could take many years based on current support and release schedules.

As a side note, it is worth mentioning that OpenSSL is not the only library using and affected by the insecure-by-default `BN_FLG_CONSTTIME` flag. Born in 2014 as forks of OpenSSL after the high-profile *HeartBleed* vulnerability, BoringSSL and LibreSSL originally inherited the `BN_FLG_CONSTTIME` flag from OpenSSL, thus influencing their code base. This is reflected in Publication I and Publication II, where both vulnerabilities affected OpenSSL, BoringSSL, and LibreSSL. At the time of writing, OpenSSL and LibreSSL are still using the `BN_FLG_CONSTTIME` flag as a security mechanism against SCA, while BoringSSL has adopted a secure-by-default approach.

OpenSSL Security Policy. Between April and May 2019 the OpenSSL Security Policy was updated with a new *threat model* section, and in this section, OpenSSL declared certain attacks outside of the scope of the library. Most relevant for this dissertation, all side-channel attacks requiring physical access or co-location, i.e., SPA, DPA, EM, and most microarchitectural side-channel attacks, were now considered out of the scope of the library. The new security policy meant that no new CVEs would be issued for these attacks, however, the OpenSSL team still tries to address security issues outside of the threat model. The change of policy is reflected in Publication VII where despite reporting and fixing multiple vulnerabilities, a CVE¹ was assigned only to the ECDSA remote timing attack.

Arguably, this policy change was motivated by the growing number of microarchitectural side-channel attacks affecting OpenSSL on the years and months prior to the change.

¹<https://www.openssl.org/news/secadv/20190910.txt>

Applicability of results. In addition to improving the security of OpenSSL, the tools, and the body of knowledge generated for this dissertation—including analysis, exploitation, remediation, and prevention—has started to permeate into other research work, and other cryptographic libraries. Hassan et al. [62] applied a combination of Triggerflow and DATA [138] frameworks to Mozilla’s NSS cryptographic library, directing them to the most common points of failure with respect to SCA in public-key cryptography, finding several issues affecting NSS. The library-wide SCA performed by the authors revealed several flaws in different cryptosystems, including RSA, DSA, and ECDSA, some of which had been present in the library for more than a decade. Similarly, Aldaya and B. B. Brumley [9] expand on the performance degradation technique of Publication II to further decrease the execution performance of the target process, opening the possibility to new attacks. The authors apply their *HyperDegrade* technique to a current version of OpenSSL to perform a successful practical Raccoon attack [94] against the DH key exchange protocol. While new side-channel attacks are getting harder to exploit, thus requiring new or improved techniques, these results suggests that knowledge about existing side-channel vulnerabilities, and protection mechanisms against them are not equally spread and applied to other widely used cryptographic libraries, and therefore further work must be done to ensure SCA protection to libraries that have not been as scrutinized as OpenSSL.

Microarchitectural attacks and beyond. Microarchitectural side-channel attacks targeting microarchitecture components other than the cache memory hierarchy have existed for over a decade, most notably the initial works on branch predictors [5, 6, 7], and arithmetic logic units [8]. As researchers improve the understanding of microarchitecture components, their interaction, and their performance features, new attacks targeting a wide variety of components have been published in recent years. Some of the microarchitecture components that have been exploited more recently include the Translation Lookaside Buffer [34, 48], the Branch Target Buffer [46], the Floating Point Unit [16], the DRAM [79, 83], and the Execution Unit ports as demonstrated in Publication VI.

Moreover, researchers discovered that microarchitecture components not only leak timing information through resource contention when using SCA-vulnerable algorithms, but also these components momentarily hold confiden-

tial information through their microarchitecture state, potentially exposing the information to a clever attacker before changes to the microarchitecture state are reverted—an operation that is hard to achieve in a complete and clean manner. Dubbed transient-execution attacks [35], this new type of attack falls under the umbrella of microarchitectural attacks, and while it is not a type of side-channel attack, it commonly uses microarchitectural side-channel attacks as a building block. Side-channel attacks permit to exfiltrate not only cryptographic information out of the microarchitecture state, but also text and generic pieces of data. Transient-execution attacks rely on operations that should not be performed but are computed, e.g., due to mispredictions or out-of-order execution, causing changes in the microarchitecture state during a small period of time called a *transient window*. This *transient window* is exploited by an attacker to leak information. Meltdown [87] and Spectre [81] are, arguably, the most widely known attacks in this category due both to their impact, and their wide coverage by the media. More importantly, these results have motivated researchers to analyze CPUs and their microarchitecture more comprehensively, looking at them not as isolated components, but as a complete system. This approach allows to increase their security from the design stage, thus reducing their ability to leak information from the lowest level.

Defenses. Several defense mechanisms can be used to prevent cache-timing attacks and side-channel attacks in general. These mechanisms can be applied at different levels such as hardware level, OS level, and software level.

On the hardware level the most prolific defense research is the design of microarchitecture components preventing or limiting resource sharing between processes with different security levels [40, 42, 80, 82, 88, 104, 128, 136, 140]. Other hardware defense mechanisms include developing microarchitecture designs where supported instructions execute independently from the input values, such as in ARM’s Data-Independent-Timing (DIT) [20], Intel’s CMOV instruction family [58], and more recently the RISC-V’s Zkt extension². Also, developing instruction set extensions to give more control for developers to protect against microarchitectural attacks [19].

At the OS level, the most common defense mechanisms proposed involve partition of the cache according to different security levels, trust, and needs [140].

²<https://github.com/rvkrypto/riscv-zkt-list>

Additionally, modifying³ ⁴ ⁵, blocking [131], or introducing noise [123] to the timing measurements have been proposed as defense mechanisms. A different approach is to accept that several microarchitecture components leak information at different degrees throughout all the system, therefore side-channel attacks cannot be easily prevented but they can be detected using for example performance counters [39, 54, 98].

At the software level the most robust defense mechanism is to adopt a constant-time and secure-by-default approach for all the algorithms dealing with secrets and confidential data [24]. Publications I–VII tried to follow this approach as much as possible during the development of the mitigations described in Section 3.8, however the mitigations were selected and sometimes limited according to OpenSSL own standards and coding preferences. This approach is considerably challenging as no generic solution exists that can be applied to all the cryptosystems, however, it is effective, and more importantly it can be retrofitted to provide SCA security to existing systems. Looking at newer cryptosystems with far fewer SCA vulnerabilities during their lifetime such as Ed25519 [25] and Ed448 [60], this is perhaps the best long-term solution to close most of the side channels, instead of applying band-aids such as the `BN_FLG_CONSTTIME` flag.

Finally, extending the approach of constant-time implementations, an idea that has recently gained traction is the adoption of formally verified cryptographic implementations [45, 151] over high performance hand-written implementations. The down-side of hand-written implementations is that often they are too complex to verify for absence of bugs and flaws. This means that formally verified implementations provide both mathematical proofs of correctness, and also proofs of absence of classes of bugs and flaws. Moreover, many formally verified implementations include side-channel attacks as part of their threat model, thus considering SCA-security from design. Multiple tools have been developed on top of these formally verified implementations [22, 109] to help developers automatically generate SCA-secure implementations that can be easily adapted to new cryptographic libraries and software in a wide variety of CPUs without the need of a cryptanalyst.

³https://bugs.webkit.org/show_bug.cgi?id=146531

⁴<https://hg.mozilla.org/integration/mozilla-inbound/rev/48ae8b5e62ab>

⁵<https://bugs.chromium.org/p/chromium/issues/detail?id=158234>

Impact. The research work done for this dissertation has resulted in significant impact on both inside and outside of academia. The publications strive to be open, self-contained, and reproducible, therefore they are often accompanied with research artifacts such as tools and datasets, allowing researchers to reproduce parts of the experiments. With respect to tooling, Triggerflow keeps continuously testing code paths in new code introduced to OpenSSL⁶, testing for possible regressions enabling new side-channel attacks. On the dataset part, perhaps the most notable is [129], resulting from Publication V, which according to Weissbart, Picek and Batina [139] is the first public SCA dataset on ECC. Outside of academia, five out of the seven publications received a CVE number to track the side-channel vulnerabilities discovered. Of those four CVEs, Publication VI gained attention from the media locally and internationally, rising awareness about the importance of information security and system security starting from the hardware design, which is controlled by CPU vendors.

Final words. Without taking merit away from this dissertation and the research involved, undeniably, an important part of the bulk of research for this dissertation is the product of the combination between an insecure-by-default architecture decision—taken more than a decade ago—and incomplete and untested countermeasures accumulated over the years. It is impossible to know where the research field would be if OpenSSL would have decided to take a secure-by-default approach from the very beginning—thus closing most of the side-channel vulnerabilities. However, it is clear that those decisions and mistakes enabled new and ground-breaking research results, moving forward the field of Side-Channel Analysis. Moreover, new threat models and attack scenarios—such as those used during microarchitectural attacks—have pushed for new methods and attack techniques. At the same time, these new methods and attack techniques have allowed attackers to circumvent restrictions, easing the attack requirements, resulting in a positive feedback loop that has rapidly advanced the research field during the last five years—when the research work for this dissertation started.

All in all, this dissertation serves as an example of the attack-defense race between cryptographers and cryptanalysts, by demonstrating practical attacks,

⁶<https://gitlab.com/nisec/openssl-triggerflow-ci>

and providing suitable countermeasures to OpenSSL, while striving for a secure-by-default approach. A final reminder regarding side-channel attacks on OpenSSL is that as long as the `BN_FLG_CONSTTIME` flag lives in the library—which continues to exist in the upcoming OpenSSL 3.0—there is potential to find and (re)introduce new side-channel vulnerabilities.

REFERENCES

- [1] A. Abel and J. Reineke. Reverse engineering of cache replacement policies in Intel microprocessors and their evaluation. *2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2014, Monterey, CA, USA, March 23-25, 2014*. IEEE Computer Society, 2014, 141–142. DOI: 10.1109/ISPASS.2014.6844475. URL: <https://doi.org/10.1109/ISPASS.2014.6844475>.
- [2] Accredited Standards Committee X9, ed. *ANSI X9.62-2005: Public Key Cryptography For The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)*. ANSI American National Standards Institute, 2005.
- [3] O. Aciğmez and W. Schindler. A Vulnerability in RSA Implementations Due to Instruction Cache Analysis and Its Demonstration on OpenSSL. *Topics in Cryptology - CT-RSA 2008, The Cryptographers' Track at the RSA Conference 2008, San Francisco, CA, USA, April 8-11, 2008. Proceedings*. Ed. by T. Malkin. Vol. 4964. Lecture Notes in Computer Science. Springer, 2008, 256–273. DOI: 10.1007/978-3-540-79263-5_16. URL: https://doi.org/10.1007/978-3-540-79263-5_16.
- [4] O. Aciğmez, B. B. Brumley and P. Grabher. New Results on Instruction Cache Attacks. *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*. Ed. by S. Mangard and F.-X. Standaert. Vol. 6225. Lecture Notes in Computer Science. Springer, 2010, 110–124. DOI: 10.1007/978-3-642-15031-9_8. URL: https://doi.org/10.1007/978-3-642-15031-9_8.
- [5] O. Aciğmez, S. Gueron and J.-P. Seifert. New Branch Prediction Vulnerabilities in OpenSSL and Necessary Software Countermeasures. *Cryptog-*

- raphy and Coding, 11th IMA International Conference, Cirencester, UK, December 18-20, 2007, Proceedings*. Ed. by S. D. Galbraith. Vol. 4887. Lecture Notes in Computer Science. Springer, 2007, 185–203. DOI: 10.1007/978-3-540-77272-9_12. URL: https://doi.org/10.1007/978-3-540-77272-9_12.
- [6] O. Aciğmez, Ç. K. Koç and J.-P. Seifert. On the Power of Simple Branch Prediction Analysis. *Proceedings of the 2007 ACM Symposium on Information, Computer and Communications Security, AsiaCCS 2007, Singapore, March 20-22, 2007*. Ed. by F. Bao and S. Miller. ACM, 2007, 312–320. DOI: 10.1145/1229285.1266999. URL: <http://doi.acm.org/10.1145/1229285.1266999>.
- [7] O. Aciğmez, Ç. K. Koç and J.-P. Seifert. Predicting Secret Keys Via Branch Prediction. *Topics in Cryptology - CT-RSA 2007, The Cryptographers' Track at the RSA Conference 2007, San Francisco, CA, USA, February 5-9, 2007, Proceedings*. Ed. by M. Abe. Vol. 4377. Lecture Notes in Computer Science. Springer, 2007, 225–242. DOI: 10.1007/11967668_15. URL: https://doi.org/10.1007/11967668_15.
- [8] O. Aciğmez and J.-P. Seifert. Cheap Hardware Parallelism Implies Cheap Security. *Fourth International Workshop on Fault Diagnosis and Tolerance in Cryptography, 2007, FDTC 2007: Vienna, Austria, 10 September 2007*. Ed. by L. Breveglieri, S. Gueron, I. Koren, D. Naccache and J.-P. Seifert. IEEE Computer Society, 2007, 80–91. DOI: 10.1109/FDTC.2007.4318988. URL: <https://doi.org/10.1109/FDTC.2007.4318988>.
- [9] A. C. Aldaya and B. B. Brumley. HyperDegrade: From GHz to MHz Effective CPU Frequencies. *CoRR* abs/2101.01077 (2021). arXiv: 2101.01077. URL: <https://arxiv.org/abs/2101.01077>.
- [10] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. Pereida García and N. Tuveri. Port Contention for Fun and Profit. *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 2019, 870–887. DOI: 10.1109/SP.2019.00066.
- [11] A. C. Aldaya, A. J. Cabrera Sarmiento and S. Sánchez-Solano. SPA vulnerabilities of the binary extended Euclidean algorithm. *J. Cryptographic*

- Engineering* 7.4 (2017), 273–285. DOI: 10.1007/s13389-016-0135-4. URL: <https://doi.org/10.1007/s13389-016-0135-4>.
- [12] A. C. Aldaya, C. Pereida García, L. M. Alvarez Tapia and B. B. Brumley. Cache-Timing Attacks on RSA Key Generation. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019.4 (2019), 213–242. DOI: 10.13154/tches.v2019.i4.213-242.
- [13] A. C. Aldaya, C. Pereida García and B. B. Brumley. From A to Z: Projective coordinates leakage in the wild. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020.3 (2020), 428–453. DOI: 10.13154/tches.v2020.i3.428-453. URL: <https://doi.org/10.13154/tches.v2020.i3.428-453>.
- [14] T. Allan, B. B. Brumley, K. E. Falkner, J. van de Pol and Y. Yarom. Amplifying side channels through performance degradation. *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016*. Ed. by S. Schwab, W. K. Robertson and D. Balzarotti. ACM, 2016, 422–435. DOI: 10.1145/2991079.2991084. URL: <http://doi.acm.org/10.1145/2991079.2991084>.
- [15] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir and M. Emmi. Verifying Constant-Time Implementations. *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*. Ed. by T. Holz and S. Savage. USENIX Association, 2016, 53–70. URL: <https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida>.
- [16] M. Andryscio, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner and H. Shacham. On Subnormal Floating Point and Abnormal Timing. *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 2015, 623–639. DOI: 10.1109/SP.2015.44. URL: <https://doi.org/10.1109/SP.2015.44>.
- [17] S. Aravamuthan and V. R. Thumparthy. A Parallelization of ECDSA Resistant to Simple Power Analysis Attacks. *Proceedings of the Second International Conference on COMMunication System softWARE and MiddlewaRE (COMSWARE 2007), January 7-12, 2007, Bangalore, India*. Ed. by S. Paul, H. Schulzrinne and G. Venkatesh. IEEE, 2007. DOI:

10.1109/COMSWA.2007.382592. URL: <https://doi.org/10.1109/COMSWA.2007.382592>.

- [18] A. Arcangeli, I. Eidus and C. Wright. Increasing memory density by using KSM. *Proceedings of the Linux Symposium*. 2009, 19–28.
- [19] *ARM A64 Instruction Set Architecture ARMv8*. ARMv8 Architecture Profile. ARM Limited, June 2021.
- [20] *Armv8, for Armv8-A architecture profile*. Arm Architecture Reference Manual. ARM Limited, Jan. 2021.
- [21] G. Balakrishnan and T. W. Reps. WYSINWYX: What you see is not what you eXecute. *ACM Trans. Program. Lang. Syst.* 32.6 (2010), 23:1–23:84. DOI: 10.1145/1749608.1749612. URL: <https://doi.org/10.1145/1749608.1749612>.
- [22] D. Belyavsky, B. B. Brumley, J.-J. Chi-Domínguez, L. Rivera-Zamarripa and I. Ustinov. Set It and Forget It! Turnkey ECC for Instant Integration. *ACSAC '20: Annual Computer Security Applications Conference, Virtual Event / Austin, TX, USA, 7-11 December, 2020*. ACM, 2020, 760–771. DOI: 10.1145/3427228.3427291. URL: <https://doi.org/10.1145/3427228.3427291>.
- [23] N. Benger, J. van de Pol, N. P. Smart and Y. Yarom. “Ooh Aah... Just a Little Bit”: A Small Amount of Side Channel Can Go a Long Way. *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*. Ed. by L. Batina and M. Robshaw. Vol. 8731. Lecture Notes in Computer Science. Springer, 2014, 75–92. DOI: 10.1007/978-3-662-44709-3_5. URL: https://doi.org/10.1007/978-3-662-44709-3_5.
- [24] D. J. Bernstein. *Cache-timing attacks on AES*. 2005. URL: <http://cr.yp.to/papers.html#cachetiming>.
- [25] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe and B.-Y. Yang. High-speed high-security signatures. *J. Cryptographic Engineering* 2.2 (2012), 77–89. DOI: 10.1007/s13389-012-0027-1. URL: <https://doi.org/10.1007/s13389-012-0027-1>.

- [26] D. J. Bernstein and B.-Y. Yang. Fast constant-time gcd computation and modular inversion. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019.3 (2019), 340–398. DOI: 10.13154/tches.v2019.i3.340-398. URL: <https://doi.org/10.13154/tches.v2019.i3.340-398>.
- [27] B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. T. V. Setty and L. Thompson. Vale: Verifying High-Performance Cryptographic Assembly Code. *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. Ed. by E. Kirda and T. Ristenpart. USENIX Association, 2017, 917–934. URL: <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/bond>.
- [28] D. D. A. Braga, P.-A. Fouque and M. Sabt. PARASITE: PASSword Recovery Attack against Srp Implementations in ThE wild. *IACR Cryptol. ePrint Arch.* 2021 (2021), 553. URL: <https://eprint.iacr.org/2021/553>.
- [29] S. Briongos, P. Malagón, J. M. Moya and T. Eisenbarth. RELOAD+REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks. *CoRR* abs/1904.06278 (2019). arXiv: 1904.06278. URL: <http://arxiv.org/abs/1904.06278>.
- [30] B. B. Brumley and R. M. Hakala. Cache-Timing Template Attacks. *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings*. Ed. by M. Matsui. Vol. 5912. Lecture Notes in Computer Science. Springer, 2009, 667–684. DOI: 10.1007/978-3-642-10366-7_39. URL: https://doi.org/10.1007/978-3-642-10366-7_39.
- [31] B. B. Brumley and N. Tuveri. Cache-timing attacks and shared contexts. *Constructive Side-Channel Analysis and Secure Design - 2nd International Workshop, COSADE 2011, Darmstadt, Germany, February 24-25, 2011. Proceedings*. 2011, 233–242. URL: <https://researchportal.tuni.fi/files/15671512/cosade2011.pdf>.
- [32] B. B. Brumley and N. Tuveri. Remote Timing Attacks Are Still Practical. *Computer Security - ESORICS 2011 - 16th European Symposium on Re-*

- search in Computer Security, Leuven, Belgium, September 12-14, 2011. Proceedings.* Ed. by V. Atluri and C. Díaz. Vol. 6879. Lecture Notes in Computer Science. Springer, 2011, 355–371. DOI: 10.1007/978-3-642-23822-2_20. URL: https://doi.org/10.1007/978-3-642-23822-2_20.
- [33] D. Brumley and D. Boneh. Remote timing attacks are practical. *Computer Networks* 48.5 (2005), 701–716. DOI: 10.1016/j.comnet.2005.01.010. URL: <https://doi.org/10.1016/j.comnet.2005.01.010>.
- [34] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. V. Bulck and Y. Yarom. Fallout: Leaking Data on Meltdown-resistant CPUs. *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019.* Ed. by L. Cavallaro, J. Kinder, X. Wang and J. Katz. ACM, 2019, 769–784. DOI: 10.1145/3319535.3363219. URL: <https://doi.org/10.1145/3319535.3363219>.
- [35] C. Canella, K. N. Khasawneh and D. Gruss. The Evolution of Transient-Execution Attacks. *GLSVLSI '20: Great Lakes Symposium on VLSI 2020, Virtual Event, China, September 7-9, 2020.* Ed. by T. Mohsenin, W. Zhao, Y. Chen and O. Mutlu. ACM, 2020, 163–168. DOI: 10.1145/3386263.3407583. URL: <https://doi.org/10.1145/3386263.3407583>.
- [36] S. Cauligi, G. Soeller, F. Brown, B. Johannesmeyer, Y. Huang, R. Jhala and D. Stefan. FaCT: A Flexible, Constant-Time Programming Language. *IEEE Cybersecurity Development, SecDev 2017, Cambridge, MA, USA, September 24-26, 2017.* IEEE Computer Society, 2017, 69–76. DOI: 10.1109/SecDev.2017.24. URL: <https://doi.org/10.1109/SecDev.2017.24>.
- [37] D. Cerdeira, N. Santos, P. Fonseca and S. Pinto. SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems. *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020.* IEEE, 2020, 1416–1432. DOI: 10.1109/SP40000.2020.00061. URL: <https://doi.org/10.1109/SP40000.2020.00061>.
- [38] P.-Y. Chang, E. Hao and Y. N. Patt. Alternative implementations of hybrid branch predictors. *Proceedings of the 28th Annual International*

- Symposium on Microarchitecture, Ann Arbor, Michigan, USA, November 29 - December 1, 1995*. Ed. by T. N. Mudge and K. Ebcioglu. ACM / IEEE Computer Society, 1995, 252–257. DOI: 10.1109/MICRO.1995.476833. URL: <https://doi.org/10.1109/MICRO.1995.476833>.
- [39] M. Chiappetta, E. Savas and C. Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. *Appl. Soft Comput.* 49 (2016), 1162–1174. DOI: 10.1016/j.asoc.2016.09.014. URL: <https://doi.org/10.1016/j.asoc.2016.09.014>.
- [40] *CoreLink Level 2 Cache Controller L2C-310*. Tech. rep. ARM, 2010.
- [41] J.-S. Coron. Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems. *Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings*. Ed. by Ç. K. Koç and C. Paar. Vol. 1717. Lecture Notes in Computer Science. Springer, 1999, 292–302. DOI: 10.1007/3-540-48059-5_25. URL: https://doi.org/10.1007/3-540-48059-5_25.
- [42] G. Dessouky, T. Frassetto and A.-R. Sadeghi. HybCache: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments. *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. Ed. by S. Capkun and F. Roesner. USENIX Association, 2020, 451–468. URL: <https://www.usenix.org/conference/usenixsecurity20/presentation/dessouky>.
- [43] C. Disselkoen, D. Kohlbrenner, L. Porter and D. M. Tullsen. Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX. *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. Ed. by E. Kirda and T. Ristenpart. USENIX Association, 2017, 51–67. URL: <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/disselkoen>.
- [44] V. Dolmatov and A. Degtyarev. *GOST R 34.10-2012: Digital Signature Algorithm*. RFC 7091. RFC Editor, Dec. 2013, 1–21. DOI: 10.17487/RFC7091. URL: <https://datatracker.ietf.org/doc/rfc7091/>.

- [45] A. Erbsen, J. Philipoom, J. Gross, R. Sloan and A. Chlipala. Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises. *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 2019, 1202–1219. DOI: 10.1109/SP.2019.00005. URL: <https://doi.org/10.1109/SP.2019.00005>.
- [46] D. Evtvyushkin, R. Riley, N. B. Abu-Ghazaleh and D. Ponomarev. Branch-Scope: A New Side-Channel Attack on Directional Branch Predictor. *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*. Ed. by X. Shen, J. Tuck, R. Bianchini and V. Sarkar. ACM, 2018, 693–707. DOI: 10.1145/3173162.3173204. URL: <https://doi.org/10.1145/3173162.3173204>.
- [47] J. A. Fisher and S. M. Freudenberger. Predicting Conditional Branch Directions From Previous Runs of a Program. *ASPLOS-V Proceedings - Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, Massachusetts, USA, October 12-15, 1992*. Ed. by B. Flahive and R. L. Wexelblat. ACM Press, 1992, 85–95. DOI: 10.1145/143365.143493. URL: <https://doi.org/10.1145/143365.143493>.
- [48] B. Gras, K. Razavi, H. Bos and C. Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. Ed. by W. Enck and A. P. Felt. USENIX Association, 2018, 955–972. URL: <https://www.usenix.org/conference/usenixsecurity18/presentation/gras>.
- [49] B. Gras, K. Razavi, E. Bosman, H. Bos and C. Giuffrida. ASLR on the Line: Practical Cache Attacks on the MMU. *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017. URL: <https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/aslr-cache-practical-cache-attacks-mmu/>.

- [50] I. Gridin, C. Pereida García, N. Tuveri and B. B. Brumley. Triggerflow: Regression Testing by Advanced Execution Path Inspection. *Detection of Intrusions and Malware, and Vulnerability Assessment - 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19-20, 2019, Proceedings*. Ed. by R. Perdisci, C. Maurice, G. Giacinto and M. Almgren. Vol. 11543. Lecture Notes in Computer Science. Springer, 2019, 330–350. DOI: 10.1007/978-3-030-22038-9_16.
- [51] D. Grunwald and S. Ghiasi. Microarchitectural denial of service: insuring microarchitectural fairness. *Proceedings of the 35th Annual International Symposium on Microarchitecture, Istanbul, Turkey, November 18-22, 2002*. Ed. by E. R. Altman, K. Ebcioglu, S. A. Mahlke, B. R. Rau and S. J. Patel. ACM/IEEE Computer Society, 2002, 409–418. DOI: 10.1109/MICRO.2002.1176268. URL: <https://doi.org/10.1109/MICRO.2002.1176268>.
- [52] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller and M. Costa. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. Ed. by E. Kirda and T. Ristenpart. USENIX Association, 2017, 217–233. URL: <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/gruss>.
- [53] D. Gruss, C. Maurice and S. Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. *Detection of Intrusions and Malware, and Vulnerability Assessment - 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings*. Ed. by J. Caballero, U. Zurutuza and R. J. Rodríguez. Vol. 9721. Lecture Notes in Computer Science. Springer, 2016, 300–321. DOI: 10.1007/978-3-319-40667-1_15. URL: https://doi.org/10.1007/978-3-319-40667-1_15.
- [54] D. Gruss, C. Maurice, K. Wagner and S. Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. *Detection of Intrusions and Malware, and Vulnerability Assessment - 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings*. Ed. by J. Caballero, U. Zurutuza and R. J. Rodríguez. Vol. 9721. Lecture Notes in Computer

- Science. Springer, 2016, 279–299. DOI: 10.1007/978-3-319-40667-1_14. URL: https://doi.org/10.1007/978-3-319-40667-1_14.
- [55] D. Gruss, R. Spreitzer and S. Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12–14, 2015*. Ed. by J. Jung and T. Holz. USENIX Association, 2015, 897–912. URL: <https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/gruss>.
- [56] S. Gueron. Intel’s New AES Instructions for Enhanced Performance and Security. *Fast Software Encryption, 16th International Workshop, FSE 2009, Leuven, Belgium, February 22–25, 2009, Revised Selected Papers*. Ed. by O. Dunkelman. Vol. 5665. Lecture Notes in Computer Science. Springer, 2009, 51–66. DOI: 10.1007/978-3-642-03317-9_4. URL: https://doi.org/10.1007/978-3-642-03317-9_4.
- [57] S. Gueron and V. Krasnov. Fast prime field elliptic-curve cryptography with 256-bit primes. *J. Cryptographic Engineering* 5.2 (2015), 141–151. DOI: 10.1007/s13389-014-0090-x. URL: <https://doi.org/10.1007/s13389-014-0090-x>.
- [58] *Guidelines for Mitigating Timing Side Channels Against Cryptographic Implementations*. Tech. rep. Intel, Sept. 2019. URL: <https://software.intel.com/content/www/us/en/develop/articles/software-security-guidance/secure-coding/mitigate-timing-side-channel-crypto-implementation.html>.
- [59] D. Gullasch, E. Bangerter and S. Krenn. Cache Games - Bringing Access-Based Cache Attacks on AES to Practice. *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22–25 May 2011, Berkeley, California, USA*. IEEE Computer Society, 2011, 490–505. DOI: 10.1109/SP.2011.22. URL: <https://doi.org/10.1109/SP.2011.22>.
- [60] M. Hamburg. Ed448-Goldilocks, a new elliptic curve. *IACR Cryptol. ePrint Arch.* 2015 (2015), 625. URL: <http://eprint.iacr.org/2015/625>.
- [61] J. Hasan, A. Jalote, T. N. Vijaykumar and C. E. Brodley. Heat Stroke: Power-Density-Based Denial of Service in SMT. *11th International Conference on High-Performance Computer Architecture (HPCA-11 2005)*,

- 12-16 February 2005, San Francisco, CA, USA. IEEE Computer Society, 2005, 166–177. DOI: 10.1109/HPCA.2005.16. URL: <https://doi.org/10.1109/HPCA.2005.16>.
- [62] S. ul Hassan, I. Gridin, I. M. Delgado-Lozano, C. Pereida García, J.-J. Chi-Domínguez, A. C. Aldaya and B. B. Brumley. Déjà Vu: Side-Channel Analysis of Mozilla’s NSS. *CCS ’20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*. Ed. by J. Ligatti, X. Ou, J. Katz and G. Vigna. ACM, 2020, 1887–1902. DOI: 10.1145/3372297.3421761. URL: <https://doi.org/10.1145/3372297.3421761>.
- [63] W. Henecka, A. May and A. Meurer. Correcting Errors in RSA Private Keys. *Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings*. Ed. by T. Rabin. Vol. 6223. Lecture Notes in Computer Science. Springer, 2010, 351–369. DOI: 10.1007/978-3-642-14623-7_19. URL: https://doi.org/10.1007/978-3-642-14623-7_19.
- [64] N. Heninger and H. Shacham. Reconstructing RSA Private Keys from Random Key Bits. *Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings*. Ed. by S. Halevi. Vol. 5677. Lecture Notes in Computer Science. Springer, 2009, 1–17. DOI: 10.1007/978-3-642-03356-8_1. URL: https://doi.org/10.1007/978-3-642-03356-8_1.
- [65] N. Howgrave-Graham and N. P. Smart. Lattice Attacks on Digital Signature Schemes. *Des. Codes Cryptogr.* 23.3 (2001), 283–290. DOI: 10.1023/A:1011214926272. URL: <https://doi.org/10.1023/A:1011214926272>.
- [66] R. Hund, C. Willems and T. Holz. Practical Timing Side Channel Attacks Against Kernel Space ASLR. *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*. The Internet Society, 2013. URL: <https://www.ndss-symposium.org/ndss2013/practical-timing-side-channel-attacks-against-kernel-space-aslr>.
- [67] M. S. Inci, B. Gülmezoglu, G. I. Apecechea, T. Eisenbarth and B. Sunar. Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Pub-

- lic Cloud. *IACR Cryptol. ePrint Arch.* 2015 (2015), 898. URL: <http://eprint.iacr.org/2015/898>.
- [68] M. S. Inci, B. Gülmezoglu, G. Irazoqui, T. Eisenbarth and B. Sunar. Cache Attacks Enable Bulk Key Recovery on the Cloud. *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*. Ed. by B. Gierlichs and A. Y. Poschmann. Vol. 9813. Lecture Notes in Computer Science. Springer, 2016, 368–388. DOI: 10.1007/978-3-662-53140-2_18. URL: https://doi.org/10.1007/978-3-662-53140-2_18.
- [69] *Intel 64 and IA-32 Architectures Software Developer’s Manual*. Volume 1. Intel, May 2021. URL: <https://software.intel.com/sites/default/files/managed/a4/60/253665-sdm-vol-1.pdf>.
- [70] *Intel 64 and IA-32 Architectures Software Developer’s Manual Combined Volumes*. Volume 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4. Intel, May 2021. URL: <https://www.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html>.
- [71] G. Irazoqui, T. Eisenbarth and B. Sunar. S\$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing - and Its Application to AES. *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 2015, 591–604. DOI: 10.1109/SP.2015.42. URL: <https://doi.org/10.1109/SP.2015.42>.
- [72] G. Irazoqui, T. Eisenbarth and B. Sunar. Systematic Reverse Engineering of Cache Slice Selection in Intel Processors. *2015 Euromicro Conference on Digital System Design, DSD 2015, Madeira, Portugal, August 26-28, 2015*. IEEE Computer Society, 2015, 629–636. DOI: 10.1109/DSD.2015.56. URL: <https://doi.org/10.1109/DSD.2015.56>.
- [73] G. Irazoqui, T. Eisenbarth and B. Sunar. Cross Processor Cache Attacks. *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi’an, China, May 30 - June 3, 2016*. Ed. by X. Chen, X. Wang and X. Huang. ACM, 2016, 353–364.

DOI: 10.1145/2897845.2897867. URL: <https://doi.org/10.1145/2897845.2897867>.

- [74] A. Jaleel, K. B. Theobald, S. C. S. Jr. and J. S. Emer. High performance cache replacement using re-reference interval prediction (RRIP). *37th International Symposium on Computer Architecture (ISCA 2010)*, June 19-23, 2010, Saint-Malo, France. Ed. by A. Seznev, U. C. Weiser and R. Ronen. ACM, 2010, 60–71. DOI: 10.1145/1815961.1815971. URL: <https://doi.org/10.1145/1815961.1815971>.
- [75] D. A. Jiménez. Improved latency and accuracy for neural branch prediction. *ACM Trans. Comput. Syst.* 23.2 (2005), 197–218. DOI: 10.1145/1062247.1062250. URL: <https://doi.org/10.1145/1062247.1062250>.
- [76] D. A. Jiménez and C. Lin. Neural methods for dynamic branch prediction. *ACM Trans. Comput. Syst.* 20.4 (2002), 369–397. DOI: 10.1145/571637.571639. URL: <https://doi.org/10.1145/571637.571639>.
- [77] S. Josefsson and S. Leonard. *Textual Encodings of PKIX, PKCS, and CMS Structures*. RFC 7468. RFC Editor, Apr. 2015, 1–20. DOI: 10.17487/RFC7468. URL: <https://datatracker.ietf.org/doc/rfc7468/>.
- [78] E. Käsper. Fast Elliptic Curve Cryptography in OpenSSL. *Financial Cryptography and Data Security - FC 2011 Workshops, RLCPS and WECSR 2011, Rodney Bay, St. Lucia, February 28 - March 4, 2011, Revised Selected Papers*. Ed. by G. Danezis, S. Dietrich and K. Sako. Vol. 7126. Lecture Notes in Computer Science. Springer, 2011, 27–39. DOI: 10.1007/978-3-642-29889-9_4. URL: https://doi.org/10.1007/978-3-642-29889-9_4.
- [79] Y. Kim, R. Daly, J. S. Kim, C. Fallin, J.-H. Lee, D. Lee, C. Wilkerson, K. Lai and O. Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*. IEEE Computer Society, 2014, 361–372. DOI: 10.1109/ISCA.2014.6853210. URL: <https://doi.org/10.1109/ISCA.2014.6853210>.

- [80] V. Kiriansky, I. A. Lebedev, S. P. Amarasinghe, S. Devadas and J. S. Emer. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. *51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*. IEEE Computer Society, 2018, 974–987. DOI: 10.1109/MICRO.2018.00083. URL: <https://doi.org/10.1109/MICRO.2018.00083>.
- [81] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz and Y. Yarom. Spectre Attacks: Exploiting Speculative Execution. *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 2019, 1–19. DOI: 10.1109/SP.2019.00002. URL: <https://doi.org/10.1109/SP.2019.00002>.
- [82] J. Kong, O. Aciicmez, J.-P. Seifert and H. Zhou. Hardware-software integrated approaches to defend against software cache-based side channel attacks. *15th International Conference on High-Performance Computer Architecture (HPCA-15 2009), 14-18 February 2009, Raleigh, North Carolina, USA*. IEEE Computer Society, 2009, 393–404. DOI: 10.1109/HPCA.2009.4798277. URL: <https://doi.org/10.1109/HPCA.2009.4798277>.
- [83] A. Kwong, D. Genkin, D. Gruss and Y. Yarom. RAMBleed: Reading Bits in Memory Without Accessing Them. *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, 695–711. DOI: 10.1109/SP40000.2020.00020. URL: <https://doi.org/10.1109/SP40000.2020.00020>.
- [84] A. Langley. *ctgrind—checking that functions are constant time with Valgrind*. <https://github.com/agl/ctgrind>. 2010.
- [85] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice and S. Mangard. ARMageddon: Cache Attacks on Mobile Devices. *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*. Ed. by T. Holz and S. Savage. USENIX Association, 2016, 549–564. URL: <https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/lipp>.
- [86] M. Lipp, V. Hazic, M. Schwarz, A. Perais, C. Maurice and D. Gruss. Take A Way: Exploring the Security Implications of AMD’s Cache Way

- Predictors. *ASIA CCS '20: The 15th ACM Asia Conference on Computer and Communications Security, Taipei, Taiwan, October 5-9, 2020*. Ed. by H.-M. Sun, S.-P. Shieh, G. Gu and G. Ateniese. ACM, 2020, 813–825. DOI: 10.1145/3320269.3384746. URL: <https://doi.org/10.1145/3320269.3384746>.
- [87] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom and M. Hamburg. Melt-down: Reading Kernel Memory from User Space. *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. Ed. by W. Enck and A. P. Felt. USENIX Association, 2018, 973–990. URL: <https://www.usenix.org/conference/usenixsecurity18/presentation/lipp>.
- [88] F. Liu, H. Wu, K. Mai and R. B. Lee. Newcache: Secure Cache Architecture Thwarting Cache Side-Channel Attacks. *IEEE Micro* 36.5 (2016), 8–16. DOI: 10.1109/MM.2016.85. URL: <https://doi.org/10.1109/MM.2016.85>.
- [89] F. Liu, Y. Yarom, Q. Ge, G. Heiser and R. B. Lee. Last-Level Cache Side-Channel Attacks are Practical. *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 2015, 605–622. DOI: 10.1109/SP.2015.43. URL: <https://doi.org/10.1109/SP.2015.43>.
- [90] D. Maimut, C. Murdica, D. Naccache and M. Tibouchi. Fault Attacks on Projective-to-Affine Coordinates Conversion. *Constructive Side-Channel Analysis and Secure Design - 4th International Workshop, COSADE 2013, Paris, France, March 6-8, 2013, Revised Selected Papers*. Ed. by E. Prouff. Vol. 7864. Lecture Notes in Computer Science. Springer, 2013, 46–61. DOI: 10.1007/978-3-642-40026-1_4. URL: https://doi.org/10.1007/978-3-642-40026-1_4.
- [91] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller and M. Upton. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal* 6.1 (2002). ISSN: 1535864X.
- [92] C. Maurice, C. Neumann, O. Heen and A. Francillon. C5: Cross-Cores Cache Covert Channel. *Detection of Intrusions and Malware, and Vulner-*

- ability Assessment - 12th International Conference, DIMVA 2015, Milan, Italy, July 9-10, 2015, Proceedings*. Ed. by M. Almgren, V. Gulisano and F. Maggi. Vol. 9148. Lecture Notes in Computer Science. Springer, 2015, 46–64. DOI: 10.1007/978-3-319-20550-2_3. URL: https://doi.org/10.1007/978-3-319-20550-2_3.
- [93] C. Maurice, N. L. Scouarnec, C. Neumann, O. Heen and A. Francillon. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters. *Research in Attacks, Intrusions, and Defenses - 18th International Symposium, RAID 2015, Kyoto, Japan, November 2-4, 2015, Proceedings*. Ed. by H. Bos, F. Monrose and G. Blanc. Vol. 9404. Lecture Notes in Computer Science. Springer, 2015, 48–65. DOI: 10.1007/978-3-319-26362-5_3. URL: https://doi.org/10.1007/978-3-319-26362-5_3.
- [94] R. Merget, M. Brinkmann, N. Aviram, J. Somorovsky, J. Mittmann and J. Schwenk. Raccoon Attack: Finding and Exploiting Most-Significant-Bit-Oracles in TLS-DH(E). *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021. URL: <https://www.usenix.org/conference/usenixsecurity21/presentation/merget>.
- [95] J. Merkle and M. Lochter. *Elliptic Curve Cryptography (ECC) Brainpool Standard Curves and Curve Generation*. RFC 5639. RFC Editor, Mar. 2010, 1–27. DOI: 10.17487/RFC5639. URL: <https://datatracker.ietf.org/doc/rfc5639/>.
- [96] P. L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Math. Comp.* 48.177 (1987), 243–264. ISSN: 0025-5718. DOI: 10.2307/2007888. URL: <https://doi.org/10.2307/2007888>.
- [97] K. Moriarty, B. Kaliski, J. Jonsson and A. Rusch. *PKCS #1: RSA Cryptography Specifications Version 2.2*. RFC 8017. RFC Editor, Nov. 2016, 1–78. DOI: 10.17487/RFC8017. URL: <https://datatracker.ietf.org/doc/rfc8017/>.
- [98] M. Mushtaq, A. Akram, M. K. Bhatti, M. Chaudhry, V. Lapotre and G. Gogniat. NIGHTs-WATCH: a cache-based side-channel intrusion detector using hardware performance counters. *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Pri-*

- vacy, HASP@ISCA 2018, Los Angeles, CA, USA, June 02-02, 2018*. Ed. by J. Szefer, W. Shi and R. B. Lee. ACM, 2018, 1:1–1:8. DOI: 10.1145/3214292.3214293. URL: <https://doi.org/10.1145/3214292.3214293>.
- [99] D. Naccache, N. P. Smart and J. Stern. Projective Coordinates Leak. *Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004, Proceedings*. Ed. by C. Cachin and J. Camenisch. Vol. 3027. Lecture Notes in Computer Science. Springer, 2004, 257–267. DOI: 10.1007/978-3-540-24676-3_16. URL: https://doi.org/10.1007/978-3-540-24676-3_16.
- [100] M. Nemeč, M. Šýs, P. Svenda, D. Klinec and V. Matyas. The Return of Coppersmith’s Attack: Practical Factorization of Widely Used RSA Moduli. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. Ed. by B. M. Thuraisingham, D. Evans, T. Malkin and D. Xu. ACM, 2017, 1631–1648. DOI: 10.1145/3133956.3133969. URL: <http://doi.acm.org/10.1145/3133956.3133969>.
- [101] P. Q. Nguyen and I. E. Shparlinski. The Insecurity of the Elliptic Curve Digital Signature Algorithm with Partially Known Nonces. *Des. Codes Cryptogr.* 30.2 (2003), 201–217. DOI: 10.1023/A:1025436905711. URL: <https://doi.org/10.1023/A:1025436905711>.
- [102] Y. Oren, V. P. Kemerlis, S. Sethumadhavan and A. D. Keromytis. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*. Ed. by I. Ray, N. Li and C. Kruegel. ACM, 2015, 1406–1418. DOI: 10.1145/2810103.2813708. URL: <https://doi.org/10.1145/2810103.2813708>.
- [103] D. A. Osvik, A. Shamir and E. Tromer. Cache Attacks and Countermeasures: The Case of AES. *Topics in Cryptology - CT-RSA 2006, The Cryptographers’ Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings*. Ed. by D. Pointcheval. Vol. 3860. Lec-

- ture Notes in Computer Science. Springer, 2006, 1–20. DOI: 10.1007/11605805_1. URL: https://doi.org/10.1007/11605805_1.
- [104] D. Page. Partitioned Cache Architecture as a Side-Channel Defence Mechanism. *IACR Cryptol. ePrint Arch.* 2005 (2005), 280. URL: <http://eprint.iacr.org/2005/280>.
- [105] C. Percival. Cache Missing for Fun and Profit. *BSDCan 2005, Ottawa, Canada, May 13-14, 2005, Proceedings.* 2005. URL: <http://www.daemonology.net/papers/cachemissing.pdf>.
- [106] C. Pereida García and B. B. Brumley. Constant-Time Callees with Variable-Time Callers. *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.* Ed. by E. Kirda and T. Ristenpart. USENIX Association, 2017, 83–98. ISBN: 978-1-931971-40-9. URL: <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/garcia>.
- [107] C. Pereida García, B. B. Brumley and Y. Yarom. “Make Sure DSA Signing Exponentiations Really are Constant-Time”. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016.* Ed. by E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers and S. Halevi. ACM, 2016, 1639–1650. DOI: 10.1145/2976749.2978420.
- [108] C. Pereida García, S. ul Hassan, N. Tuveri, I. Gridin, A. C. Aldaya and B. B. Brumley. Certified Side Channels. *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020.* Ed. by S. Capkun and F. Roesner. USENIX Association, 2020, 2021–2038. URL: <https://www.usenix.org/conference/usenixsecurity20/presentation/garcia>.
- [109] J. Protzenko, B. Parno, A. Fromherz, C. Hawblitzel, M. Polubelova, K. Bhargavan, B. Beurdouche, J. Choi, A. Delignat-Lavaud, C. Fournet, N. Kulatova, T. Ramananandro, A. Rastogi, N. Swamy, C. M. Wintersteiger and S. Z. Béguelin. EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider. *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020.* IEEE, 2020, 983–1002. DOI: 10.1109/SP40000.2020.00114. URL: <https://doi.org/10.1109/SP40000.2020.00114>.

- [110] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. S. Jr. and J. S. Emer. Adaptive insertion policies for high performance caching. *34th International Symposium on Computer Architecture (ISCA 2007), June 9-13, 2007, San Diego, California, USA*. Ed. by D. M. Tullsen and B. Calder. ACM, 2007, 381–391. DOI: 10.1145/1250662.1250709. URL: <https://doi.org/10.1145/1250662.1250709>.
- [111] O. Reparaz, J. Balasch and I. Verbauwhede. Dude, is my code constant time?: *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017*. Ed. by D. Aienza and G. D. Natale. IEEE, 2017, 1697–1702. DOI: 10.23919/DATE.2017.7927267. URL: <https://doi.org/10.23919/DATE.2017.7927267>.
- [112] T. Ristenpart, E. Tromer, H. Shacham and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*. Ed. by E. Al-Shaer, S. Jha and A. D. Keromytis. ACM, 2009, 199–212. DOI: 10.1145/1653662.1653687. URL: <http://doi.acm.org/10.1145/1653662.1653687>.
- [113] B. Rodrigues, F. M. Q. Pereira and D. F. Aranha. Sparse representation of implicit flows with applications to side-channel detection. *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*. Ed. by A. Zaks and M. V. Hermenegildo. ACM, 2016, 110–120. DOI: 10.1145/2892208.2892230. URL: <http://doi.acm.org/10.1145/2892208.2892230>.
- [114] R. Ronen, A. Mendelson, K. Lai, S.-L. Lu, F. J. Pollack and J. P. Shen. Coming challenges in microarchitecture and architecture. *Proc. IEEE* 89.3 (2001), 325–340. DOI: 10.1109/5.915377. URL: <https://doi.org/10.1109/5.915377>.
- [115] A. Saini. Design of the Intel PentiumTM Processor. *Proceedings 1993 International Conference on Computer Design: VLSI in Computers & Processors, ICCD '93, Cambridge, MA, USA, October 3-6, 1993*. IEEE Computer Society, 1993, 258–261. DOI: 10.1109/ICCD.1993.393370. URL: <https://doi.org/10.1109/ICCD.1993.393370>.

- [116] M. Schwarz and D. Gruss. How Trusted Execution Environments Fuel Research on Microarchitectural Attacks. *IEEE Secur. Priv.* 18.5 (2020), 18–27. DOI: 10.1109/MSEC.2020.2993896. URL: <https://doi.org/10.1109/MSEC.2020.2993896>.
- [117] M. Schwarz, S. Weiser, D. Gruss, C. Maurice and S. Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. *Detection of Intrusions and Malware, and Vulnerability Assessment - 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings*. Ed. by M. Polychronakis and M. Meier. Vol. 10327. Lecture Notes in Computer Science. Springer, 2017, 3–24. DOI: 10.1007/978-3-319-60876-1_1. URL: https://doi.org/10.1007/978-3-319-60876-1_1.
- [118] *SEC 1: Elliptic Curve Cryptography*. SEC 1. Standards for Efficient Cryptography Group, May 2009. URL: <http://www.secg.org/sec1-v2.pdf>.
- [119] *SEC 2: Recommended Elliptic Curve Domain Parameters*. SEC 2. Standards for Efficient Cryptography Group, Jan. 2010. URL: <http://www.secg.org/sec2-v2.pdf>.
- [120] R. Seggelmann, M. Tüxen and M. G. Williams. Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension. *RFC 6520* (2012), 1–9. DOI: 10.17487/RFC6520. URL: <https://doi.org/10.17487/RFC6520>.
- [121] A. Sez nec. Tage-sc-l branch predictors. *JILP-Championship Branch Prediction*. 2014.
- [122] D. Shanks. Class number, a theory of factorization, and genera. *Proc. of Symp. Math. Soc., 1971*. Vol. 20. 1971, 41–440.
- [123] P. Shrestha, M. Mohamed and N. Saxena. Slogger: Smashing Motion-based Touchstroke Logging with Transparent System Noise. *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks, WISEC 2016, Darmstadt, Germany, July 18-22, 2016*. Ed. by M. Hollick, P. Papadimitratos and W. Enck. ACM, 2016, 67–77. DOI: 10.1145/2939918.2939924. URL: <https://doi.org/10.1145/2939918.2939924>.

- [124] L. Simon, D. Chisnall and R. J. Anderson. What You Get is What You C: Controlling Side Effects in Mainstream C Compilers. *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*. IEEE, 2018, 1–15. DOI: 10.1109/EuroSP.2018.00009. URL: <https://doi.org/10.1109/EuroSP.2018.00009>.
- [125] R. Spreitzer and B. Gérard. Towards More Practical Time-Driven Cache Attacks. *Information Security Theory and Practice. Securing the Internet of Things - 8th IFIP WG 11.2 International Workshop, WISTP 2014, Heraklion, Crete, Greece, June 30 - July 2, 2014. Proceedings*. Ed. by D. Naccache and D. Sauveron. Vol. 8501. Lecture Notes in Computer Science. Springer, 2014, 24–39. DOI: 10.1007/978-3-662-43826-8_3. URL: https://doi.org/10.1007/978-3-662-43826-8%5C_3.
- [126] R. Spreitzer and T. Plos. Cache-Access Pattern Attack on Disaligned AES T-Tables. *Constructive Side-Channel Analysis and Secure Design - 4th International Workshop, COSADE 2013, Paris, France, March 6-8, 2013, Revised Selected Papers*. Ed. by E. Prouff. Vol. 7864. Lecture Notes in Computer Science. Springer, 2013, 200–214. DOI: 10.1007/978-3-642-40026-1_13. URL: https://doi.org/10.1007/978-3-642-40026-1_13.
- [127] R. Spreitzer and T. Plos. On the Applicability of Time-Driven Cache Attacks on Mobile Devices. *Network and System Security - 7th International Conference, NSS 2013, Madrid, Spain, June 3-4, 2013. Proceedings*. Ed. by J. López, X. Huang and R. S. Sandhu. Vol. 7873. Lecture Notes in Computer Science. Springer, 2013, 656–662. DOI: 10.1007/978-3-642-38631-2_53. URL: https://doi.org/10.1007/978-3-642-38631-2%5C_53.
- [128] Y. Tan, J. Wei and W. Guo. The Micro-architectural Support Countermeasures against the Branch Prediction Analysis Attack. *13th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2014, Beijing, China, September 24-26, 2014*. IEEE Computer Society, 2014, 276–283. DOI: 10.1109/TrustCom.2014.38. URL: <https://doi.org/10.1109/TrustCom.2014.38>.

- [129] N. Tuveri, S. ul Hassan, C. Pereida García and B. B. Brumley. *Electromagnetic (EM) side-channel traces of elliptic curve point multiplication during SM2 decryption in OpenSSL*. Zenodo, Sept. 2018. DOI: 10.5281/zenodo.1436828. URL: <https://doi.org/10.5281/zenodo.1436828>.
- [130] N. Tuveri, S. ul Hassan, C. Pereida García and B. B. Brumley. Side-Channel Analysis of SM2: A Late-Stage Featurization Case Study. *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. ACM, 2018, 147–160. DOI: 10.1145/3274694.3274725.
- [131] B. C. Vattikonda, S. Das and H. Shacham. Eliminating fine grained timers in Xen. *Proceedings of the 3rd ACM Cloud Computing Security Workshop, CCSW 2011, Chicago, IL, USA, October 21, 2011*. Ed. by C. Cachin and T. Ristenpart. ACM, 2011, 41–46. DOI: 10.1145/2046660.2046671. URL: <https://doi.org/10.1145/2046660.2046671>.
- [132] P. Vila, P. Ganty, M. Guarnieri and B. Köpf. CacheQuery: learning replacement policies from hardware caches. *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*. Ed. by A. F. Donaldson and E. Torlak. ACM, 2020, 519–532. DOI: 10.1145/3385412.3386008. URL: <https://doi.org/10.1145/3385412.3386008>.
- [133] P. Vila, B. Köpf and J. F. Morales. Theory and Practice of Finding Eviction Sets. *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 2019, 39–54. DOI: 10.1109/SP.2019.00042. URL: <https://doi.org/10.1109/SP.2019.00042>.
- [134] I. Vougioukas, N. Nikoleris, A. Sandberg, S. Diestelhorst, B. M. Al-Hashimi and G. V. Merrett. BRB: Mitigating Branch Predictor Side-Channels. *25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019, Washington, DC, USA, February 16-20, 2019*. IEEE, 2019, 466–477. DOI: 10.1109/HPCA.2019.00058. URL: <https://doi.org/10.1109/HPCA.2019.00058>.
- [135] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. *5th Symposium on Operating System Design and Implementation (OSDI 2002), Boston, Massachusetts, USA, December 9-11, 2002*. Ed. by

- D. E. Culler and P. Druschel. USENIX Association, 2002. URL: <http://www.usenix.org/events/osdi02/tech/waldspurger.html>.
- [136] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. *34th International Symposium on Computer Architecture (ISCA 2007), June 9-13, 2007, San Diego, California, USA*. Ed. by D. M. Tullsen and B. Calder. ACM, 2007, 494–505. DOI: 10.1145/1250662.1250723. URL: <https://doi.org/10.1145/1250662.1250723>.
- [137] S. Weiser, R. Spreitzer and L. Bodner. Single Trace Attack Against RSA Key Generation in Intel SGX SSL. *Proceedings of the 2018 on Asia Conference on Computer and Communications Security, AsiaCCS 2018, Incheon, Republic of Korea, June 04-08, 2018*. Ed. by J. Kim, G.-J. Ahn, S. Kim, Y. Kim, J. López and T. Kim. ACM, 2018, 575–586. DOI: 10.1145/3196494.3196524. URL: <http://doi.acm.org/10.1145/3196494.3196524>.
- [138] S. Weiser, A. Zankl, R. Spreitzer, K. Miller, S. Mangard and G. Sigl. DATA - Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries. *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. Ed. by W. Enck and A. P. Felt. USENIX Association, 2018, 603–620. URL: <https://www.usenix.org/conference/usenixsecurity18/presentation/weiser>.
- [139] L. Weissbart, S. Picek and L. Batina. One Trace Is All It Takes: Machine Learning-Based Side-Channel Attack on EdDSA. *Security, Privacy, and Applied Cryptography Engineering - 9th International Conference, SPACE 2019, Gandhinagar, India, December 3-7, 2019, Proceedings*. Ed. by S. Bhasin, A. Mendelson and M. Nandi. Vol. 11947. Lecture Notes in Computer Science. Springer, 2019, 86–105. DOI: 10.1007/978-3-030-35869-3_8. URL: https://doi.org/10.1007/978-3-030-35869-3_8.
- [140] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss and S. Mangard. ScatterCache: Thwarting Cache Attacks via Cache Set Randomization. *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*. Ed. by N. Heninger and

- P. Traynor. USENIX Association, 2019, 675–692. URL: <https://www.usenix.org/conference/usenixsecurity19/presentation/werner>.
- [141] J. Wichelmann, A. Moghimi, T. Eisenbarth and B. Sunar. MicroWalk: A Framework for Finding Side Channels in Binaries. *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. ACM, 2018, 161–173. DOI: 10.1145/3274694.3274741. URL: <https://doi.org/10.1145/3274694.3274741>.
- [142] M. Wu, S. Guo, P. Schaumont and C. Wang. Eliminating timing side-channel leaks using program repair. *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*. Ed. by F. Tip and E. Bodden. ACM, 2018, 15–26. DOI: 10.1145/3213846.3213851. URL: <https://doi.org/10.1145/3213846.3213851>.
- [143] M. Yan, R. Sprabery, B. Gopireddy, C. W. Fletcher, R. H. Campbell and J. Torrellas. Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World. *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 2019, 888–904. DOI: 10.1109/SP.2019.00004. URL: <https://doi.org/10.1109/SP.2019.00004>.
- [144] Y. Yarom and N. Benger. Recovering OpenSSL ECDSA Nonces Using the FLUSH+RELOAD Cache Side-channel Attack. *IACR Cryptol. ePrint Arch.* 2014 (2014), 140. URL: <http://eprint.iacr.org/2014/140>.
- [145] Y. Yarom and K. Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. USENIX Association, 2014, 719–732. ISBN: 978-1-931971-15-7. URL: <https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom>.
- [146] Y. Yarom, Q. Ge, F. Liu, R. B. Lee and G. Heiser. Mapping the Intel Last-Level Cache. *IACR Cryptol. ePrint Arch.* 2015 (2015), 905. URL: <http://eprint.iacr.org/2015/905>.

- [147] Y. Yarom, D. Genkin and N. Heninger. CacheBleed: A Timing Attack on OpenSSL Constant Time RSA. *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*. Ed. by B. Gierlichs and A. Y. Poschmann. Vol. 9813. Lecture Notes in Computer Science. Springer, 2016, 346–367. DOI: 10.1007/978-3-662-53140-2_17. URL: https://doi.org/10.1007/978-3-662-53140-2_17.
- [148] F. Zhang and H. Zhang. SoK: A Study of Using Hardware-assisted Isolated Execution Environments for Security. *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016, HASP@ICSA 2016, Seoul, Republic of Korea, June 18, 2016*. ACM, 2016, 3:1–3:8. DOI: 10.1145/2948618.2948621. URL: <https://doi.org/10.1145/2948618.2948621>.
- [149] L. Zhang, N. Wu, F. Ge, F. Zhou and M. R. Yahya. A Dynamic Branch Predictor Based on Parallel Structure of SRNN. *IEEE Access* 8 (2020), 86230–86237. DOI: 10.1109/ACCESS.2020.2992643. URL: <https://doi.org/10.1109/ACCESS.2020.2992643>.
- [150] Y. Zhang, A. Juels, M. K. Reiter and T. Ristenpart. Cross-VM side channels and their use to extract private keys. *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*. Ed. by T. Yu, G. Danezis and V. D. Gligor. ACM, 2012, 305–316. DOI: 10.1145/2382196.2382230. URL: <https://doi.org/10.1145/2382196.2382230>.
- [151] J. K. Zinzindohoué, K. Bhargavan, J. Protzenko and B. Beurdouche. HACl*: A Verified Modern Cryptographic Library. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. Ed. by B. M. Thuraisingham, D. Evans, T. Malkin and D. Xu. ACM, 2017, 1789–1806. DOI: 10.1145/3133956.3134043. URL: <http://doi.acm.org/10.1145/3133956.3134043>.

PUBLICATIONS

PUBLICATION

I

**“Make Sure DSA Signing Exponentiations Really are
Constant-Time”**

C. Pereida García, B. B. Brumley and Y. Yarom

*Proceedings of the 2016 ACM SIGSAC Conference on Computer and
Communications Security, Vienna, Austria, October 24-28, 2016.* Ed. by
E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers and S. Halevi. 2016,
1639–1650

DOI: 10.1145/2976749.2978420

Publication reprinted with the permission of the copyright holders

“Make Sure DSA Signing Exponentiations Really are Constant-Time”

Cesar Pereida García
Department of Computer
Science
Aalto University, Finland
cesar.pereida@aalto.fi

Billy Bob Brumley
Department of Pervasive
Computing
Tampere University of
Technology, Finland
billy.brumley@tut.fi

Yuval Yarom
The University of Adelaide and
Data61, CSIRO, Australia
yval@cs.adelaide.edu.au

ABSTRACT

TLS and SSH are two of the most commonly used protocols for securing Internet traffic. Many of the implementations of these protocols rely on the cryptographic primitives provided in the OpenSSL library. In this work we disclose a vulnerability in OpenSSL, affecting all versions and forks (e.g. LibreSSL and BoringSSL) since roughly October 2005, which renders the implementation of the DSA signature scheme vulnerable to cache-based side-channel attacks. Exploiting the software defect, we demonstrate the first published cache-based key-recovery attack on these protocols: 260 SSH-2 handshakes to extract a 1024/160-bit DSA host key from an OpenSSH server, and 580 TLS 1.2 handshakes to extract a 2048/256-bit DSA key from a stunnel server.

Keywords

applied cryptography; digital signatures; side-channel analysis; timing attacks; cache-timing attacks; DSA; OpenSSL; CVE-2016-2178

1. INTRODUCTION

One of the contributing factors to the explosion of the Internet in the last decade is the security provided by the underlying cryptographic protocols. Two of those protocols are the Transport Layer Security (TLS) protocol, which provides security to network communication and the more specialized Secure Shell (SSH), which provides secure login to remote hosts.

Software implementations of these protocols often use the cryptographic primitives’ implementations of the OpenSSL cryptographic library. Consequently, the security of these implementations depends on the security of OpenSSL.

In this paper we present a novel side-channel cache-timing attack against OpenSSL’s DSA implementation. The attack exploits a vulnerability in OpenSSL, which fails to use a side-channel-secure implementation of modular exponentiation — the core mathematical operation used in DSA signatures.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored.

CCS ’16 October 24–28, 2016, Vienna, Austria

© 2016 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-4139-4/16/10.

DOI: <http://dx.doi.org/10.1145/2976749.2978420>



Our attack builds upon several techniques to profile the cache memory and capture timing signals. The signals are processed and converted into a sequence of square and multiplication (SM) operations from which we extract information to create a lattice problem. The solution to the lattice problem yields the secret key of digital signatures.

FLUSH+RELOAD [40] is a powerful technique to perform cache-timing attacks. We adapt the FLUSH+RELOAD technique to OpenSSL’s implementation of DSA and, exploiting properties of the Intel implementation of the x86 and x64 processor architectures, our spy program probes relevant memory addresses to create a signal trace.

We process the captured signal to get the SM sequence performed by the *sliding window exponentiation* (SWE) algorithm. Then we observe and analyze the number of bits that can be extracted and used from each of those sequences. Later, the variable amount of bits extracted from each trace is used as input to a lattice attack that recovers the private key.

To bridge the gap between the limited resolution of the FLUSH+RELOAD technique [4] and the high-performance of the OpenSSL code we apply the performance-degradation technique of Allan et al. [4]. This technique slows the exponentiation by an average factor of 20, giving a high resolution trace and allowing us to extract up to 8 bits of information from some of the traces.

Similar to previous works [9, 14, 21, 32], we perform a lattice attack to recover the secret key. We use the lattice construction of Bengier et al. [9] and solve the resulting lattice problem using the lattice reduction technique of Nguyen and Shparlinski [28].

A unique feature of our work is that we target common cryptographic protocols. Previous works that demonstrate cache-timing key-recovery attacks only target the cryptographic primitives, ignoring potential cache noise from the protocol implementation. In contrast, we present end-to-end attacks on two common cryptographic protocols: SSH and TLS. We are, therefore, the first to demonstrate that cache-timing attacks are a threat not only when executing the cryptographic primitives but also in the presence of the cache activity of the whole protocol suite.

Our contributions in this work are the following:

- We identify a security weakness in OpenSSL which fails to use a side-channel safe implementation when performing DSA signatures. (Section 3)
- We describe how to use a combination of the FLUSH+

RELOAD technique with a performance-degradation attack to leak information from the unsafe SWE algorithm. (Section 4)

- We present the first key-recovery cache-timing attack on the TLS and SSH cryptographic protocols. (Section 5)
- We construct and solve a lattice problem with the side-channel information and the digital signatures in order to recover the secret key. (Section 6)

2. BACKGROUND

2.1 Memory Hierarchy

Accessing data and instructions from main memory is a time consuming operation which delays the work of the fast processors, for that reason the memory hierarchy includes smaller and faster memories called *caches*. Caches improve the performance by exploiting the spatial and temporal locality of the memory access.

In modern processors the hierarchy of caches is structured as follows, higher-level caches, located closer to the processor core, are smaller and faster than low-level caches, which are located closer to main memory. Recent Intel architecture typically has three levels of cache: L1, L2 and Last-Level Cache (LLC).

Each core has two L1 caches, a data cache and an instruction cache, each 32 KiB in size with an access time of 4 cycles. L2 caches are also core-private and have an intermediate size (256 KiB) and latency (7 cycles). The LLC is shared among all of the cores and is a unified cache, containing both data and instructions. Typical LLC sizes are in megabytes and access time is in the order of 40 cycles.

The unit of memory and allocation in a cache is called *cache line*. Cache lines are of a fixed size B , which is typically 64 bytes. The $\lg(B)$ low-order bits of the address, called *line offset*, are used to locate the datum in the cache line.

When a memory address is accessed, the processor checks the availability of the address line in the top-level L1 cache. If the data is there then it is served to the processor, a situation referred to as a *cache hit*. In a *cache miss*, when the data is not found in the L1 cache, the processor repeats the search for the line in the next cache level and continues through all the caches. Once the line is found, the processor stores a copy in the cache for future use.

Most caches are *set-associative*. They are composed of S *cache sets* each containing a fixed number of cache lines. The number of cache lines in a set is the *cache associativity*, i.e., a cache with W lines in each set is a W -way *set-associative* cache.

Since the main memory is orders of magnitude larger than the cache, more than W memory lines may map to the same cache set. If a cache miss occurs and all the cache lines in the matching cache set are in use, one of the cached lines is evicted, freeing a slot for a new line to be fetched from a lower-level memory. Several *cache replacement policies* exist to determine the cache line to evict when a cache miss occurs but the typical policy in use is an approximation to the least-recently-used (LRU).

The last-level cache in modern Intel processors is *inclusive*. Inclusive caches contain a superset of the contents of the

cache levels above them. In the case of Intel processors, the contents of the L1 and L2 caches is also stored in the last-level cache. A consequence of the inclusion property is that when data is evicted from the last-level cache it is also evicted from all of the other levels of cache in the processor.

Intel architecture implements several cache optimizations. The spatial pre-fetcher pairs cache lines and attempt to fetch the pair of a missed line [17]. Consecutive accesses to memory addresses are detected and pre-fetched when the processor anticipates they may be required [17]. Additionally, when the processor is presented with a conditional branch, *speculative execution* brings the data of both branches into the cache before the branch condition is evaluated [35].

Page [30] noted that tracing the sequence of cache hits and misses of software may leak information on the internal working of the software, including information that may lead to recovering cryptographic keys.

This idea was later extended and used for mounting several cache-based side-channel attacks [10, 29, 31]. Other attacks were shown against the L1-instruction cache [3], the branch prediction buffer [1, 2] and the last-level cache [20, 22, 25, 40].

2.2 The FLUSH+RELOAD Attack

Our LLC-based attack is based on the FLUSH+RELOAD [20, 40] attack, which is a cache-based side-channel attack technique.

Unlike the earlier PRIME+PROBE technique [29, 31] that detects activity in cache sets, the FLUSH+RELOAD technique identifies access to memory lines, giving it a higher resolution, a high accuracy and high signal-to-noise ratio.

Like PRIME+PROBE, FLUSH+RELOAD relies on cache sharing between processes. Additionally, it requires data sharing, which is typically achieved through the use of shared libraries or using page de-duplication [6, 36].

A round of the attack, which identifies victim access to a shared memory line, consists of three phases. (See Algorithm 1.) In the first phase the adversary evicts the monitored memory line from the cache. In the second phase, the adversary waits a period of time so the victim has an opportunity to access the memory line. In the third phase, the adversary measures the time it takes to reload the memory line. If during the second phase the victim accesses the memory line, the line will be available in the cache and the reload operation in the third phase will take a short time. If, on the other hand, the victim does not access the memory line then the third phase takes a longer time as the memory line is loaded from main memory.

Algorithm 1: FLUSH+RELOAD Attack

Input: Memory Address $addr$.

Result: True if the victim accessed the address.

```

begin
  flush(addr)
  wait for the victim.
  time ← current_time()
  tmp ← read(addr)
  readTime ← current_time() - time
  return readTime < threshold

```

The execution of the victim and the adversary processes are independent of each other, thus synchronization of prob-

ing is important and several factors need to be considered when processing the side-channel data. Some of those factors are the waiting period for the adversary between probes, memory lines to be probed, size of the side-channel trace and *cache-hit* threshold. One important goal for this attack is to achieve the best resolution possible while keeping the error rate low and one of the ways to achieve this is by targeting memory lines that occur frequently during execution, such as loop bodies. Several processor optimizations are in place during a typical process execution and an attacker must be aware of these optimizations to filter them during the analysis of the attack results. See [4, 39, 40] for discussions of some of these parameters.

A typical implementation of the FLUSH+RELOAD attack makes use of the `clflush` instruction of the x86 and x64 instruction sets. The `clflush` instruction evicts a specific memory line from *all* the cache hierarchy and being an unprivileged instruction, it can be used by any process.

The inclusiveness of the LLC is essential for the FLUSH+RELOAD attack. Whenever a memory line is evicted from the LLC, the processor also evicts the line from all of the L1 and L2 caches. On processors that do not have an inclusive LLC, e.g., AMD processors, the attack does not work [40]. See, however, Lipp et al. [24] for a variant of the technique that does not require an inclusive LLC.

2.3 The Digital Signature Algorithm (DSA)

A variant of the ElGamal signature scheme, DSA was first proposed by the U.S. National Institute of Standards and Technology (NIST). DSA uses the multiplicative group of a finite field. We use the following notation for DSA.

Parameters: Primes p, q such that q divides $(p - 1)$, a generator g of multiplicative order q in $GF(p)$ and an approved hash function h (e.g. SHA-1, SHA-256, SHA-512).

Private-Public key pairs: The private key α is an integer uniformly chosen such that $0 < \alpha < q$ and the corresponding public key y is given by $y = g^\alpha \bmod p$. Calculating the private key given the public key requires solving the discrete logarithm problem and for correctly chosen parameters, this is an intractable problem.

Signing: A given party, Alice, wants to send a signed message m to Bob—the message m is not necessarily encrypted. Using her private-public key pair (α_A, y_A) , Alice performs the following steps:

1. Select uniformly at random a secret nonce k such that $0 < k < q$.
2. Compute $r = (g^k \bmod p) \bmod q$ and $h(m)$.
3. Compute $s = k^{-1}(h(m) + \alpha_A r) \bmod q$.
4. Alice sends (m, r, s) to Bob.

Verifying: Bob wants to be sure the message he received comes from Alice—a valid DSA signature gives strong evidence of authenticity. Bob performs the following steps to verify the signature:

1. Reject the signature if it does not satisfy $0 < r < q$ and $0 < s < q$.
2. Compute $w = s^{-1} \bmod q$ and $h(m)$.
3. Compute $u_1 = h(m)w \bmod q$ and $u_2 = rw \bmod q$.

4. Compute $v = (g^{u_1} y_A^{u_2} \bmod p) \bmod q$.

5. Accept the signature if and only if $v = r$ holds.

2.3.1 DSA in Practice

Putting it mildly, there is no consensus on key sizes, and furthermore keys seen in the wild and used in ubiquitous protocols have varying sizes—sometimes dictated by existing and deployed standards. For example, NIST defines 1024-bit p with 160-bit q as “legacy-use” and 2048-bit p with 256-bit q as “acceptable” [8]. We focus on these two parameter sets.

SSH’s Transport Layer Protocol¹ lists DSA key type `ssh-dss` as “required” and defines r and s as 160-bit integers, implying 160-bit q . In fact the OpenSSH tool `ssh-keygen` defaults to 160-bit q and 1024-bit p for these key types, not allowing the user to override that option, and using the same parameters to generate the server’s host key. It is worth noting that recently as of version 7.0, OpenSSH disables host server DSA keys by a configurable default option², but of course this does not affect already deployed solutions.

As a countermeasure to previous timing attacks, OpenSSL’s DSA implementation pads nonces by adding either q or $2q$ to k —see details in Section 3.

For the DSA signing algorithm, Step 2 is the performance bottleneck and the exponentiation algorithm used will prove to be of extreme importance when we later collect our side-channel information in Section 4.

2.4 Sliding Window Exponentiation

Sliding window exponentiation (SWE) is a widely implemented software method to perform integer exponentiations, e.g. featured alongside other methods in the OpenSSL codebase. SWE is fairly popular due to its performance since it reduces the amount of pre-computation needed and, moreover, reduces the average amount of multiplications performed during the exponentiation.

An exponent e is represented and processed as a sequence of windows e_i , each of length $L(e_i)$ bits. Processing the exponent in windows reduces the amount of multiplications at the cost of increased memory utilization since a table of pre-computed values is used.

A window e_i can be a zero window represented as a string of “0”s or non-zero window represented as a string starting and ending with “1”s and such window is of width w (determined in OpenSSL by the size in bits of the exponent e). The length of non-zero windows satisfy $1 \leq L(e_i) \leq w$, thus the value of any given non-zero window is an odd number between 1 and $2^w - 1$.

As mentioned before, the algorithm pre-computes values and stores them in a table to be used later during multiplication operations. The multipliers computed are $b^v \bmod m$ for each odd value of v where $1 \leq v \leq 2^w - 1$ and these values are stored in table index $g[i]$ where $i = (v - 1)/2$. For example, with the standard 160-bit q size, OpenSSL uses a window width $w = 4$, the algorithm pre-computes multipliers $b^1, b^3, b^5, \dots, b^{15} \bmod m$ and stores them in $g[0], g[1], g[2], \dots, g[7]$, respectively.

Using the SWE representation of the exponent e , Algorithm 2 computes the corresponding exponentiation through a combination of squares and multiplications in a left-to-right approach. The algorithm scans every window e_i from

¹<https://tools.ietf.org/html/rfc4253>

²<http://www.openssh.com/legacy.html>

Algorithm 2: Sliding window exponentiation.

Input: Window size w , base b , modulo m , N -bit exponent e represented as n windows e_i , each of length $L(e_i)$.
Output: $b^e \bmod m$.
// Pre-computation
 $g[0] \leftarrow b \bmod m$;
 $s \leftarrow \text{MULT}(g[0], g[0]) \bmod m$;
for $j \leftarrow 1$ **to** $2^{w-1} - 1$ **do**
 $g[j] \leftarrow \text{MULT}(g[j-1], s) \bmod m$;
// Exponentiation
 $r \leftarrow 1$;
for $i \leftarrow n$ **to** 1 **do**
 for $j \leftarrow 1$ **to** $L(e_i)$ **do**
 $r \leftarrow \text{MULT}(r, r) \bmod m$;
 if $e_i \neq 0$ **then** $r \leftarrow \text{MULT}(r, g[(e_i - 1)/2]) \bmod m$;
return r ;

the most significant bit (MSB) to the least significant bit (LSB).

For any window, a square operation is executed for each bit and additionally for a non-zero window, the algorithm executes an extra multiplication when it reaches the LSB of the window.

For novel reasons explained later in Section 3, the side-channel part of our attack focuses on this algorithm. Specifically, in getting the sequence of squares and multiplies performed during its execution. Then we extract partial information from the sequence for later use in the lattice attack.

2.5 Partial key disclosure

Recall that the nonce k and the secret key α satisfy the following linear congruence.

$$s = k^{-1}(h(m) + \alpha r) \bmod q$$

The constants of the linear combination are specified by s , $h(m)$, and r , which, typically for a signed message, are all public. Hence, knowing the nonce k reveals the secret key α .

Typically, side-channel leakage from SWE only recovers partial information about the nonce. The adversary, therefore, has to use that partial information to recover the key. The usual technique for recovering the secret key from the partial information is to express the problem as a *hidden number problem* [12] which is solved using a lattice technique.

2.5.1 The hidden number problem

In the hidden number problem (HNP) the task is to find a hidden number given some of the MSBs of several modular linear combinations of the hidden number. More specifically, the problem is to find a secret number α given a number of triples (t_i, u_i, ℓ_i) such that for $v_i = |\alpha \cdot t_i - u_i|_q$ we have $|v_i| \leq q/2^{\ell_i+1}$, where $|\cdot|_q$ is the reduction modulo q into the range $(-q/2, \dots, q/2)$.

Boneh and Venkatesan [12] initially investigate HNP with a constant $\ell_i = \ell$. They show that for $\ell < \log^{1/2} q + \log \log q$ and random t_i , the hidden number α can be recovered given a number of triples linear in $\log q$.

Howgrave-Graham and Smart [21] extend the work of

Boneh and Venkatesan [12] showing how to construct an HNP instance from leaked LSBs and MSBs of DSA nonces. Nguyen and Shparlinski [27] prove that for a good enough hash function and for a linear number of randomly chosen nonces, knowing the ℓ LSBs of a certain number of nonces, the $\ell + 1$ MSBs or $2 \cdot \ell$ consecutive bits anywhere in the nonces is enough for recovering the long term key α . They further demonstrate that a DSA-160 key can be broken if only the 3 LSBs of a certain number of nonces are known. Nguyen and Shparlinski [28] extend the results to ECDSA, and Liu and Nguyen [26] demonstrate that only 2 LSBs are required for breaking a DSA-160 key. Bengier et al. [9] extend the technique to use a different number of leaked LSBs for each signature.

2.5.2 Lattice attack

To find the hidden number from the triples we solve a lattice problem. The construction of the lattice problem presented here is due to Bengier et al. [9], and is based on the constructions in earlier publications [12, 27].

Given d triples, we construct a $d + 1$ -dimensional lattice using the rows of the matrix

$$B = \begin{pmatrix} 2^{\ell_1+1} \cdot q & & & & \\ & \ddots & & & \\ & & 2^{\ell_d+1} \cdot q & & \\ 2^{\ell_1+1} \cdot t_1 & \dots & 2^{\ell_d+1} \cdot t_d & & 1 \end{pmatrix}.$$

By the definition of v_i , there are integers λ_i such that $v_i = \lambda_i \cdot q + \alpha \cdot t_i - u_i$. Consequently, for the vectors $\mathbf{x} = (\lambda_1, \dots, \lambda_d, \alpha)$, $\mathbf{y} = (2^{\ell_1+1} \cdot v_1, \dots, 2^{\ell_d+1} \cdot v_d, \alpha)$ and $\mathbf{u} = (2^{\ell_1+1} \cdot u_1, \dots, 2^{\ell_d+1} \cdot u_d, 0)$ we have

$$\mathbf{x} \cdot B - \mathbf{u} = \mathbf{y}.$$

The 2-norm of the vector \mathbf{y} is about $\sqrt{d+1} \cdot q$ whereas the determinant of the lattice $L(B)$ is $2^{d+\sum \ell_i} \cdot q^d$. Hence \mathbf{y} is a short vector in the lattice and the vector \mathbf{u} is close to the lattice vector $\mathbf{x} \cdot B$. We can now solve the Closest Vector Problem (CVP) with inputs B and \mathbf{u} to find \mathbf{x} , revealing the value of the hidden number α .

2.5.3 Related Work

Several authors describe attacks on cryptographic systems that exploit partial nonce disclosure to recover long-term private keys.

Brumley and Hakala [14] use an L1 data cache-timing attack to recover the LSBs of ECDSA nonces from the `dgst` command line tool in OpenSSL 0.9.8k. They collect 2,600 signatures (8K with noise) and use the Howgrave-Graham and Smart [21] attack to recover a 160-bit ECDSA private key. In a similar vein, Acıçmez et al. [3] use an L1 instruction cache-timing attack to recover the LSBs of DSA nonces from the same tool in OpenSSL 0.9.8l, requiring 2,400 signatures (17K with noise) to recover a 160-bit DSA private key. Both attacks require HyperThreading architectures.

Brumley and Tuveri [15] mount a remote timing attack on the implementation of ECDSA with binary curves in OpenSSL 0.9.8o. They show that the timing leaks information on the MSBs of the nonce used and that after collecting that information over 8,000 TLS handshakes the private key can be recovered.

Bengier et al. [9] recover the secret key of OpenSSL's ECDSA implementation for the curve `secp256k1` using less than

256 signatures. They use the FLUSH+RELOAD technique to find some LSBs of the nonces and extend the lattice technique of Howgrave-Graham and Smart [21] to use all of the leaked bits rather than limiting to a fixed number.

Van de Pol et al. [32] exploit the structure of the modulus in some elliptic curves to use all of the information leaked in consecutive sequences of bits anywhere in the top half of the nonces, allowing them to recover the secret key after observing only a handful of signatures. Allan et al. [4] improve on these results by using a performance-degradation attack to amplify the side-channel. The amplification allows them to observe the sign bit in the w NAF representation used in OpenSSL 1.0.2a and to recover a 256 bit key after observing only 6 signatures.

Genkin et al. [19] perform electromagnetic and power analysis attacks on mobile phones. They show how to construct HNP triples when the signature uses the low s -value [38].

3. A NEW SOFTWARE DEFECT

Percival [31] demonstrated that the SWE implementation of modular exponentiation in OpenSSL version 0.9.7g is vulnerable to cache-timing attacks, applied to recover RSA private keys. Following the issue, the OpenSSL team committed two code changes relevant to this work. The first³ adds a “constant-time” implementation of modular exponentiation, with a fixed-window implementation and using the scatter-gather method [13, 41] of masking table access to the multipliers.

The new implementation is slower than the original SWE implementation. To avoid using the slower new code when the exponent is not secret, OpenSSL added a flag (`BN_FLG_CONSTTIME`) to its representation of big integers. When the exponent should remain secret (e.g. in decryption and signing) the flag is set (e.g. in the case of DSA nonces, Figure 1, Line 252) at runtime and the exponentiation code takes the “constant-time” execution path (Figure 2, Line 413). Otherwise, the original SWE implementation is used.

The execution time of the “constant-time” implementation still depends on the bit length of the exponent, which in the case of DSA should be kept secret [12, 15, 27]. The second commit⁴ aims to “make sure DSA signing exponentiations really are constant-time” by ensuring that the bit length of the exponent is fixed. This safe default behavior can be disabled by applications enabling the `DSA_FLAG_NO_EXP_CONSTTIME` flag at runtime within the DSA structure, although we are not aware of any such cases.

To get a fixed bit length, the DSA implementation adds γq to the randomly chosen nonce, where $\gamma \in \{1, 2\}$, such that the bit length of the sum is one more than the bit length of q . More precisely, the implementation creates a copy of the nonce k (Figure 1, Line 264), adds q to it (Line 274), checks if the bit length of the sum is one more than that of q (Line 276), otherwise it adds q again to the sum (Line 277). If q is n bits, then $k + q$ is either n or $n + 1$ bits. In the former case, indeed $k + 2q$ is $n + 1$ bits. As an aside, we note the code in question is not constant-time and potentially leaks the value of γ . Such a leak would create a bias that can be exploited to mount the Bleichenbacher attack [5, 11, 18].

³<https://github.com/openssl/openssl/commit/46a643763de6d8e39ecf6f76fa79b4d04885aa59>

⁴<https://github.com/openssl/openssl/commit/0ebfcc8f92736c900bae4066040b67f6e5db8ed8>

While the procedure in this commit ensures that the bit length of the sum kq is fixed, unfortunately it introduces a software defect. The function `BN_copy` is not designed to propagate flags from the source to the destination. In fact, OpenSSL exposes a distinct API `BN_with_flags` for that functionality—quoting the documentation:

```
BN_with_flags creates a temporary shallow copy of b in dest . . . Any flags provided in flags will be set in dest in addition to any flags already set in b. For example this might commonly be used to create a temporary copy of a BIGNUM with the BN_FLG_CONSTTIME flag set for constant time operations.
```

In contrast, with `BN_copy` the `BN_FLG_CONSTTIME` flag does not propagate to kq . Consequently, the sum is not treated as secret, reverting the change made in the first commit—when the exponentiation wrapper subsequently gets called (Figure 1, Line 285), it fails the security-critical branch. Following a debug session in Figure 2, indeed the flag (explicit value `0x4`) is not set, and the execution skips the call to `BN_mod_exp_mont_consttime` and instead continues with the insecure SWE code path for DSA exponentiation.

In addition to testing our attack against OpenSSL (1.0.2h), we reviewed the code of two popular OpenSSL forks: LibreSSL⁵ and BoringSSL⁶. Using builds with debugging symbols, we confirm both LibreSSL⁷ and BoringSSL⁸ share the same defect. It is worth noting that BoringSSL stripped out TLS DSA cipher suites in late 2014⁹.

4. EXPLOITING THE DEFECT

In this section we describe how we use and combine the FLUSH+RELOAD technique with a performance degradation technique [4] to attack the OpenSSL implementation of DSA.

We tested our attack on an Intel Core i5-4570 Haswell Quad-Core 3.2GHz (22nm) with 16GB of memory running 64-bit Ubuntu 14.04 LTS “Trusty”. Each core has an 8-way 32KB L1 data cache, an 8-way 32KB L1 instruction cache, an 8-way 256KB L2 unified cache, and all the cores share a 12-way 6MB unified LLC (all with 64B lines). It does not feature HyperThreading.

We used our own build of OpenSSL 1.0.2h which is the same default build of OpenSSL but with debugging symbols on the executable. Debugging symbols facilitate mapping source code to memory addresses but they are not loaded during run time, thus the victim’s performance is not affected. Debugging symbols are, typically, not available to attackers but using reverse engineering techniques [16] is possible to map source code to memory addresses.

As previously discussed in Section 2.5, for DSA-type signatures, knowing a few bits of sufficiently many signature nonces allows an attacker to recover the secret key. This is the goal of our attack: we trace and recover side-channel information of the SWE algorithm that reveals the sequence

⁵<https://www.libressl.org>

⁶<https://boringssl.googleusercontent.com/boringssl>

⁷https://github.com/libressl-portable/openbsd/blob/master/src/lib/libssl/src/crypto/dsa/dsa_oss.c

⁸<https://boringssl.googleusercontent.com/boringssl/+master/crypto/dsa/dsa.c>

⁹<https://boringssl.googleusercontent.com/boringssl/+ef2116d33c3c1b38005eb59caa2aa6300a9b450>

```

246 /* Get random k */
247 do
248     if (!BN_rand_range(&k, dsa->q))
249         goto err;
250 while (!BN_is_zero(&k));
251 if ((dsa->flags & DSA_FLAG_NO_EXP_CONSTTIME) == 0) {
252     BN_set_flags(&k, BN_FLG_CONSTTIME);
253 }
254 ...
255
256
257
258 if ((dsa->flags & DSA_FLAG_NO_EXP_CONSTTIME) == 0) {
259     if (!BN_copy(&kq, &k))
260         goto err;
261 }
262 /*
263  * We do not want timing information to leak the length of k, so we
264  * compute g^k using an equivalent exponent of fixed length. (This
265  * is a kludge that we need because the BN_mod_exp_mont() does not
266  * let us specify the desired timing behaviour.)
267  */
268 if (!BN_add(&kq, &kq, dsa->q))
269     goto err;
270 if (BN_num_bits(&kq) <= BN_num_bits(dsa->q)) {
271     if (!BN_add(&kq, &kq, dsa->q))
272         goto err;
273 }
274 }
275 K = &kq;
276 } else {
277     K = &k;
278 }
279 DSA_BN_MOD_EXP(goto err, dsa, r, dsa->g, K, dsa->p, ctx,
280                dsa->method_mont_p);

```

Figure 1: Excerpt from OpenSSL’s `dsa_sign_setup` in `crypto/dsa/dsa_ossl.c`. Line 252 sets the `BN_FLG_CONSTTIME` flag, yet `BN_copy` on Line 264 does not propagate it. The subsequent Line 285 exponentiation call will have pointer `K` with the flag clear.

```

+--bn_exp.c-----
1402 int BN_mod_exp_mont(BIGNUM *rr, const BIGNUM *a, const BIGNUM *p,
1403                    const BIGNUM *m, BN_CTX *ctx, BN_MONT_CTX *in_mont)
1404 {
B+ 1405     int i, j, bits, ret = 0, wstart, vend, window, sval;
1406     int start = 1;
1407     BIGNUM *d, *r;
1408     const BIGNUM *aa;
1409     /* Table of variables obtained from 'ctx' */
1410     BIGNUM *val[TABLE_SIZE];
1411     BN_MONT_CTX *mont = NULL;
1412
1413     if (BN_get_flags(p, BN_FLG_CONSTTIME) != 0) {
1414         return BN_mod_exp_mont_consttime(rr, a, p, m, ctx, in_mont);
1415     }
1416
1417     bn_check_top(a);
-----
|0x7ffff79db3e <BN_mod_exp_mont+92>    mov     0x14(%rax),%eax
|0x7ffff79db41 <BN_mod_exp_mont+95>    and     $0x4,%eax
|0x7ffff79db44 <BN_mod_exp_mont+98>    test   %eax,%eax
|0x7ffff79db46 <BN_mod_exp_mont+100>   je      0x7ffff79db85 <BN_mod_exp_mont+163>
|0x7ffff79db48 <BN_mod_exp_mont+102>   mov     -0x1b0(%rbp),%r8
-----
child process 29096 In: BN_mod_exp_mont          Line: 413 PC: 0x7ffff79db46
(gdb) break BN_mod_exp_mont
Breakpoint 1 (BN_mod_exp_mont) pending.
(gdb) run dgst -ds1 -sign /dsa.pem -out /lab-release.sig /etc/lab-release
Starting program: /usr/local/ssl/bin/openssl \
dgst -ds1 -sign /dsa.pem -out /lab-release.sig /etc/lab-release
Breakpoint 1, BN_mod_exp_mont (...) at bn_exp.c:405
(gdb) backtrace
#0  BN_mod_exp_mont (...) at bn_exp.c:405
#1  0x00007ffff7ee62 in dsa_sign_setup (...) at dsa_ossl.c:285
#2  0x00007ffff7ee344 in DSA_sign_setup (...) at dsa_ossl.c:87
#3  0x00007ffff7ee344 in dsa_do_sign (...) at dsa_ossl.c:159
#4  0x00007ffff7ee30c in DSA_do_sign (...) at dsa_ossl.c:75
...
(gdb) stepi
(gdb) info register eax
eax             0x0
(gdb) print BN_get_flags(p, BN_FLG_CONSTTIME)
$1 = 0
(gdb) macro expand BN_get_flags(p, BN_FLG_CONSTTIME)
expands to: ((p)->flags&(0x04))
(gdb) print (p)->flags&(0x04)
$2 = 0
(gdb)

```

Figure 2: Debugging OpenSSL DSA signing in `crypto/bn/bn_exp.c`. The Line 413 branch is not taken since `BN_FLG_CONSTTIME` is not set, as seen from the print command outputs. Hence `BN_mod_exp_mont_consttime` is not called—the control flow continues with classical SWE code.

of *squares* and *multiplications*, from that sequence we recover a few bits that we use for the lattice attack described in Section 6.

As seen in Figure 2, every time OpenSSL performs a DSA signature, the exponentiation method `BN_mod_exp_mont` in `crypto/bn/bn_exp.c` gets called. There, the `BN_FLG_CONSTTIME` flag is checked but due to the software defect discussed in Section 3 the condition fails and the routine continues with the SWE pre-computation and then the actual exponentiation. For the finite field operations, `BN_mod_exp_mont` calls `BN_mod_mul_montgomery` in `crypto/bn/bn_mont.c` and from there, the multiply wrapper `bn_mul_mont` is called, where, by default for x64 targets, assembly code is executed to perform low level operations using BIGNUMs for square and multiplication. OpenSSL uses Montgomery representation for efficiency. Note that for other platforms and/or non-default build configurations, the actual code executed ranges from pure C implementation to entirely different assembly. The attacker can easily adapt to these different execution paths, but the discussion that follows is geared towards our target platform.

The threshold set for the load time in the FLUSH+RELOAD technique (cache hit vs. cache miss) is system and software dependent. From our measurements we set this threshold accordingly since the load times from LLC and from memory were clearly defined. Figure 4 shows that loads from LLC take less than 100 cycles, while loads from main memory take more than 200 cycles.

As mentioned before, to get better resolution and granularity during the attack one effective strategy is to target body loops or routines that are invoked several times. For that reason we probe, using the FLUSH+RELOAD technique, inner routines used for square and multiply. Since squares can be computed more efficiently than multiplication, OpenSSL’s multiply wrapper checks if the two pointer operands are the same and, if so, calls to assembly squaring code (`bn_sqr8x_mont`)—otherwise, to assembly multiply code (`bn_mul4x_mont`).

At the same time we run a performance degradation attack, flushing actively used memory addresses during these routines (e.g. assembly labels `Lsqr4x_inner` and `Linner4x`, respectively). We slow down the execution time to a safe, but not noticeable by the victim, threshold that ensures a good trace by our spy program. In our experiments, we observe slow down factors of roughly 16 and 26 for 1024-bit and 2048-bit DSA, respectively due to the degrade technique.

Using this strategy, our spy program collects data from two channels: one for square latencies and the other for multiply latencies. We then apply signal processing techniques to this raw channel data. A moving average filter on the data results in Figure 3 and Figure 4 for 1024-bit and 2048-bit DSA, respectively. There is a significant amount of information to extract from these signals on the SWE algorithm state transitions and hence exponent bit values. Generally, extracted multiplications yield a single bit of information and the squares yield the position for these bits. Some short examples follow.

Stepping through Figure 3, the initial low amplitude for the multiply signal is the multiplication for converting the base operand to Montgomery representation. The subsequent low amplitude for the square signal is the temporary square value used to build the odd powers for the SWE pre-computation table (i.e. `s` in Algorithm 2). The subsequent

long period of low multiply amplitude is the successive multiplications to build the pre-computation table itself. Then begins the main loop of SWE. As an upward sloping multiply amplitude intersects a downward sloping square amplitude, this marks the transition from a multiplication operation to a square operation (and vice versa). This naturally occurs several times as the main exponentiation loop iterates. The end of this particular signal shows a final transition from multiply to a single square, indicating that the exponent is even and the two LSBs are 1 and 0.

Stepping through Figure 4 is similar, yet the end of this particular signal shows a final transition from square to multiply—indicating that the exponent is odd, i.e. the LSB is 1.

Even when employing the degrade technique, it is important to observe the vast granularity difference between these two cryptographic settings. On average, a 2048-bit signal is roughly ten times the length of a 1024-bit signal, even when the exponent is only 60% longer (i.e. 256-bit vs. 160-bit). This generally suggests we should be able to extract more accurate information from 2048-bit signals than 1024-bit—i.e., the higher security cryptographic parameters are more vulnerable to side-channel attack in this case. See [37, 39, 40] for similar examples of this phenomenon.

Granularity is vital to determining the number of squares interleaved between multiplications. Since, in our environment, there appears to be no reliable indicator in the signal for transitions from one square to the next, we estimate the number of adjacent squares by the horizontal distance between multiplications. Since the channel is latency data, we also have reference clock cycle counter values so another estimate is based on the counter differences at these points. Our experiments showed no significant advantage of one approach over the other.

Extracting the multiplications from the signal and interleaving them with a number of consecutive squares proportional to the width of the corresponding gap gives us the square and multiplication sequence, or *SM sequence*, that the SWE algorithm passed through. Figure 7 shows an example of an SM sequence recorded by the spy program when OpenSSL signs using 2048-bit DSA.

Our spy program is able to capture most of the SM sequence accurately. It can miss or duplicate a few squares due to drift but is able to capture all of the multiplication operations. Closer to the LSBs, the information extracted from the SM sequence is more reliable since the bit position is lost if any square operation is missed during probing.

5. VICTIMIZING APPLICATIONS

The defect from the previous section is in a shared library. Potentially any application that links against OpenSSL for DSA functionality can be affected by this vulnerability. But to make our attack concrete, we focus on two ubiquitous protocols and applications: TLS within stunnel and SSH within OpenSSH.

As we discuss later in Section 6, the trace data alone is not enough for private key recovery—we also need the digital signatures themselves and (hashed) messages. To this end, the goal of this section is to describe the practical tooling we developed to exploit the defect within these applications, collecting both trace data and protocol messages.

5.1 Attacking TLS

To feature TLS support, one option for network applications that do not natively support TLS communication is to use stunnel¹⁰, a popular portable open source software package that forwards network connections from one port to another and provides a TLS wrapper. A typical stunnel use case is listening on a public port to expose a TLS-enabled network service, then connecting to a localhost port where a non-TLS network service is listening—stunnel provides a TLS layer between the two ports. It links against the OpenSSL shared library to provide this functionality. For our experiments, we used stunnel 5.32 compiled from stock source and linked against OpenSSL 1.0.2h. We generated a 2048-bit DSA certificate for the stunnel service and chose the `DHE-DSS-AES128-SHA256` TLS 1.2 cipher suite.

We wrote a custom TLS client that connects to this stunnel service. It launches our spy to collect the timing signals, but its main purpose is to carry out the TLS handshake and collect the digital signatures and protocol messages. Figure 5 shows the TLS handshake. Relevant to this work, the initial `ClientHello` message contains a 32-byte `random` field, and similarly the server's `ServerHello` message. In practice, these are usually a 4-byte UNIX timestamp concatenated with a 28-byte nonce. The `Certificate` message contains the DSA certificate we generated for the stunnel service. The `ServerKeyExchange` message contains a number of critical fields for our attack: Diffie-Hellman key exchange parameters, the signature algorithm and hash function identifiers, and finally the digital signature itself in the `signed_params` field. Given our stunnel configuration and certificate, the 2048-bit DSA signature is over the concatenated string

```
ClientHello.random + ServerHello.random +
ServerKeyExchange.params
```

and the hash function is SHA-512, both dictated by the `SignatureAndHashAlgorithm` field (explicit values `0x6, 0x2`). Our client saves the hash of this string and the DER-encoded digital signature sent from the server. All subsequent messages, including `ServerHelloDone` and any client responses, are not required by our attack. Our client therefore drops the connection at this stage, and repeats this process several hundred times to build up a set of distinct trace, digital signature, and digest tuples. See Section 6 for our explicit attack parameters. Figure 4 is a typical signal extracted by our spy program in parallel to the handshake between our client and the victim stunnel service.

5.2 Attacking SSH

OpenSSH¹¹ is a suite of tools whose main goal is to provide secure communications over an insecure channel using the SSH network protocol.

OpenSSH is linked to the OpenSSL shared library to perform several cryptographic operations, including digital signatures (excluding `ed25519` signatures). For our experiments we used the stock OpenSSH 6.6.1p1 binary package from the Ubuntu repository, and pointed the run-time shared library loader at OpenSSL 1.0.2h. The DSA key pair used by the server and targeted by our attack is the default 1024-bit key pair generated during installation of OpenSSH.

Similar to Section 5.1, we wrote a custom SSH client that

¹⁰<https://www.stunnel.org>

¹¹<http://www.openssh.com>

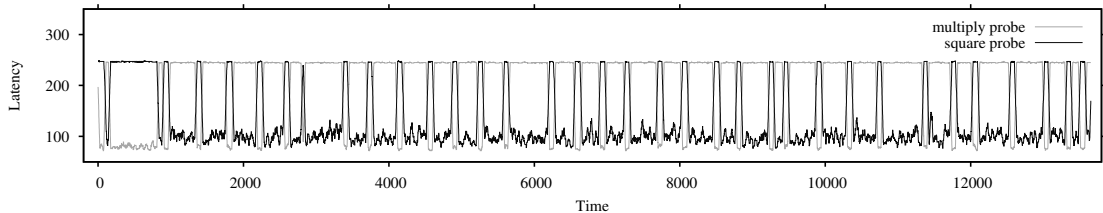


Figure 3: Complete filtered trace of a 1024-bit DSA sign operation during an OpenSSH SSH-2 handshake.

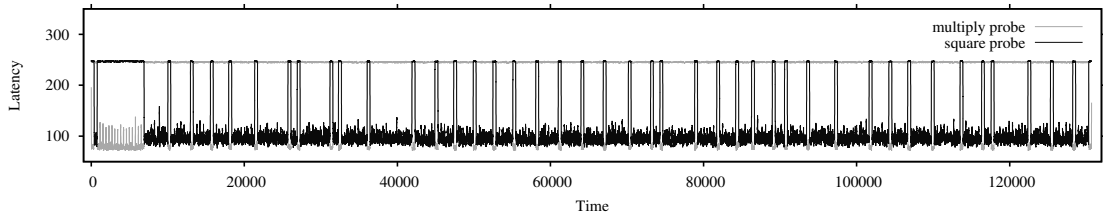


Figure 4: Complete filtered trace of a 2048-bit DSA sign operation during a stunnel TLS 1.2 handshake.

launches our spy program, the spy program collects the timing signals during the handshake. At the same time it performs an SSH handshake where the protocol messages and the digital signature are collected for our attack.

Relevant to this work, the SSH protocol defines the Diffie-Hellman key exchange parameters in the `SSH_MSG_KEXINIT` message, along with the signature algorithm and the hash function identifiers. Additionally a 16-byte `random` nonce is sent for host authentication by the client and the server.

The `SSH_MSG_KEXDH_REPLY` message contains the server’s public key (used to create and verify the signature), server’s DH public key f (used to compute the shared secret K in combination with the client’s DH public key e) and the signature itself. Figure 6 shows the SSH handshake with the critical parameters sent in every message relevant for the attack. To be more precise, the signature is over the SHA-1 hash of the concatenated string

```
ClientVersion + ServerVersion +
Client.SSH_MSG_KEXINIT + Server.SSH_MSG_KEXINIT +
Server.publicKey + minSize + prefSize + maxSize +
p + g + e + f + K
```

As the key exchange¹² and public key parameters, our SSH client was configured to use `diffie-hellman-group-exchange-sha1` and `ssh-dss` respectively. Note that two different hashing functions may be used, one hash function for key derivation following Diffie-Hellman key exchange and another hash function for the signing algorithm, which for DSA is the SHA-1 hash function.

Similarly to the TLS case, our client saves the hash of the concatenated string and the digital signature raw bytes sent from the server. All subsequent messages, including `SSH_MSG_NEWKEYS` and any client responses, are not required by our attack. Our client therefore drops the connection at this stage, and repeats this process several hundred times to

¹²<https://tools.ietf.org/html/rfc4419>

build up a set of distinct trace, digital signature, and digest tuples. See Section 6 for our explicit attack parameters. Figure 3 is a typical signal extracted by our spy program in parallel to the handshake between our client and the victim SSH server.

5.3 Observations

These two widely deployed protocols share many similarities in their handshakes regarding e.g. signaling, content of messages, and security context of messages. However, in the process of designing and implementing our attacker clients we observe a subtle difference in the threat model between the two. In TLS, all values that go into the hash function to compute the digital signature are public and can be observed (unencrypted) in various handshake messages. In SSH, *most* of the values are public—the exception is the last input to the hash function: the shared DH key. The consequence is side-channel attacks against TLS can be passive, listening to legitimate handshakes not initiated by the attacker yet collecting side-channel data as this occurs. In SSH, the attacker must be active and initiate its own handshakes—without knowing the shared DH key, a passive attacker cannot compute the corresponding digest needed later for the lattice stage of the attack. We find this innate protocol level side-channel property to be an intriguing feature, and a factor that should be carefully considered during protocol design.

6. RECOVERING THE PRIVATE KEY

In previous sections we showed how our attack can recover the sequence of square and multiply operations that the victim performs. We further showed how to get the signature information matching each sequence for both SSH and TLS. We now turn to recovering the private key from the information we collect.

The scheme we use is similar to past works. We first

6.2 Lattice attack implementation

Recall that to protect against timing attacks OpenSSL uses an exponent k equivalent to the randomly selected nonce k . \bar{k} is calculated by adding the modulus q once or twice to k to ensure that \bar{k} is of a fixed length. That is, $\bar{k} = k + \gamma q$ such that $2^n \leq \bar{k} < 2^{n+q}$ where $n = \lceil \lg(q) \rceil$ and $\gamma \in \{1, 2\}$.

The side-channel leaks information on bits of the exponent \bar{k} rather than directly on the nonce. To create HNP instances from the leak we need to handle the unknown value of γ . In previous works, due to ECC parameters the modulus is close to a power of two hence the value of γ is virtually constant [9]. For DSA, the modulus is not close to a power of two and the value of γ varies between signatures. The challenge is, therefore, to construct an HNP instance without the knowledge of γ . We now show how to address this challenge.

Recall that $s = k^{-1}(h(m) + \alpha r) \bmod q$. Equivalently, $k = s^{-1}(h(m) + \alpha r) \bmod q$. The side-channel information recovers the ℓ LSBs of k . We, therefore, have $\bar{k} = b2^\ell + a$ where $a = \bar{k} \bmod 2^\ell$ is known, and

$$2^{n-\ell} \leq b < 2^{n-\ell} + \left\lceil \frac{q}{2^\ell} \right\rceil. \quad (1)$$

Following previous works we use $\lfloor \cdot \rfloor_q$ to denote the reduction modulo q to the range $[0, q)$ and $\lfloor \cdot \rfloor_q$ for the reduction modulo q to the range $(-q/2, q/2)$. Within these expressions division operations are carried over the reals whereas all other operations are carried over $GF(q)$.

We now look at $\lfloor b - 2^{n-\ell} \rfloor_q$.

$$\begin{aligned} & \lfloor b - 2^{n-\ell} \rfloor_q \\ &= \lfloor (\bar{k} - a) \cdot 2^{-\ell} - 2^{n-\ell} \rfloor_q \\ &= \lfloor \bar{k} \cdot 2^{-\ell} - a \cdot 2^{-\ell} - 2^{n-\ell} \rfloor_q \\ &= \lfloor k \cdot 2^{-\ell} + \gamma \cdot q \cdot 2^{-\ell} - a \cdot 2^{-\ell} - 2^{n-\ell} \rfloor_q \\ &= \lfloor k \cdot 2^{-\ell} - a \cdot 2^{-\ell} - 2^{n-\ell} \rfloor_q \\ &= \lfloor (s^{-1} \cdot (h(m) + \alpha \cdot r)) \cdot 2^{-\ell} - a \cdot 2^{-\ell} - 2^{n-\ell} \rfloor_q \\ &= \lfloor \alpha \cdot s^{-1} \cdot r \cdot 2^{-\ell} - (2^n + a - s^{-1} \cdot h(m)) \cdot 2^{-\ell} \rfloor_q \end{aligned}$$

Hence, we can set:

$$\begin{aligned} t &= \lfloor s^{-1} \cdot r \cdot 2^{-\ell} \rfloor_q \\ u &= \left\lfloor (2^n + a - s^{-1} \cdot h(m)) \cdot 2^{-\ell} + \left\lceil \frac{q}{2^{\ell+1}} \right\rceil \right\rfloor_q \\ v &= |\alpha \cdot t - u|_q \end{aligned}$$

and by (1) we have $|v| \leq \lceil q/2^{\ell+1} \rceil$.

Out of the HNP instances we generate, we select at random 49 for the SSH attack, 130 for the TLS attack and construct a lattice as described in Section 2.5.2. We solve the CVP problem with a Sage script, performing lattice reduction using BKZ [34], and enumerate the lattice points using Babai's Nearest Plane (NP) algorithm [7]. We apply two different techniques to extend NP to a larger search space. First, we take multiple rounding values to explore 2^{10} different solutions in the tree paths [23, Sec. 4]. Second, we use a randomization technique [26, Sec. 3.5] and shuffle the rows of B between lattice reductions. We repeat with a different random selection of instances until we find the private key.

Table 2: Empirical lattice attack results over a thousand trials. Set size and errors are mean values. Iterations and CPU time are median values.

Victim	OpenSSH (SSH)	stunnel (TLS)
Key size	1024/160-bit	2048/256-bit
Handshakes	260	580
Lattice size	50	131
Set size	70.8	158.1
Errors	2.1	1.7
Iterations	13	22
CPU minutes	5.9	38.8
Success rate (%)	100.0	100.0

6.3 Results

We implemented the attack and evaluated it against the two protocols, SSH with 1024/160-bit DSA and TLS with 2048/256-bit DSA. Table 2 contains the results. For both protocols we only utilize traces with $\ell \geq 3$. With this value we experimentally found that we require 49 such signatures for SSH and 130 for TLS in order to achieve a reasonable probability of solving the resulting CVP.

Because the nonces are chosen uniformly at random, only about one in every four signatures has an ℓ that we can utilize. To gather enough signatures and to compensate for possible trace errors, we collect 580 SM sequences from TLS handshakes and 260 from SSH.

On average, these collected sequences yield 70.8 (SSH) and 158.1 (TLS) traces with $\ell \geq 3$. Comparing the traces to the ground truth, we know that on average less than 3 have trace errors. However, because an adversary cannot check against the ground truth, we leave these erroneous traces in the set and use them in the attack. We note that due to the smaller key size in SSH, trace errors are much more prevalent there.

We construct a lattice from a random selection of the collected traces and attempt to solve the resulting CVP. Due to the presence of the error traces there is a non-negligible probability that our selected set contains an error. Furthermore, even if all the chosen traces are correct, the algorithm may fail to find the target solution due to the heuristic nature of lattice techniques. In case of failure, we repeat the process with a new random selection from the same set. We need to execute a median value of 13 iterations for SSH and 22 for TLS until we find the target solution.

As seen from Table 2, repeating our experiment over a thousand trials on a cluster with hundreds of nodes, mixed between Intel X5660 and AMD Opteron 2435 cores, we find the private key in all cases requiring a median 5.9 CPU minutes for the SSH key and 38.8 CPU minutes for the TLS key. Although we executed each trial on a single core, in reality the iterations are independent of each other—the lattice attack is embarrassingly parallel.

7. CONCLUSION

In this work we disclose a programming error in OpenSSL that results in a security weakness. We show that as a result of the defect, the DSA implementation in OpenSSL is vulnerable to cache-timing attacks, and exploit the vulnerability to mount end-to-end attacks against SSH (via OpenSSH) and TLS (via stunnel).

It is all too easy to dismiss the bug as an innocent pro-

gramming error. However, we believe that the core issue is a design problem. When designing the “constant-time” fix, the developers elected to use an insecure default behavior. From an engineering perspective the decision is justified—it is much easier to identify the handful of locations where we know that the exponent should be kept secret than to analyze the entire library identifying exponents that can be leaked. However, from a security perspective, this design decision breaches the principle of *fail-safe defaults*, which Saltzer and Schroeder [33] justify by saying: *a design or implementation mistake in a mechanism that explicitly excludes access tends to fail by allowing access, a failure which may go unnoticed in normal use*.

It is hard not to appreciate the extraordinary prescience of Saltzer and Schroeder’s justification. Had OpenSSL elected to use a better design, that defaults to the constant-time behavior, a similar bug could have resulted in a small performance loss for non-sensitive exponentiations, but the omission to preserve the flag in question would have been unlikely to jeopardize the security of the system. A more secure design would also improve the security of third-party products in the case that developers may not be aware of the intricacies of the constant-time flags.

The simplest software-based solution to mitigate our attack is to fix the software defect. During responsible disclosure, OpenSSL, LibreSSL, and BoringSSL merged patches for CVE-2016-2178¹³, assigned as a result of this work.

Broader, the `clflush` instruction does not require elevated privileges to execute, hence we suggest access control mechanisms. We recommend that cache flush instructions be privileged-only execution, or at least restricted to memory pages to which the process has write access and to memory pages explicitly allowed by the kernel. Partially or fully disabling caching during sensitive code execution can prevent cache-timing attacks at the cost of performance [3]. Preventing page sharing between processes is a partial solution at the cost of increased memory requirements and avoiding sharing of sensitive code is possible by changing the program loader.

We close with some practical advice regarding this vulnerability. OpenSSH supports building without OpenSSL as a dependency. We recommend that OpenSSH package maintainers switch to this option. For OpenSSH administrators and users, we recommend migrating to `ssh-ed25519` key types, the implementation of which has many desirable side-channel properties. Furthermore, ensure that `ssh-dss` is absent from the `HostKeyAlgorithms` configuration field, and any such `HostKey` entries removed. On the TLS side, we recommend disabling cipher suites that have DSA functionality as a pre-requisite.

Acknowledgments

The first author is supported by the Erasmus Mundus Nord-SecMob Master’s Programme and the European Commission.

The first and second authors are supported in part by TEKES grant 3772/31/2014 Cyber Trust.

This article is based in part upon work from COST Action IC1403 CRYPTACUS, supported by COST (European Cooperation in Science and Technology).

¹³<https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-2178>

We thank Tampere Center for Scientific Computing (TCSC) for generously granting us access to computing cluster resources.

References

- [1] Onur Aciğmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *2007 CT-RSA*, pages 225–242, 2007.
- [2] Onur Aciğmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *2nd AsiaCCS*, Singapore, 2007.
- [3] Onur Aciğmez, Billy Bob Brumley, and Philipp Grabher. New results on instruction cache attacks. In *CHES*, Santa Barbara, CA, US, 2010.
- [4] Thomas Allan, Billy Bob Brumley, Katrina Falkner, Joop van de Pol, and Yuval Yarom. Amplifying side channels through performance degradation. IACR Cryptology ePrint Archive, Report 2015/1141, Nov 2015.
- [5] Diego F. Aranha, Pierre-Alain Fouque, Benoît Gérard, Jean-Gabriel Kammerer, Mehdi Tibouchi, and Jean-Christophe Zapolowicz. GLV/GLS decomposition, power analysis, and attacks on ECDSA signatures with single-bit nonce bias. In *ASIACRYPT*, pages 262–281, Kaohsiung, TW, Dec 2014.
- [6] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using KSM. In *Linux symposium*, pages 19–28, 2009.
- [7] László Babai. On Lovász’ lattice reduction and the nearest lattice point problem. *Combinatorica*, 6(1): 1–13, March 1986.
- [8] Elaine Barker and Allen Roginsky. Transitions: Recommendation for transitioning the use of cryptographic algorithms and key lengths. NIST Special Publication 800-131A Revision 1, Nov 2015. URL <http://dx.doi.org/10.6028/NIST.SP.800-131Ar1>.
- [9] Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. “Ooh aah... just a little bit”: A small amount of side channel can go a long way. In *CHES*, pages 75–92, Busan, KR, Sep 2014.
- [10] Daniel J Bernstein. Cache-timing attacks on AES, 2005. Preprint available at <http://cr.yp.to/papers.html#cachetiming>.
- [11] Daniel Bleichenbacher. On the generation of one-time keys in DL signature schemes. Presentation at IEEE P1363 Working Group meeting, Nov 2000.
- [12] Dan Boneh and Ramarathnam Venkatesan. Hardness of computing the most significant bits of secret keys in Diffie-Hellman and related schemes. In *CRYPTO’96*, pages 129–142, Santa Barbara, CA, US, Aug 1996.
- [13] Ernie Brickell, Gary Graunke, and Jean-Pierre Seifert. Mitigating cache/timing based side-channels in AES and RSA software implementations. RSA Conference 2006 session DEV-203, Feb 2006.

- [14] Billy Bob Brumley and Risto M. Hakala. Cache-timing template attacks. In *15th ASIACRYPT*, pages 667–684, Tokyo, JP, Dec 2009.
- [15] Billy Bob Brumley and Nicola Tuveri. Remote timing attacks are still practical. In *16th ESORICS*, Leuven, BE, 2011.
- [16] Teodoro Ciproso and Mark Stamp. Software reverse engineering. In *Handbook of Information and Communication Security*, pages 659–696. 2010.
- [17] Intel Corporation. Intel 64 and ia-32 architectures optimization reference manual, Jan 2016.
- [18] Elke De Mulder, Michael Hutter, Mark E. Marson, and Peter Pearson. Using Bleichenbacher’s solution to the hidden number problem to attack nonce leaks in 384-bit ECDSA. In *CHES*, pages 435–452, Santa Barbara, CA, US, Aug 2013.
- [19] Daniel Genkin, Lev Pachmanov, Itamar Pipman, Eran Tromer, and Yuval Yarom. ECDSA key extraction from mobile devices via nonintrusive physical side channels. IACR Cryptology ePrint Archive, Report 2016/230, Mar 2016.
- [20] D. Gullasch, E. Bangerter, and S. Krenn. Cache games – bringing access-based cache attacks on AES to practice. In *S&P*, pages 490–505, May 2011.
- [21] Nick Howgrave-Graham and Nigel P. Smart. Lattice attacks on digital signature schemes. *DCC*, 23(3): 283–290, Aug 2001.
- [22] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S&A: A shared cache attack that works across cores and defies VM sandboxing – and its application to AES. In *S&P*, San Jose, CA, US, May 2015.
- [23] Richard Lindner and Chris Peikert. Better key sizes (and attacks) for LWE-based encryption. In *2011 CT-RSA*, pages 319–339, 2011.
- [24] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. ARMageddon: Last-level cache attacks on mobile devices. *arXiv preprint arXiv:1511.04897*, 2015.
- [25] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *S&P*, pages 605–622, May 2015.
- [26] Mingjie Liu and Phong Q Nguyen. Solving BDD by enumeration: An update. In *Topics in Cryptology–CT-RSA 2013*, pages 293–309. 2013.
- [27] Phong Q. Nguyen and Igor E. Shparlinski. The insecurity of the digital signature algorithm with partially known nonces. *J. Cryptology*, 15(2):151–176, Jun 2002.
- [28] Phong Q. Nguyen and Igor E. Shparlinski. The insecurity of the elliptic curve digital signature algorithm with partially known nonces. *DCC*, 30(2): 201–217, Sep 2003.
- [29] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *2006 CT-RSA*, 2006.
- [30] Dan Page. Theoretical use of cache memory as a cryptanalytic side-channel. *IACR Cryptology ePrint Archive*, 2002:169, 2002.
- [31] Colin Percival. Cache missing for fun and profit. In *BSDCan 2005*, Ottawa, CA, 2005.
- [32] Joop van de Pol, Nigel P. Smart, and Yuval Yarom. Just a little bit more. In *2015 CT-RSA*, pages 3–21, San Francisco, CA, USA, Apr 2015.
- [33] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proc. IEEE*, 63(9):1278–1308, Sep 1975.
- [34] C. P. Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Math. Prog.*, 66(1–3):181–199, Aug 1994.
- [35] Augustus K. Uht, Vijay Sindagi, and Kelley Hall. Disjoint eager execution: An optimal form of speculative execution. *MICRO* 28, pages 313–325, 1995.
- [36] Carl A. Waldspurger. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.*, pages 181–194, Dec 2002.
- [37] Colin D. Walter. Longer keys may facilitate side channel attacks. In *SAC*, pages 42–57, Waterloo, ON, Canada, Aug 2004.
- [38] Pieter Wuille. Dealing with malleability. <https://github.com/bitcoin/bips/blob/master/bip-0062.mediawiki>, March 2014.
- [39] Yuval Yarom and Naomi Benger. Recovering OpenSSL ECDSA nonces using the FLUSH+RELOAD cache side-channel attack. IACR Cryptology ePrint Archive, Report 2014/140, Feb 2014.
- [40] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security*, pages 719–732, San Diego, CA, US, 2014.
- [41] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: A timing attack on OpenSSL constant time RSA. In *CHES*, 2016.

PUBLICATION

II

Constant-Time Callees with Variable-Time Callers

C. Pereida García and B. B. Brumley

26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017. Ed. by E. Kirda and T. Ristenpart. 2017, 83–98

Publication reprinted with the permission of the copyright holders



Constant-Time Callees with Variable-Time Callers

Cesar Pereida García and Billy Bob Brumley, *Tampere University of Technology*

<https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/garcia>

This paper is included in the Proceedings of the
26th USENIX Security Symposium
August 16–18, 2017 • Vancouver, BC, Canada

ISBN 978-1-931971-40-9

Open access to the Proceedings of the
26th USENIX Security Symposium
is sponsored by USENIX

Constant-Time Callees with Variable-Time Callers

Cesar Pereida García Billy Bob Brumley
Tampere University of Technology
{cesar.pereidagarcia,billy.brumley}@tut.fi

Abstract

Side-channel attacks are a serious threat to security-critical software. To mitigate remote timing and cache-timing attacks, many ubiquitous cryptography software libraries feature constant-time implementations of cryptographic primitives. In this work, we disclose a vulnerability in OpenSSL 1.0.1u that recovers ECDSA private keys for the standardized elliptic curve P-256 despite the library featuring both constant-time curve operations and modular inversion with microarchitecture attack mitigations. Exploiting this defect, we target the errant modular inversion code path with a cache-timing and improved performance degradation attack, recovering the inversion state sequence. We propose a new approach of extracting a variable number of nonce bits from these sequences, and improve upon the best theoretical result to recover private keys in a lattice attack with as few as 50 signatures and corresponding traces. As far as we are aware, this is the first timing attack against OpenSSL ECDSA that does not target scalar multiplication, the first side-channel attack on cryptosystems leveraging P-256 constant-time scalar multiplication and furthermore, we extend our attack to TLS and SSH protocols, both linked to OpenSSL for P-256 ECDSA signing.

Keywords: applied cryptography; elliptic curve cryptography; digital signatures; side-channel analysis; timing attacks; cache-timing attacks; performance degradation; ECDSA; modular inversion; binary extended Euclidean algorithm; lattice attacks; constant-time software; OpenSSL; NIST P-256; CVE-2016-7056

1 Introduction

Being a widely-deployed open-source cryptographic library, OpenSSL is a popular target for different cryptanalytic attacks, including side-channel attacks that target cryptosystem implementation weaknesses that can leak

critical algorithm state. As a software library, OpenSSL provides not only TLS functionality but also cryptographic functionality for applications such as SSH, IPsec, and VPNs.

Due to its ubiquitous usage, OpenSSL contains arguably one of the most popular software implementations of the Elliptic Curve Digital Signature Algorithm (ECDSA). OpenSSL's scalar multiplication algorithm was shown vulnerable to cache-timing attacks in 2009 [6], and attacks continue on the same code path to this date [2, 4, 10, 27]. Recognizing and responding to the threat cache-timing attacks pose to cryptosystem implementations, OpenSSL mainlined constant-time scalar multiplication for several popular standardized curves already in 2011 [16].

In this work, we disclose a software defect in the OpenSSL (1.0.1 branch) ECDSA implementation that allows us to design and implement a side-channel cache-timing attack to recover private keys. Different from previous work, our attack focuses on the modular inversion operation instead of the typical scalar multiplication, thus allowing us to target the standardized elliptic curve P-256, circumventing its constant-time scalar multiplication implementation. The root cause of the defect is failure to set a flag in ECDSA signing nonces that indicates only constant-time code paths should be followed.

We leverage the state-of-the-art FLUSH+RELOAD [28] technique to perform our cache-timing attack. We adapt the technique to OpenSSL's implementation of ECDSA and the *Binary Extended Euclidean Algorithm* (BEEA). Our spy program probes relevant memory addresses to create a timing signal trace, then the signal is processed and converted into a sequence of right-shift and subtraction (LS) operations corresponding to the BEEA execution state from which we extract bits of information to create a lattice problem. The solution to the lattice problem yields the ECDSA secret key.

We discover that observing as few as 5 operations

from the LS sequence allows us to use every single captured trace for our attack. This significantly reduces both the required amount of signatures and side-channel data compared to previous work [8], and maintains a good signature to lattice dimension ratio.

We build upon the performance degradation technique of Allan et al. [2] to efficiently find the memory addresses with the highest impact to the cache during the degrading attack. This new approach allows us to accurately find the best candidate memory addresses to slow the modular inversion by an average factor of 18, giving a high resolution trace and allowing us to extract the needed bits of information from all of the traces.

Unlike previous works targeting the *wNAF* scalar multiplication code path (for curves such as Bitcoin’s *secp256k1*) or performing theoretical side-channel analysis of the BEEA, we are the first to demonstrate a practical cache-timing attack against the BEEA modular inversion, and furthermore OpenSSL’s ECDSA signing implementation with constant-time P-256 scalar multiplication.

Our contributions in this work include the following:

- We identify a bug in OpenSSL that allows a cache-timing attack on ECDSA signatures, despite constant-time P-256 scalar multiplication. (Section 3)
- We describe a new quantitative approach that accurately identifies the most accessed victim memory addresses w.r.t. data caching, then we use them for an improved performance degradation attack in combination with the FLUSH+RELOAD technique. (Section 4.3)
- We describe how to combine the FLUSH+RELOAD technique with the improved performance degradation attack to recover side-channel traces and algorithm state from the BEEA execution. (Section 4)
- We present an alternate approach to recovering nonce bits from the LS sequences, focused on minimizing required side-channel information. Using this approach, we recover bits of information from every trace, allowing us to use every signature query to construct and solve a lattice problem, revealing the secret key with as few as 50 signatures and corresponding traces. (Section 4.4)
- We perform a key-recovery cache-timing attack on the TLS and SSH protocols utilizing OpenSSL for ECDSA functionality. (Section 5)

2 Background

2.1 Elliptic Curve Cryptography

ECC. Developed in the mid 1980’s, elliptic curves were introduced to cryptography by Miller [20] and Koblitz

[17] independently. Elliptic Curve Cryptography (ECC) became popular mainly for two important reasons: no sub-exponential time algorithm to solve the elliptic curve discrete logarithm problem is known for well-chosen parameters and it operates in the group of points on an elliptic curve, compared to the classic multiplicative group of a finite field, thus allowing the use of smaller parameters to achieve the same security levels—consequently smaller keys and signatures.

Although there are more general forms of elliptic curves, for the purposes of this paper we restrict to short Weierstrass curves over prime fields. With prime $p > 3$, all of the $x, y \in GF(p)$ solutions to the equation

$$E : y^2 = x^3 + ax + b$$

along with an identity element form an abelian group. Due to their performance characteristics, the parameters of interest are the NIST standard curves that set $a = -3$ and p a Mersenne-like prime.

ECDSA. Throughout this paper, we use the following notation for the Elliptic Curve Digital Signature Algorithm (ECDSA).

Parameters: A generator $G \in E$ of an elliptic curve group of prime order n and an approved hash function h (e.g. SHA-1, SHA-256, SHA-512).

Private-Public key pairs: The private key α is an integer uniformly chosen from $\{1 \dots n-1\}$ and the corresponding public key $D = [\alpha]G$ where $[j]G$ denotes scalar-by-point multiplication using additive group notation. Calculating the private key given the public key requires solving the elliptic curve discrete logarithm problem and for correctly chosen parameters, this is an intractable problem.

Signing: A given party, Alice, wants to send a signed message m to Bob. Using her private-public key pair (α_A, D_A) , Alice performs the following steps:

1. Select uniformly at random a secret nonce k such that $0 < k < n$.
2. Compute $r = ([k]G)_x \bmod n$.
3. Compute $s = k^{-1}(h(m) + \alpha_A r) \bmod n$.
4. Alice sends (m, r, s) to Bob.

Verifying: Bob wants to be sure the message he received comes from Alice—a valid ECDSA signature gives strong evidence of authenticity. Bob performs the following steps to verify the signature:

1. Reject the signature if it does not satisfy $0 < r < n$ and $0 < s < n$.
2. Compute $w = s^{-1} \bmod n$ and $h(m)$.
3. Compute $u_1 = h(m)w \bmod n$ and $u_2 = rw \bmod n$.
4. Compute $(x, y) = [u_1]G + [u_2]D_A$.
5. Accept the signature if and only if $x = r \bmod n$ holds.

2.2 Side-Channel Attacks

Thanks to the adoption of ECC and the increasing use of digital signatures, ECDSA has become a popular algorithm choice for digital signatures. ECDSA's popularity makes it a good target for side-channel attacks.

At a high level, an established methodology for ECDSA is to query multiple signatures, then partially recover nonces k_i from the side-channel, leading to a bound on the value $\alpha t_i - u_i$ that is shorter than the interval $\{1..n-1\}$ for some known integers t_i and u_i . This leads to a version of the Hidden Number Problem (HNP) [5]: recover α given many (t_i, u_i) pairs. The HNP instances are then reduced to Closest Vector Problem (CVP) instances, solved with lattice methods.

Over the past decade, several authors have described practical side-channel attacks on ECDSA that exploit partial nonce disclosure by different microprocessor features to recover long-term private keys.

Brumley and Hakala [6] describe the first practical side-channel attack against OpenSSL's ECDSA implementation. They use the EVICT+RELOAD strategy and an L1 data cache-timing attack to recover the LSBs of ECDSA nonces from the library's wNAF (a popular low-weight signed-digit representation) scalar multiplication implementation in OpenSSL 0.9.8k. After collecting 2,600 signatures (8K with noise) from the `dgst` command line tool and using the Howgrave-Graham and Smart [15] lattice attack, the authors recover a 160-bit ECDSA private key from standardized curve `secp160r1`.

Brumley and Tuveri [7] attack ECDSA with binary curves in OpenSSL 0.9.8o. Mounting a remote timing attack, the authors show the library's Montgomery Ladder scalar multiplication implementation leaks timing information on the MSBs of the nonce used and after collecting that information over 8,000 TLS handshakes a 162-bit NIST B-163 private key can be recovered with lattice methods.

Benger et al. [4] target OpenSSL's wNAF implementation and 256-bit private keys for the standardized GLV curve [11] `secp256k1` used in the Bitcoin protocol. Using as few as 200 ECDSA signatures and the FLUSH+RELOAD technique [28], the authors find some LSBs of the nonces and extend the lattice technique of [21, 22] to use a varying amount of leaked bits rather than limiting to a fixed number.

van de Pol et al. [27] attack OpenSSL's 1.0.1e wNAF implementation for the curve `secp256k1`. Leveraging the structure of the modulus n , the authors use more information leaked in consecutive sequences of bits anywhere in the top half of the nonces, allowing them to recover the secret key after observing as few as 25 ECDSA signatures.

Allan et al. [2] improve on previous results by using

a performance-degradation attack to amplify the side-channel. This amplification allows them to additionally observe the sign bit of digits in the wNAF representation used in OpenSSL 1.0.2a and to recover `secp256k1` private keys after observing only 6 signatures.

Fan et al. [10] increase the information extracted from each signature by analyzing the wNAF implementation in OpenSSL. Using the curve `secp256k1` as a target, they perform a successful attack after observing as few as 4 signatures.

Our work differs from previous ECDSA side-channel attacks in two important ways. (1) We focus on NIST standard curve P-256, featured in ubiquitous security standards such as TLS and SSH. Later in Section 2.5, we explain the reason previous works were unable to target this extremely relevant curve. (2) We do not target the scalar-by-point multiplication operation (i.e. the bottleneck of the signing algorithm), but instead Step 3 of the signing algorithm, the modular inversion operation.

2.3 The FLUSH+RELOAD Attack

The FLUSH+RELOAD technique is a cache-based side-channel attack technique targeting the Last-Level Cache (LLC) and used during our attack. FLUSH+RELOAD is a high resolution, high accuracy and high signal-to-noise ratio technique that positively identifies accesses to specific memory lines. It relies on cache sharing between processes, typically achieved through the use of shared libraries or page de-duplication.

Input: Memory Address *addr*.

Result: True if the victim accessed the address.

begin

```
flush(addr)
Wait for the victim.
time ← current_time()
tmp ← read(addr)
readTime ← current_time() - time
return readTime < threshold
```

Figure 1: FLUSH+RELOAD Attack

A round of attack, depicted in Figure 1, consists of three phases: (1) The attacker evicts the target memory line from the cache. (2) The attacker waits some time so the victim has an opportunity to access the memory line. (3) The attacker measures the time it takes to reload the memory line. The latency measured in the last step tells whether or not the memory line was accessed by the victim during the second step of the attack, i.e. identifies cache-hits and cache-misses.

The FLUSH+RELOAD attack technique tries to achieve the best resolution possible while keeping the

error rate low. Typically, an attacker encounters multiple challenges due to several processor optimizations and different architectures. See [2, 24, 28] for discussions of these challenges.

2.4 Binary Extended Euclidean Algorithm

The modular inversion operation is one of the most basic and essential operations required in public key cryptography. Its correct implementation and constant-time execution has been a recurrent topic of research [1, 3, 8].

A well known algorithm used for modular inversion is the Euclidean Extended Algorithm and in practice is often substituted by a variant called the *Binary Extended Euclidean Algorithm (BEEA)* [18, Chap. 14.4.3]. This variant replaces costly division operations by simple right-shift operations, thus, achieving performance benefits over the regular version of the algorithm. BEEA is particularly efficient for very long integers—e.g. RSA, DSA, and ECDSA operands.

Input: Integers k and p such that $\gcd(k, p) = 1$.

Output: $k^{-1} \bmod p$.

$v \leftarrow p, u \leftarrow k, X \leftarrow 1, Y \leftarrow 0$

while $u \neq 0$ **do**

while $\text{even}(u)$ **do**

$u \leftarrow u/2$ /* u loop */

if $\text{odd}(X)$ **then** $X \leftarrow X + p$

$X \leftarrow X/2$

while $\text{even}(v)$ **do**

$v \leftarrow v/2$ /* v loop */

if $\text{odd}(Y)$ **then** $Y \leftarrow Y + p$

$Y \leftarrow Y/2$

if $u \geq v$ **then**

$u \leftarrow u - v$

$X \leftarrow X - Y$

else

$v \leftarrow v - u$

$Y \leftarrow Y - X$

return $Y \bmod p$

Figure 2: Binary Extended Euclidean Algorithm.

Figure 2 shows the BEEA. Note that in each iteration only one u or v while-loop is executed, but not both. Additionally, in the very first iteration only the u while-loop can be executed since v is a copy of p which is a large prime integer n for ECDSA.

In 2007, independent research done by Aciiçmez et al. [1], Aravamuthan and Thumparthy [3] demonstrated side-channel attacks against the BEEA. Aravamuthan and Thumparthy [3] attacked BEEA using Power Analysis attacks, whereas Aciiçmez et al. [1] attacked BEEA

through Simple Branch Prediction Analysis (SBPA), demonstrating the fragility of this algorithm against side-channel attacks.

Both previous works reach the conclusion that in order to reveal the value of the nonce k , it is necessary to identify four critical input-dependent branches leaking information, namely:

1. Number of right-shift operations performed on v .
2. Number of right-shift operations performed on u .
3. Number and order of subtractions $u := u - v$.
4. Number and order of subtractions $v := v - u$.

Moreover, both works present a BEEA reconstruction algorithm that allows them to fully recover the nonce k —and therefore the secret signing key—given a perfect side-channel trace that distinguish the four critical branches.

Aravamuthan and Thumparthy [3] argue that a countermeasure to secure BEEA against side-channel attacks is to render u and v subtraction branches indistinguishable, thus the attack is computationally expensive to carry out. As a response, Cabrera Aldaya et al. [8] demonstrated a Simple Power Analysis (SPA) attack against a custom implementation of the BEEA. The authors' main contribution consists of demonstrating it is possible to partially determine the order of subtractions on branches u and v only by knowing the number of right-shift operations performed in every while-loop iteration. Under a perfect SPA trace, the authors use an algebraic algorithm to determine a short execution sequence of u and v subtraction branches.

They manage to recover various bits of information for several ECDSA key sizes. The authors are able to recover information only from some but not all of their SPA traces by using their algorithm and the partial information about right-shift and subtraction operations. Finally, using a lattice attack they recover the secret signing key.

As can be seen from the previous works, depending on the identifiable branches in the trace and quality of the trace it is possible to recover full or partial information about the nonce k . Unfortunately, the information leaked by most of the real world side-channels does not allow us to differentiate between subtraction branches u and v , therefore limiting the leaked information to three input-dependent branches:

1. Number of right-shift operations performed on v .
2. Number of right-shift operations performed on u .
3. Number of subtractions.

2.5 OpenSSL History

OpenSSL has a rich and storied history as a prime security attack target [19], a distinction ascribed to the li-

brary’s ubiquitous real world application. One of the main contributions of our work is identifying a new OpenSSL vulnerability described later in Section 3. To understand the nature of this vulnerability and facilitate root cause analysis, in this section we give a brief overview of side-channel defenses in the OpenSSL library, along with some context and insight into what prompted these code changes. Table 1 summarizes the discussion.

0.9.7. Side-channel considerations started to induce code changes in OpenSSL starting with the 0.9.7 branch. The RSA cache-timing attack by Percival [23] recovered secret exponent bits used as lookup table indices in sliding window exponentiation using an EVICT+RELOAD strategy on HyperThreading architectures. His work prompted introduction of the `BN_FLG_CONSTTIME` flag, with the intention of allowing special security treatment of `BIGNUMs` having said flag set. At the time—and arguably still—the most important use case of the flag is modular exponentiation. Introduced alongside the flag, the `BN_mod_exp_mont_consttime` function is a fixed-window modular exponentiation algorithm featuring data cache-timing countermeasures. Recent research brings the security of this solution into question [29].

0.9.8. The work by Aciğmez et al. [1] targeting BEEA prompted the introduction of the `BN_mod_inverse_no_branch` function, an implementation with more favorable side-channel properties than that of BEEA. The implementation computes modular inversions in a way that resembles the classical extended Euclidean algorithm, calculating quotients and remainders in each step by calling `BN_div` updated to respect the `BN_FLG_CONSTTIME` flag. Tracking callers to `BN_mod_inverse`, the commit¹ enables the `BN_FLG_CONSTTIME` across several cryptosystems where the modular inversion inputs were deemed security critical, notably the published attack targeting RSA.

1.0.1. Based on the work by Käsper [16], the 1.0.1 branch introduced constant-time scalar multiplication implementations for several popular elliptic curves. This code change was arguably motivated by the data cache-timing attack of Brumley and Hakala [6] against OpenSSL that recovered digits of many ECDSA nonces during scalar multiplication on HyperThreading architectures using the EVICT+RELOAD strategy. This information was then used to construct a lattice problem and calculate ECDSA private keys. The commit² included several new `EC_METHOD` implementations, of which arguably `EC_GFp_nistp256_method` has the most real world application to date. This new scalar multiplication imple-

Table 1: OpenSSL side-channel defenses across versions. Although `BN_mod_exp_mont_consttime` was introduced in the 0.9.7 branch, here we are referring to its use for modular inversion via FLT.

OpenSSL version	0.9.6	0.9.7	0.9.8	1.0.0	1.0.1	1.0.2
<code>BN_mod_inverse</code>	✓	✓	✓	✓	✓	✓
<code>BN_FLG_CONSTTIME</code>	—	✓	✓	✓	✓	✓
<code>BN_mod_inverse_no_branch</code>	—	—	✓	✓	✓	✓
<code>ec_nistp_64_gcc_128</code>	—	—	—	—	—	✓
<code>BN_mod_exp_mont_consttime</code>	—	—	—	—	—	✓
<code>EC_GFp_nistz256_method</code>	—	—	—	—	—	✓

mentation uses fixed-window combing combined with secure table lookups via software multiplexing (masking), and is enabled with the `ec_nistp_64_gcc_128` option at build time. For example, Debian 8.0 “Jessie” (current LTS, not EOL) and 7.0 “Wheezy” (previous LTS, not EOL) and Ubuntu 14.04 “Trusty” (previous LTS, not EOL) enable said option when possible for their OpenSSL 1.0.1 package builds. From the side-channel attack perspective, we note that this change is the reason academic research (see Section 2.2) shifted to the `secp256k1` curve—NIST P-256 no longer takes the generic `wNAF` scalar multiplication code path like `secp256k1`.

1.0.2. Motivated by performance and the potential to utilize Intel AVX extensions, a contribution by Gueron and Krasnov [14] included fast and secure curve P-256 operations with their custom `EC_GFp_nistz256_method`. Here we focus on a cherry picked commit³ that affected the ECDSA sign code path for all elliptic curves. While speed motivated the contribution, Möller observes⁴: “It seems that the `BN_MONT_CTX`-related code (used in `crypto/ecdsa` for constant-time signing) is entirely independent of the remainder of the patch, and should be considered separately.” Gueron confirms: “The optimization made for the computation of the modular inverse in the ECDSA sign, is using const-time mod-exp. Indeed, this is independent of the rest of the patch, and it can be used independently (for other usages of the library). We included this addition in the patch for the particular usage in ECDSA.” Hence following this code change, ECDSA signing for all curves now compute modular inversion via `BN_mod_exp_mont_consttime` and Fermat’s Little Theorem (FLT).

3 A New Vulnerability

From Table 1, starting with 1.0.1 the reasonable expectation is that cryptosystems utilizing P-256 resist timing attacks, whether they be remote, data cache, instruction

¹<https://github.com/openssl/openssl/commit/bd31fb21454609b125ade1ad569ebcc2a2b9b73c>

²<https://github.com/openssl/openssl/commit/3e00b4c9db42818c621f609e70569c7d9ae85717>

³<https://github.com/openssl/openssl/commit/8aed2a7548362e88e84a7feb795a3a97e8395008>

⁴<https://rt.openssl.org/Ticket/Display.html?id=3149&user=guest&pass=guest>

cache, or branch predictor timings. We focus here on the combination of ECDSA and P-256 within the library. The reason this is a reasonable expectation is that `ec_nistp_64_gcc_128` provides constant-time scalar multiplication to protect secret scalar nonces, and `BN_mod_inverse_no_branch` provides microarchitecture attack defenses when inverting these nonces. For ECDSA, these are the two most critical locations where the secret nonce is an operand—to produce r and s , respectively.

The vulnerability we now disclose stems from the changes introduced in the 0.9.8 branch. The `BN_mod_inverse` function was modified to first check the `BN_FLG_CONSTTIME` flag of the `BIGNUM` operands—if set, the function then early exits to `BN_mod_inverse_no_branch` to protect the security-sensitive inputs. If the flag is not set, i.e. inputs are not secret, the control flow continues to the stock BEEA implementation.

Paired with this code change, the next task was to identify callers to `BN_mod_inverse` within the library, and enable the `BN_FLG_CONSTTIME` flag for `BIGNUM`s in cryptosystem implementations that are security-sensitive. Our analysis suggests this was done by searching the code base for uses of the `BN_FLG_EXP_CONSTTIME` flag that was replaced with `BN_FLG_CONSTTIME` as part of the changeset, given the evolution of constant-time as concept within OpenSSL and no longer limited to modular exponentiation. As a result, the code changes permeated RSA, DSA, and Diffie-Hellman implementations, but not ECC-based cryptosystems such as ECDH and ECDSA.

This leaves a gap for 1.0.1 with respect to ECDSA. While `ec_nistp_64_gcc_128` provides constant-time scalar multiplication to compute the r component of P-256 ECDSA signatures, the s component will compute modular inverses of security-critical nonces with the stock `BN_mod_inverse` function, not taking the `BN_mod_inverse_no_branch` code path. In the end, the root cause is that the ECDSA signing implementation does not set the `BN_FLG_CONSTTIME` flag for nonces. Scalar multiplication with `ec_nistp_64_gcc_128` is oblivious to this flag and always treats single scalar inputs as security-sensitive, yet `BN_mod_inverse` requires said flag to take the new secure code path.

Figure 3 illustrates this vulnerability running in OpenSSL 1.0.1u. The caller function `ecdsa_sign_setup` contains the bulk of the ECDSA signing cryptosystem—generating a nonce, computing the scalar multiple, inverting the nonce, computing r , and so on. When control flow reaches callee `BN_mod_inverse`, inputs a and n are the nonce and generator order, respectively. Stepping by instruction, it shows that the call to `BN_mod_inverse_no_branch` never takes place, since the `BN_FLG_CONSTTIME` flag is not set for either of these operands. Failing this security critical branch, the control flow continues to

```

+--bn_gcd.c-----
|226  BIGNUM *BN_mod_inverse(BIGNUM *in,
|227  const BIGNUM *a, const BIGNUM *n, BN_CTX *ctx)
|228  {
Bn |229  BIGNUM *A, *B, *X, *Y, *M, *D, *T, *R = NULL;
|230  BIGNUM *ret = NULL;
|231  int sign;
|232
|233  if ((BN_get_flags(a, BN_FLG_CONSTTIME) != 0)
>|234  || (BN_get_flags(n, BN_FLG_CONSTTIME) != 0)) {
|235  return BN_mod_inverse_no_branch(in, a, n, ctx);
|236  }
+-----+
|0x7ffff7da1c7 <BN_mod_inverse+56> mov    -0x90(%rbp),%rax
|0x7ffff7da1ce <BN_mod_inverse+63> mov    0x14(%rax),%eax
|0x7ffff7da1d1 <BN_mod_inverse+66> and    $0x4,%eax
|0x7ffff7da1d4 <BN_mod_inverse+69> test   %eax,%eax
|0x7ffff7da1d6 <BN_mod_inverse+71> jne   0x7ffff7da1e9 <BN_mod_inverse+90>
|0x7ffff7da1d8 <BN_mod_inverse+73> mov    -0x98(%rbp),%rax
|0x7ffff7da1df <BN_mod_inverse+80> mov    0x14(%rax),%eax
|0x7ffff7da1e2 <BN_mod_inverse+83> and    $0x4,%eax
|0x7ffff7da1e5 <BN_mod_inverse+86> test   %eax,%eax
>|0x7ffff7da1e7 <BN_mod_inverse+88> je     0x7ffff7da212 <BN_mod_inverse+131>
+-----+
native process 3399 in: BN_mod_inverse L234 PC: 0x7ffff7da1e7
(gdb) run dgst -sha256 -sign prime256v1.pem -out lab-release.sig /etc/lab-release
Starting program: /usr/local/ssl/bin/openssl dgst -sha256 -sign prime256v1.pem ...
Breakpoint 1, BN_mod_inverse (...) at bn_gcd.c:229
(gdb) backtrace
#0  BN_mod_inverse (...) at bn_gcd.c:229
#1  0x00007ffff782aed9 in ecdsa_sign_setup (...) at ecs_ossl.c:182
#2  0x00007ffff782bc35 in ECDSA_sign_setup (...) at ecs_sign.c:105
#3  0x00007ffff782b29a in ecdsa_do_sign (...) at ecs_ossl.c:269
#4  0x00007ffff782bafd in ECDSA_do_sign_ex (...) at ecs_sign.c:74
#5  0x00007ffff782bb97 in ECDSA_sign_ex (...) at ecs_sign.c:89
#6  0x00007ffff782bb44 in ECDSA_sign (...) at ecs_sign.c:80 ...
(gdb) stepi
(gdb) macro expand BN_get_flags(a, BN_FLG_CONSTTIME)
expands to: ((a)->flags&0x04)
(gdb) print BN_get_flags(a, BN_FLG_CONSTTIME)
$1 = 0
(gdb) print BN_get_flags(n, BN_FLG_CONSTTIME)
$2 = 0

```

Figure 3: Modular inversion within OpenSSL 1.0.1u (built with `ec_nistp_64_gcc_128` enabled) for P-256 ECDSA signing. Operands a and n are the nonce and generator order, respectively. The early exit to `BN_mod_inverse_no_branch` never takes place, since the caller `ecdsa_sign_setup` fails to set the `BN_FLG_CONSTTIME` flag on the operands. Control flow continues to the stock, classical BEEA implementation.

the stock, classical BEEA implementation.

3.1 Forks

OpenSSL is not the only software library affected by this vulnerability. Following HeartBleed, OpenBSD forked OpenSSL to LibreSSL in July 2014, and Google forked OpenSSL to BoringSSL in June 2014. We now discuss this vulnerability within the context of these two forks. **LibreSSL.** An 04 Nov 2016 commit⁵ cherry picked the `EC_GFp_nistz256.method` for LibreSSL. Interestingly, LibreSSL is the library most severely affected by this vulnerability. The reason is they did not cherry pick the `BN_mod_exp_mont_consttime` ECDSA nonce inversion. That is, as of this writing (fixed during disclosure) the current LibreSSL master branch can feature constant-time P-256 scalar multiplication with either `EC_GFp_nistz256.method` or `EC_GFp_nistp256.method` callees depending on compile-time options and minor code changes, but inverts all ECDSA nonces with

⁵<https://github.com/libressl-portable/openbsd/commit/85b48e7c232e1dd18292a78a266c95dd317e23d3>

the `BN_mod_inverse` callee that fails the same security critical branch as OpenSSL, due to the caller `ecdsa_sign_setup` not setting the `BN_FLG_CONSTTIME` flag for ECDSA signing nonces. We confirmed the vulnerability using a LibreSSL build with debug symbols, checking the inversion code path with a debugger.

BoringSSL. An 03 Nov 2015 commit⁶ picked up the `EC_GFp_nistz256_method` implementation for BoringSSL. That commit also included the `BN_mod_exp_mont_consttime` ECDSA nonce inversion callee, which OpenSSL cherry picked. The parent tree⁷ is slightly older on the same day. Said tree features constant-time P-256 scalar multiplication with callee `EC_GFp_nistp256_method`, but inverts ECDSA signing nonces with callee `BN_mod_inverse` that fails the same security critical branch, again due to the `BN_FLG_CONSTTIME` flag not being set by the caller—i.e. it follows essentially the same code path as OpenSSL. We verified the vulnerability affects said tree using a debugger.

4 Exploiting the Vulnerability

Exploiting the vulnerability and performing our cachetiming attack is a long and complex process, therefore the analysis details are decomposed in several subsections. Section 4.1 discusses the hardware and software setup used during our experimentation phase. Section 4.2 analyzes and describes the sources of leakage in OpenSSL and the exploitation techniques. Section 4.3 and Section 4.4 describe in detail our improvements on the performance degradation technique and key recovery, respectively. Figure 4 gives an overview of the attack scenario followed during our experiments.

4.1 Attack Setup

Our attack setup consists of an Intel Core i5-2400 Sandy Bridge 3.10GHz (32 nm) with 8GB of memory running 64-bit Ubuntu 16.04 LTS “Xenial”. Each CPU core has an 8-way 32KB L1 data cache, an 8-way 32KB L1 instruction cache, an 8-way 256KB L2 unified cache, and all the cores share a 12-way 6MB unified LLC (all with 64B cache lines). It does not feature HyperThreading.

We built OpenSSL 1.0.1u with debugging symbols on the executable. Debugging symbols facilitate mapping source code to memory addresses, serving a double purpose to us: (1) Improving our degrading attack (see Section 4.3); (2) Probing the sequence of operations accurately. Note that debugging symbols are not

⁶<https://boringssl.googleusercontent.com/boringssl/+18954938684e269ccd59152027d2244040e2b819%5E%21/>

⁷<https://boringssl.googleusercontent.com/boringssl/+27a0d086f7bbf7076270dbeee5e65552eb2eab3a>

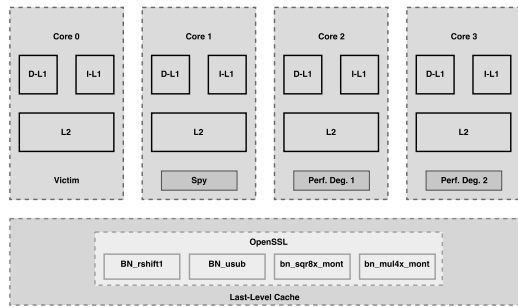


Figure 4: Simplified attack scenario depicting a victim, a spy and two performance degradation processes each running on a different core. OpenSSL is a shared library and all the processes have a shared LLC.

loaded during run time, thus not affecting victim’s performance. Attackers can map source code to memory addresses by using reverse engineering techniques [9] if debugging symbols are not available. We set `enable-ec_nistp_64_gcc_128` and shared as configuration options at build time to ensure faster execution, constant-time scalar multiplication and compile OpenSSL as a shared object.

4.2 Source of Leakage

As seen in the Figure 3 backtrace, when performing an ECDSA digital signature, OpenSSL calls `ecdsa_sign_setup` to prepare the required parameters and compute most of the actual signature. The random nonce k is created and to avoid possible timing attacks [7] an equivalent fixed bit-length nonce is computed. The length of the equivalent nonce \hat{k} is fixed to one bit more than that of the group’s prime order n , thus the equivalent nonce satisfies $\hat{k} = k + \gamma \cdot n$ where $\gamma \in \{1, 2\}$.

Additionally, `ecdsa_sign_setup` computes the signature’s r using a scalar multiplication function pointer wrapper (i.e. for P-256, traversing the constant-time code path instead of generic $wNAF$) followed by the modular inverse k^{-1} , needed for the s component of the signature. To compute the inversion, it calls `BN_mod_inverse`, where the `BN_FLG_CONSTTIME` flag is checked but due to the vulnerability discussed in Section 3 the condition fails, therefore proceeding to compute k^{-1} using the classical BEEA.

Note that before executing the BEEA, the equivalent nonce \hat{k} is unpadded through a modular reduction operation, resulting in the original nonce k and voiding the fixed bit-length countermeasure applied shortly before by `ecdsa_sign_setup`.

The goal of our attack is to accurately trace and re-

cover side-channel information leaked from the BEEA execution, allowing us to construct the sequence of right-shift and subtraction operations. To that end, we identify the routines used in the `BN_mod_inverse` method leaking side-channel information.

The `BN_mod_inverse` method operates with very large integers, therefore it uses several specific routines to perform basic operations with BIGNUMs. Addition operations call the routine `BN_uadd`, which is a wrapper for `bn_add_words`—assembly code performing the actual addition. Two different routines are called to perform right-shift operations. The `BN_rshift1` routine performs a single right-shift by one bit position, used on X and Y in their respective loops. The `BN_rshift` routine receives the number of bit positions to shift right as an argument, used on u and v at the end of their respective loops. OpenSSL keeps a counter for the shift count, and the loop conditions test u and v bit values at this offset. This is an optimization allowing u and v to be right-shifted all at once in a single call instead of iteratively. Additionally, subtraction is achieved through the use of the `BN_usub` routine, which is a pure C implementation.

Similar in spirit to previous works [4, 24, 27] that instead target other functionality within OpenSSL, we use the `FLUSH+RELOAD` technique to attack OpenSSL’s BEEA implementation. As mentioned before in Section 2.4, unfortunately the side-channel and the algorithm implementation do not allow us to efficiently probe and distinguish the four critical input-dependent branches, therefore we are limited to knowing only the execution of addition, right-shift and subtraction operations.

After identifying the input-dependent branches in OpenSSL’s implementation of the BEEA, using the `FLUSH+RELOAD` technique we place probes in code routines `BN_rshift1` and `BN_usub`. These two routines provide the best resolution and combination of probes, allowing us to identify the critical input-dependent branches.

The modular inversion is an extremely fast operation and only a small fraction of the entire digital signature. It is challenging to get good resolution and enough granularity with the `FLUSH+RELOAD` technique due to the speed of the technique itself, therefore, we apply a variation of the performance degradation attack to slow down the modular inversion operation by a factor of ~ 18 . (See Section 4.3.)

Maximizing performance degradation by identifying the best candidate memory lines gives us the granularity required for the attack. Combining the `FLUSH+RELOAD` technique with a performance degradation attack allows us to determine the number of right-shift operations executed between subtraction calls by the BEEA. From the trace, we reconstruct the sequence of right-shift and subtraction operations (*LS sequence*) executed by the BEEA.

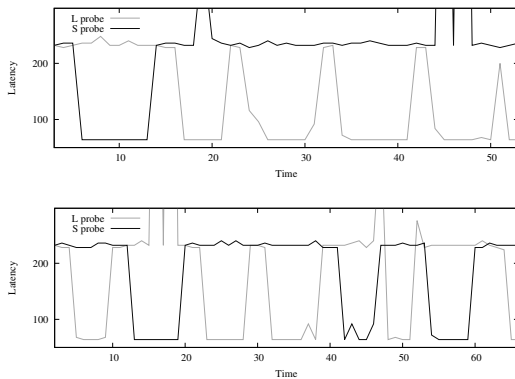


Figure 5: Raw traces for the beginning of two BEEA executions. The L probe tracks right-shift latencies and the S probe tracks subtraction. Latency is in CPU clock cycles. For visualization, focus on the amplitude valleys, i.e. low latency. Top: LS sequence starting SLLLL corresponds to $j = 5$, $\ell_i = 4$, $a_i = 1$. Bottom: LS sequence starting LSLLSLS corresponds to $j = 7$, $\ell_i = 5$, $a_i = 10$. See Section 4.4 for notation.

As Figure 4 illustrates, our attack scenario exploits three CPU cores by running a malicious process in every core and the victim process in the fourth core. The attack consists of a spy process probing the right-shift and subtraction operations running in parallel with the victim application. Additionally, two degrading processes slow down victim’s execution, allowing us to capture the LS sequence almost perfectly. Unfortunately there is not always a reliable indicator in the signal for transitions from one right-shift operation to the next, therefore we estimate the number of adjacent right-shift operations by taking into account the latency and the horizontal distance between subtractions. Figure 5 contains sample raw traces captured in our test environment.

Our spy process accurately captures all the subtraction operations but duplicates some right-shift operations, therefore we focus on the first part of the sequence to recover a variable amount of bits of information from every trace. (See Section 4.4.)

4.3 Improving Performance Degradation

Performance degradation attacks amplify side-channel signals, improving the quality and the amount of information leaked. Our performance degradation attack improves upon the work of Allan et al. [2]. In their work, the authors first need to identify “hot” memory addresses, i.e. memory addresses frequently accessed. They suggest two approaches to find suitable memory lines to degrade. The first approach is to read and under-

stand the victim code in order to identify frequently accessed code sections such as tight loops. This approach requires understanding the code, a task that might not always be possible, takes time and it is prone to errors [26], therefore the authors propose another option.

The second and novel approach they propose is to automate code analysis by collecting code coverage information using the `gcov` tool. The code coverage tool outputs accessed code lines and then using this information it is possible for an attacker to locate the memory lines corresponding to the code lines. Some caveats of this approach are that source lines can be replicated due to compiler optimizations, thus the `gcov` tool might misreport the number of memory accesses. Moreover, code lines containing function calls can be twice as effective compared to the `gcov` output. In addition to the caveats mentioned previously, we note that the `gcov` profiling tool adds instrumentation to the code. The instrumentation skews the performance of the program, therefore this approach is suboptimal since it requires building the target code twice, one with instrumentation to identify code lines and other only with debugging symbols to measure the real performance.

Once the “hot” memory addresses are identified, the next step is to evict them from the cache in a tight loop, thus increasing the execution time of the process accessing those addresses. This technique allows to stealthily degrade a process without alerting the victim, since the increased execution time is not noticeable by a typical user. Performance degradation attacks have been used previously in conjunction with other side-channel attacks (see e.g. [24]).

We note that it can be difficult and time consuming to identify the “hot” memory addresses to degrade that result in the best information leak. To that end, we follow a similar but faster and more quantitative approach, potentially more accurate since it leverages additional metrics. Similar to [2] we test the efficiency of the attack for several candidate memory lines. We compare cache-misses between a regular modular inversion and a degraded modular inversion execution, resulting in a list of the “hottest” memory lines, building the code only once with debugging symbols and using hardware register counters.

The `perf` command in Linux offers access to performance counters—CPU hardware registers counting hardware events (e.g. CPU cycles, instructions executed, cache-misses and branch mispredictions). We execute calls to OpenSSL’s modular inverse operation, counting the number of cache-misses during a regular execution of the operation. Next, we degrade—by flushing in a loop from the cache—one memory line at a time from the caller `BN_mod_inverse` and callees `BN_rshift1`, `BN_rshift`, `BN_uadd`, `bn_add_words`, `BN_usub`.

The `perf` command output gives us the real count of cache-misses during the regular execution of `BN_mod_inverse`, then under degradation of each candidate memory line. This effectively identifies the “hottest” addresses during a modular inverse operation with respect to both the cache and the actual malicious processes we will use during the attack.

Table 2 summarizes the results over 1,000 iterations of a regular modular inversion execution versus the degradation of different candidate memory lines identified using our technique. The table shows cache-miss rates ranging from ~35% (`BN_rshift` and `BN_usub`) to ~172% (`BN_rshift1`) for one degrading address. Degrading the overall 6 “hottest” addresses accessed by the `BN_mod_inverse` function results in an impressive cache-miss rate of ~1,146%.

Interestingly, the last column of Table 2 reveals the real impact of cache-misses in the execution time of the modular inversion operation. Despite the impressive cache-miss rates, the clock cycle slow down is more modest with a maximum slow down of ~18. These results suggest that in order to get a quality trace, the goal is to achieve an increased rate of cache-misses rather than a CPU clock cycle slow down because whereas the cache-misses suggest a CPU clock cycle slow down, it is not the case for the opposite direction.

The effectiveness of the attack varies for each use case and for each routine called. Some of the routines iterate over internal loops several times (e.g. `BN_rshift1`) whereas in some other routines, iteration over internal loops happens few times (e.g. `BN_usub`) or none at all. Take for example previous “hot” addresses from Table 2—degrading the most used address from each routine does not necessarily give the best result. Overall “hottest” addresses in Table 2 shows the result of choosing the best strategy for our use case, where the addresses degraded in every routine varies from multiple addresses per routine to no addresses at all.

For our use case, we observe the best results with 6 degrading addresses across two degrading processes executing in different CPU cores. Additional addresses do not provide any additional slow down, instead they impact negatively the `FLUSH+RELOAD` technique.

4.4 Improving Key Recovery

Arguably the most significant contribution of [8] is they show the LS sequence is sufficient to extract a certain number of LSBs from nonces, even when it is not known whether branch u or v gets taken. They give an algebraic method to recover these LSBs, and utilize these partial nonce bits in a lattice attack, using the formalization in [21, 22]. The disadvantage of that approach is that it fixes the number of known LSBs (denoted ℓ) per equation [8,

Table 2: perf cache-misses and CPU clock cycle statistics over 1,000 iterations for relevant routines called by the BN_mod_inverse method.

Target	Cache misses (CM)	Clock cycles (CC)	$\frac{CM}{CM_{BL}}$	$\frac{CC}{CC_{BL}}$
Baseline (BL)	13	211,324	1.0	1.0
BN_rshift1	2,396	947,925	172.6	4.4
BN_usub	489	364,399	35.2	1.7
BN_mod_inverse	956	540,357	68.9	2.5
BN_uadd	855	485,088	61.6	2.2
bn_add_words	1,124	558,839	81.0	2.6
BN_rshift	514	367,929	37.0	1.7
Previous “hot”	10,280	2,576,360	740.5	12.1
Overall “hottest”	15,910	3,817,748	1,146.2	18.0

Sec. 5]: “when a set of signatures are collected such that, for each of them, $[\ell]$ bits of the nonce are known, a set of equations . . . can be obtained and the problem of finding the private key can be reduced to an instance of the [HNP].” Fixing ℓ impacts their results in two important ways. First, since their lattice utilizes a fixed ℓ , they focus on the ability to algebraically recover only a fixed number of bits from the LS sequence. From [8, Tbl. 1], our target implementation is similar to their “Standard-M0” target, and they focus on $\ell \in \{8, 12, 16, 20\}$. For example, to extract $\ell = 8$ LSBs they need to query on average 4 signatures, discarding all remaining signatures that do not satisfy $\ell \geq 8$. Second, this directly influences the number of signatures needed in the lattice phase. From [8, Tbl. 2-3], for 256-bit n and $\ell = 8$, they require 168 signatures. This is because they are discarding three out of four signatures on average where $\ell < 8$, then go on to construct a $d + 1$ -dimension lattice where $d = 168/4 = 42$ from the signatures that meet the $\ell \geq 8$ restriction. The metric of interest from the attacker perspective is the number of required signatures.

In this section, we improve with respect to both points—extracting a varying number of bits from every nonce, subsequently allowing our lattice problem to utilize every signature queried, resulting in a significantly reduced number of required signatures.

Extracting nonce bits. Rather than focusing on the average number of required signatures as a function of a number of target LSBs, our approach is to examine the average number of bits extracted as a function of LS sequence length. We empirically measured this quantity by generating β_i uniformly at random from $\{1 \dots n - 1\}$ for P-256 n , running the BEEA on β_i and n to obtain the ground truth LS sequence, and taking the first j operations from this sequence. We then grouped the β_i by these length- j subsequence values, and finally determined the maximal shared LSBs value of each group. Intuitively, this maps any length- j subsequence to a known LSBs value. For example, a sequence beginning LLS has $j = 3$, $\ell = 3$,

$a = 4$ interpreted as a length-3 subsequence that leaks 3 LSBs with a value of 4.

We performed 2^{26} trials (i.e. $1 \leq i \leq 2^{26}$) for each length $1 \leq j \leq 16$ independently and Figure 6 contains the results (see Table 6 in the appendix for the raw data). Naturally as the length of the sequence grows, we are able to extract more bits. But at the same time, in reality for practical side-channels longer sequences are more likely to contain trace errors (i.e. incorrectly inferred LS sequences), ultimately leading to nonsensical lattice problems for key recovery. So we are looking for the right balance between these two factors. Figure 6 allows us to draw several conclusions, including but not limited to: (1) Sequences of length 5 or more allow us to extract a minimum of 3 nonce bits per signature; (2) Similarly length 7 or more for a minimum of 4 nonce bits; (3) The average number of bits extracted grows rapidly at first, then the growth slows as the sequence length increases. This observation pairs nicely with the nature of side-channels: attempting to target longer sequences (risking trace errors) only marginally increases the average number of bits extracted. From the lattice perspective, $\ell \geq 3$ is a practical requirement [21, Sec. 4.2] so in that respect sequences of length 5 is the minimum to guarantee that every signature can be used as an equation for the lattice problem.

To summarize, the data used to produce Figure 6 allows us to essentially build a dictionary that maps LS sequences of a given length to an (ℓ_i, a_i) pair, which we now define and utilize.

Recovering private keys. We follow the formalization of [21, 22] with the use of per-equation ℓ_i due to [4, Sec. 4]. Extracted from our side-channel, we are left with equations $k_i = 2^{\ell_i} b_i + a_i$ where ℓ_i and a_i are known, and since $0 < k_i < n$ it follows that $0 \leq b_i \leq n/2^{\ell_i}$. Denote $\lfloor x \rfloor_n$ modular reduction of x to the interval $\{0 \dots n - 1\}$ and $\lfloor x \rfloor$ to the interval $\{-(n-1)/2 \dots (n-1)/2\}$. Define the following (attacker-known) values.

$$t_i = \lfloor r_i / (2^{\ell_i} s_i) \rfloor_n$$

$$\hat{u}_i = \lfloor (a_i - h_i / s_i) / 2^{\ell_i} \rfloor_n$$

It now follows that $0 \leq \lfloor \alpha t_i - \hat{u}_i \rfloor_n < n/2^{\ell_i}$. Setting

$$u_i = \hat{u}_i + n/2^{\ell_i+1}, \text{ we obtain}$$

$$v_i = \lfloor \alpha t_i - u_i \rfloor_n \leq n/2^{\ell_i+1},$$

i.e. integers λ_i exist such that $|\alpha t_i - u_i - \lambda_i n| \leq n/2^{\ell_i+1}$ holds. The u_i approximate αt_i since they are closer than a uniformly random value from $\{1 \dots n - 1\}$, leading to an instance of the HNP [5]: recover α given many (t_i, u_i) pairs.

Consider the rational $d + 1$ -dimension lattice gener-

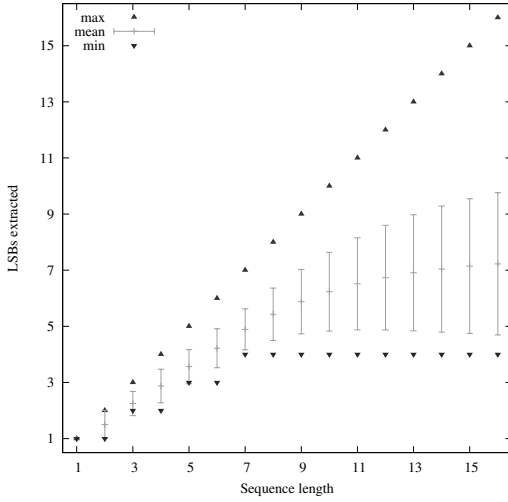


Figure 6: Empirical number of extracted bits for various sequence lengths. Each sequence length consisted of 2^{26} trials, over which we calculated the mean (with deviation), maximum, and minimum number of recovered LSBs. Error bars are one standard deviation on each side. See Table 6 in the appendix for the raw data.

ated by the rows of the following matrix.

$$B = \begin{bmatrix} 2^{\ell_1+1}n & 0 & \dots & \dots & 0 \\ 0 & 2^{\ell_2+1}n & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & 0 & \vdots \\ 0 & \dots & 0 & 2^{\ell_{d+1}}n & 0 \\ 2^{\ell_1+1}t_1 & \dots & \dots & 2^{\ell_{d+1}}t_d & 1 \end{bmatrix}$$

Setting

$$\begin{aligned} \vec{x} &= (\lambda_1, \dots, \lambda_d, \alpha) \\ \vec{y} &= (2^{\ell_1+1}v_1, \dots, 2^{\ell_{d+1}}v_d, \alpha) \\ \vec{u} &= (2^{\ell_1+1}u_1, \dots, 2^{\ell_{d+1}}u_d, 0) \end{aligned}$$

establishes the relationship $\vec{x}B - \vec{u} = \vec{y}$. Solving the CVP with inputs B and \vec{u} yields \vec{x} and hence α . We use the embedding strategy [13, Sec. 3.4] to heuristically reduce CVP approximations to Shortest Vector Problem (SVP) approximations. Consider the rational $d+2$ -dimension lattice generated by the rows of the following matrix.

$$\hat{B} = \begin{bmatrix} B & 0 \\ \vec{u} & n \end{bmatrix}$$

There is a reasonable chance that lattice-reduced \hat{B} will contain the short lattice basis vector $(\vec{x}, -1)\hat{B} = (\vec{y}, -n)$,

revealing α . To extend the search space, we use the randomization technique inspired by Gama et al. [12, Sec. 5], shuffling the order of t_i and u_i and multiplying by a random sparse unimodular matrix between lattice reductions.

Empirical results. Table 3 contains our empirical results for various lattice parameters targeting P-256. As part of our experiments, we were able to successfully reproduce and verify the $\ell \in \{8, 12\}$, $\lg n \approx 256$ lattice results of Cabrera Aldaya et al. [8] in our environment for comparison. While the goal is to minimize the number of required signatures, this should be weighed with observed HNP success probability, affecting search duration. From Figure 6 we focus on LS subsequence lengths $j \in \{5, 7\}$ that yield ℓ_i nonce LSBs from ranges $\{3..5\}$ and $\{4..7\}$, respectively. Again this is in contrast to [8] that fixes ℓ and discards signatures—this is the reason their signature count is much higher than the $d+2$ lattice dimension in their case, but equal in ours.

A relevant metric affecting success probability is the total number of known nonce bits for each HNP instance. Naturally as this sum approaches $\lg n$ one expects correct solutions to start emerging. On the other hand, increasing this sum demands querying more signatures, at the same time increasing d and lattice methods become less precise. For a given HNP instance, denote $l = \sum_{i=1}^d \ell_i$, i.e. the total number of known nonce bits over all the equations for the particular HNP instance. Table 3 denotes μ_l the mean value of l over all successful HNP instances—intuitively tracking how many known nonce bits needed in total to reasonably expect success.

We ran 200 independent trials for each set of parameters on a computing cluster with Intel Xeon X5650 nodes. We allowed each trial to execute at most four hours, and we say successful trials are those HNP instances recovering the private key within this allotted time. Our lattice implementation uses Sage software with BKZ [25] reduction, block size 30.

To summarize, utilizing every signature in our HNP instances leads to a significant improvement over previous work with respect to both the number of required signatures and amount of side-channel data required.

5 Attacking Applications

OpenSSL is a shared library and therefore any vulnerability present in it can potentially be exploited from any application linked against it. This is the case for the present work and to demonstrate the feasibility of our attack in a concrete real-life scenario, we focus on two applications implementing two ubiquitous security protocols: TLS within stunnel and SSH within OpenSSH.

OpenSSL provides ECDSA functionality for both applications and therefore we mount our attack against

Table 3: P-256 ECDSA lattice attack improvements for BEEA leakage. Empirical values are over 200 trials (4hr max trial duration). Lattice dimension is $d + 2$. The number of leaked LSBs per nonce is ℓ . LS subsequence length is j . The average total number of leaked nonce bits per successful HNP instance is μ_l . CPU time is the median.

Source	Signatures	d	ℓ	j	μ_l	Success Rate (%)	CPU Minutes
Prev. [8]	168	42	8	—	336.0	100.0	0.7
Prev. [8]	312	24	12	—	288.0	100.0	0.6
This work	50	50	{4..7}	7	249.7	14.0	79.5
This work	55	55	{4..7}	7	268.8	98.0	1.7
This work	60	60	{4..7}	7	293.4	100.0	0.7
This work	70	70	{3..5}	5	258.2	5.0	130.8
This work	80	80	{3..5}	5	286.1	94.5	13.2
This work	90	90	{3..5}	5	321.2	100.0	4.0

OpenSSL’s ECDSA running within them. More precisely, this section describes the tools and the setup followed to successfully exploit the vulnerability within these applications. In addition, we explain the relevant messages collected for each application, later used for private key recovery together with the trace data and the signatures.

5.1 TLS

Stunnel⁸ is a popular portable open source software application that forwards network connections from one port to another and provides a TLS wrapper. Network applications that do not natively support TLS communication benefit from the use of stunnel. More precisely, stunnel can be used to provide a TLS connection between a public port exposing a TLS-enabled network service and a localhost port providing a non-TLS network service. It links against OpenSSL to provide TLS functionality.

For our experiments, we used stunnel 5.39 compiled from stock source and linked against OpenSSL 1.0.1u. We generated a P-256 ECDSA certificate for the stunnel service and chose the ECDHE-ECDSA-AES128-SHA TLS 1.2 cipher suite.

In order to collect digital signature and digest tuples, we wrote a custom TLS client that connects to the stunnel service. Our TLS client initiates TLS connections, collects the protocol messages and continues the handshake until it receives the `ServerHelloDone` message, then it drops the connection. The protocol messages contain relevant information for the attack. The `ClientHello` and `ServerHello` messages contain each a 32-byte random field, in practice these bytes represent a 4-byte UNIX timestamp concatenated with a 28-byte nonce. The `Certificate` message contains the P-256

ECDSA certificate generated for the stunnel service. The `ServerKeyExchange` message contains ECDH key exchange parameters including the curve type (named `_curve`), the curve name (`secp256r1`) and the `SignatureHashAlgorithm`. Finally, the digital signature itself is sent as part of the `ServerKeyExchange` message. The ECDSA signature is over the concatenated string

```
ClientHello.random + ServerHello.random +
ServerKeyExchange.params
```

and the hash function is SHA-512, proposed by the client in the `ClientHello` message and accepted by the server in the `SignatureHashAlgorithm` field (explicit values `0x06`, `0x03`). Our TLS client saves the hash of the concatenated string and the DER-encoded ECDSA signature sent by the server.

In order to achieve synchronization between the spy and the victim processes, our spy process is launched prior to the TLS handshakes, therefore it collects the trace for each ECDSA signature performed during the handshakes, then it stops when the `ServerHelloDone` message is received. The process is repeated as needed to build up a set of distinct trace, digital signature, and digest tuples. Section 5.3 contains accuracy results for several LS subsequence patterns for an stunnel victim.

5.2 SSH

OpenSSH⁹ is a widely used open source software suite to provide secure communication over an insecure channel. OpenSSH is a set of tools implementing the SSH network protocol and it is typically linked against OpenSSL to perform several cryptographic operations, including digital signatures (excluding `ed25519` signatures) and key exchange.

For our experiments, we used OpenSSH 7.4p1 compiled from stock source and linked against OpenSSL 1.0.1u. The ECDSA key pair used by the server and targeted by our attack is the default P-256 key pair generated during installation of OpenSSH.

Following a similar approach to Section 5.1, we wrote a custom SSH client that connects to the OpenSSH server to collect digital signatures and digest tuples. At the same time, our spy process running on the server side collects the timing signals leaked by the server during the handshake.

Relevant to this work, the OpenSSH server was configured with the `ecdsa-sha2-nistp256` host key algorithm and the default P-256 key pair. After the initial `ClientVersion` and `ServerVersion` messages, the protocol defines the Diffie-Hellman key exchange parameters, the signature algorithm and the hash function

⁸<https://www.stunnel.org>

⁹<http://www.openssh.com/>

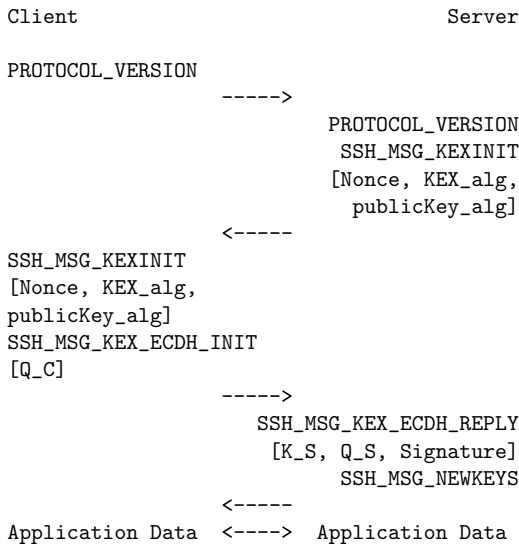


Figure 7: ECC SSH handshake flow with corresponding parameters from all the messages to construct the digest. Our spy process collects timing traces in parallel to the server’s ECDSA sign operation, said digital signature being included in a SSH_MSG_KEX_ECDH_REPLY field and collected by our client.

identifiers in the SSH_MSG_KEXINIT message. To provide host authentication by the client and the server, a 16-byte random nonce is included in the SSH_MSG_KEXINIT message. The SSH_MSG_KEX_ECDH_REPLY¹⁰ message contains the server’s public host key K_S (used to create and verify the signature), server’s ECDH ephemeral public key Q_S (used to compute the shared secret K in combination with the client’s ECDH ephemeral public key Q_C) and the signature itself. The ECDSA signature is over the hash of the concatenated string

ClientVersion + ServerVersion +
Client.SSH_MSG_KEXINIT +
Server.SSH_MSG_KEXINIT +
 $K_S + Q_C + Q_S + K$

Our SSH client was configured to use `ecdh-sha2-nistp256` and `ecdsa-sha2-nistp256` as key exchange and public key algorithms, respectively.

Similar to the previous case, our SSH client saves the hash of the concatenated string and the raw bytes of the ECDSA signature sent by the server. To synchronize the spy and victim processes, our spy process is launched prior to the SSH handshakes and it stops when the SSH_MSG_NEWKEYS message is received, therefore it collects

¹⁰<https://tools.ietf.org/html/rfc5656>

Table 4: Accuracy for length $j = 5$ subsequences over 15,000 TLS/SSH handshakes.

Pattern	ℓ_i	a_i	Accuracy (%)	
			TLS	SSH
LLLLL	5	0	77.9	73.3
SLLLL	4	1	99.8	98.0
LSLLL	4	2	99.3	98.9
SLSLL	3	3	98.9	97.2
LLSLL	4	4	98.0	96.7
SLLSL	3	5	95.8	95.5
LSLSL	3	6	85.5	97.2
SLSLS	3	7	99.2	97.8
LLLSL	4	8	93.3	92.5
SLLLS	4	9	94.4	94.6
LSLLS	4	10	81.1	93.5
LLSLS	4	12	96.4	96.7
LLLLS	5	16	89.8	85.0

the trace for each ECDSA signature performed during the handshakes. All the protocol messages starting from SSH_MSG_NEWKEYS and any client responses are not required by our attack, therefore the client drops the connection and repeats the process as needed to build up a set of distinct trace, digital signature, and digest tuples. Section 5.3 contains accuracy results for several LS subsequence patterns for an SSH server victim.

5.3 Attack Results

Procurement accuracy. Table 4 and Table 5 show the empirical accuracy results for patterns of length $j = 5$ and $j = 7$, respectively. These patterns represent the beginning of the LS sequence in the context of OpenSSL ECDSA executing in real world applications (TLS via stunnel, SSH via OpenSSH). From our empirical results we note three trends: (1) Similar to previous works [4, 24, 27], the accuracy of the subsequence decreases as ℓ increases due to the deviation in the right-shift operation width. (2) The accuracy also decreases for subsequences containing several contiguous right-shift operations, e.g. first and last rows, due to the variable width of right-shift operations within a single trace. (3) SSH traces are slightly noisier than TLS traces; we speculate this is due to the computation of the ECDH shared secret prior to the ECDSA signature itself. Using our improved degradation technique (Section 4.3) we can recover a with very high probability, despite the speed of the modular inversion operation and the imperfect traces. **Key recovery.** We close with a few data points for our end-to-end attack, here focusing on TLS. In this context, end-to-end means all steps from the attacker perspective—i.e. launching the degrade processes,

Table 5: Accuracy for length $j = 7$ subsequences over 15,000 TLS/SSH handshakes.

Pattern	ℓ_i	a_i	TLS		SSH	
			Accuracy (%)	Accuracy (%)	Accuracy (%)	Accuracy (%)
LLLLLLL	7	0	43.8		30.1	
SLLLLLS	5	1	93.4		93.1	
LSLLLLS	6	2	82.6		88.0	
SLSLLSL	4	3	94.8		93.4	
LLSLLLL	6	4	92.9		86.4	
SLLSLSL	4	5	95.2		94.1	
LSLSLLS	5	6	79.2		92.3	
SLSLSLL	4	7	98.8		96.6	
LLLSLLL	6	8	84.8		80.5	
SLLLSLL	5	9	80.0		81.1	
LSLLSLS	5	10	80.8		90.9	
SLSLLLS	5	11	91.7		85.4	
LLSLSLL	5	12	94.3		94.5	
SLLSLSL	5	13	90.9		90.6	
LSLSLSL	4	14	83.5		95.1	
SLSLSLS	4	15	97.8		97.1	
LLLSLLL	6	16	87.7		83.8	
SLLLLLL	6	17	92.0		92.4	
LSLLLLS	5	18	81.8		90.7	
LLSLLSL	5	20	94.3		94.7	
LSLSLLL	5	22	80.0		91.5	
LLLSLSL	5	24	94.4		91.1	
SLLLSLS	5	25	94.3		94.3	
LSLLSLL	5	26	74.7		86.1	
SLSLLLL	5	27	92.9		89.7	
LLSLSLS	5	28	94.6		93.6	
SLLSLLL	5	29	85.4		84.8	
LLLLLSL	6	32	65.7		61.1	
LSLLLLL	6	34	91.5		91.5	
LLSLLLS	6	36	93.0		89.3	
LLLSLLS	6	40	89.0		88.5	
LLLLSLS	6	48	87.2		82.7	
SLLLLLS	6	49	86.8		85.5	
LLLLLLS	7	64	25.6		33.0	

launching the spy process, and launching our custom TLS client. Finally, repeating these steps to gather multiple trace and signature pairs, then running the lattice attack for key recovery. That is, no steps in the attack chain are abstracted away.

The experiments for Table 3 assume perfect traces. However, as seen in Table 4 and Table 5, while we observe quite high accuracy, in our environment we are unable to realize absolutely perfect traces. Trace errors will occur, and lattice methods have no recourse to compensate for them. We resort to oversampling and randomized brute force search to achieve key recovery in practice.

For the $j = 5$ case, we procured 150 signatures with

(potentially imperfect) trace data. Consulting Table 3, we took 400 random subsets of size 80 from this set and ran lattice attack instances on a computing cluster. The first instance to succeed in recovering the private key did so in roughly 8 minutes. Checking the ground truth afterwards, 142 of these original 150 traces were correct, i.e. $\sim 0.18\%$ of all possible subsets are error-free. This successful attack is consistent with the probability $1 - (1 - 0.0018)^{400} \approx 51.4\%$.

Similarly for the $j = 7$ case, we procured 150 signatures with (potentially imperfect) trace data. Consulting Table 3, we took 400 random subsets of size 55 from this set and ran lattice attack instances on a computing cluster. The first instance to succeed in recovering the private key did so in under a minute. Checking the ground truth afterwards, 137 of these original 150 traces were correct, i.e. $\sim 0.19\%$ of all possible subsets are error-free. This successful attack is also consistent with the probability $1 - (1 - 0.0019)^{400} \approx 53.3\%$.

It is worth noting that with this naïve strategy, it is always possible to trade signatures for more offline search effort. Moreover, it is possible to traverse the search space by weighting trace data subsets according to known pattern accuracy, e.g. explore patterns with accuracy $\geq 95\%$ sooner.

6 Conclusion

In this work, we disclose a new vulnerability in widely-deployed software libraries that causes ECDSA nonce inversions to be computed with the BEEA instead of a code path with microarchitecture attack mitigations. We design and demonstrate a practical cache-timing attack against this insecure code path, leveraging our new performance degradation metric. Combined with our improved nonce bits recovery approach and lattice parameterization, this enable us to recover P-256 ECDSA private keys from OpenSSL despite constant-time scalar multiplication. As far as we are aware, this is the first cache-timing attack targeting nonce inversion in OpenSSL, and furthermore the first side-channel attack against cryptosystems leveraging its constant-time P-256 scalar multiplication methods. Our contributions traverse both practice and theory, recovering keys with as few as 50 signatures and corresponding traces.

Stepping back from the concrete side-channel attack we realized here, our improved nonce bit recovery approach coupled with tuned lattice parameters demonstrates that even small leaks of BEEA execution can have disastrous consequences. Observing as few as the first 5 operations in the LS sequence allows every signature to be used as an equation for the lattice problem. Moreover, our work highlights the fact that constant-time considerations are ultimately about the software stack, and not

necessarily a single component in isolation.

The rapid development of cache-timing attacks paired with the need for fast solutions and mitigations led to the inclusion of the `BN_FLG_CONSTTIME` flag in OpenSSL. Over the years, the flag proved to be useful when introducing new constant-time implementations, but unfortunately its usage is now beyond OpenSSL's original design. As new cache-timing attacks emerged, the usage of the flag increased throughout the library. At the same time the programming error probability increased, and many of those errors permeated to forks such as LibreSSL and BoringSSL. The recent exploitation surrounding the flag's usage, this work included, highlights it as a prime example of why failing securely is a fundamental concept in security by design. For example, P-256 takes the constant-time scalar multiplication code path by default, oblivious to the flag, while in stark contrast modular inversion relies critically on this flag being set to follow the code path with microarchitecture attack mitigations.

Following responsible disclosure procedures, we reported the issue to the developers of the affected products after our findings. We lifted the embargo in December 2016. Despite OpenSSL's 1.0.1 branch being a standard package shipped with popular Linux distributions such as Ubuntu (12.04 LTS and 14.04 LTS), Debian (7.0 and 8.0), and SUSE, it reached EOL in January 2017. Backporting security fixes to EOL packages is a necessary and challenging task, and to contribute we provide a patch to mitigate our attack. OpenSSL assigned CVE-2016-7056 based on our work. See the appendix for the patch.

Acknowledgments

We thank Tampere Center for Scientific Computing (TCSC) for generously granting us access to computing cluster resources.

Supported in part by Academy of Finland grant 303814.

This research was supported in part by COST Action IC1306.

The first author was supported in part by the Pekka Ahonen Fund through the Industrial Research Fund of Tampere University of Technology.

References

- [1] ACHIÇMEZ, O., GUERON, S., AND SEIFERT, J. 2007. New branch prediction vulnerabilities in OpenSSL and necessary software countermeasures. In *Cryptography and Coding, 11th IMA International Conference, Cirencester, UK, December 18-20, 2007, Proceedings*. 185–203.
- [2] ALLAN, T., BRUMLEY, B. B., FALKNER, K. E., VAN DE POL, J., AND YAROM, Y. 2016. Amplifying side channels through performance degradation. In *Proceedings of the 32nd Annual Confer-*

ence on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016, S. Schwab, W. K. Robertson, and D. Balzarotti, Eds. ACM, 422–435.

- [3] ARAVAMUTHAN, S. AND THUMPARTHY, V. R. 2007. A parallelization of ECDSA resistant to simple power analysis attacks. In *2007 2nd International Conference on Communication Systems Software and Middleware*. 1–7.
- [4] BENGER, N., VAN DE POL, J., SMART, N. P., AND YAROM, Y. 2014. "Ooh aah... just a little bit" : A small amount of side channel can go a long way. In *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, L. Batina and M. Robshaw, Eds. Lecture Notes in Computer Science, vol. 8731. Springer, 75–92.
- [5] BONEH, D. AND VENKATESAN, R. 1996. Hardness of computing the most significant bits of secret keys in Diffie-Hellman and related schemes. In *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*. 129–142.
- [6] BRUMLEY, B. B. AND HAKALA, R. M. 2009. Cache-timing template attacks. In *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings*, M. Matsui, Ed. Lecture Notes in Computer Science, vol. 5912. Springer, 667–684.
- [7] BRUMLEY, B. B. AND TUVERI, N. 2011. Remote timing attacks are still practical. In *Computer Security - ESORICS 2011 - 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12-14, 2011. Proceedings*. 355–371.
- [8] CABRERA ALDAYA, A., CABRERA SARMIENTO, A. J., AND SÁNCHEZ-SOLANO, S. 2016. SPA vulnerabilities of the binary extended Euclidean algorithm. *J. Cryptographic Engineering*.
- [9] CIPRESSO, T. AND STAMP, M. 2010. Software reverse engineering. In *Handbook of Information and Communication Security*. 659–696.
- [10] FAN, S., WANG, W., AND CHENG, Q. 2016. Attacking OpenSSL implementation of ECDSA with a few signatures. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. 1505–1515.
- [11] GALLANT, R. P., LAMBERT, R. J., AND VANSTONE, S. A. 2001. Faster point multiplication on elliptic curves with efficient endomorphisms. In *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, J. Kilian, Ed. Lecture Notes in Computer Science, vol. 2139. Springer, 190–200.
- [12] GAMA, N., NGUYEN, P. Q., AND REGEV, O. 2010. Lattice enumeration using extreme pruning. In *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30 - June 3, 2010. Proceedings*, H. Gilbert, Ed. Lecture Notes in Computer Science, vol. 6110. Springer, 257–278.
- [13] GOLDBREICH, O., GOLDWASSER, S., AND HALEVI, S. 1997. Public-key cryptosystems from lattice reduction problems. In *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings*, B. S. K. Jr., Ed. Lecture Notes in Computer Science, vol. 1294. Springer, 112–131.

PUBLICATION

III

Cache-Timing Attacks on RSA Key Generation

A. C. Aldaya, C. Pereida García, L. M. Alvarez Tapia and B. B. Brumley

IACR Trans. Cryptogr. Hardw. Embed. Syst. 2019.4 (2019), 213–242

DOI: 10.13154/tches.v2019.i4.213-242

Publication reprinted with the permission of the copyright holders

Cache-Timing Attacks on RSA Key Generation

Alejandro Cabrera Aldaya¹, Cesar Pereida García²,
Luis Manuel Alvarez Tapia¹ and Billy Bob Brumley²

¹ Universidad Tecnológica de la Habana (CUJAE), Habana, Cuba
{aldaya, lalvarezt89}@gmail.com

² Tampere University, Tampere, Finland
{cesar.pereidagarcia, billy.brumley}@tuni.fi

Abstract. During the last decade, constant-time cryptographic software has quickly transitioned from an academic construct to a concrete security requirement for real-world libraries. Most of OpenSSL’s constant-time code paths are driven by cryptosystem implementations enabling a dedicated flag at runtime. This process is perilous, with several examples emerging in the past few years of the flag either not being set or software defects directly mishandling the flag. In this work, we propose a methodology to analyze security-critical software for side-channel insecure code path traversal. Applying our methodology to OpenSSL, we identify three new code paths during RSA key generation that potentially leak critical algorithm state. Exploiting one of these leaks, we design, implement, and mount a single trace cache-timing attack on the GCD computation step. We overcome several hurdles in the process, including but not limited to: (1) granularity issues due to word-size operands to the GCD function; (2) bulk processing of desynchronized trace data; (3) non-trivial error rate during information extraction; and (4) limited high-confidence information on the modulus factors. Formulating lattice problem instances after obtaining and processing this limited information, our attack achieves roughly a 27% success rate for key recovery using the empirical data from 10K trials.

Keywords: applied cryptography · public key cryptography · RSA · side-channel analysis · timing attacks · cache-timing attacks · OpenSSL · CVE-2018-0737

1 Introduction

Side-channel analysis (SCA) continues to be a serious threat against the security of systems and cryptography libraries. Specifically, microarchitecture attacks and cache-timing attacks are gaining more traction due to the severe architecture flaws recently discovered in many microprocessors [Koc+19, Lip+18]. Cache-timing attacks are attractive for attackers and researchers due to the ability to perform them semi-remotely and without special privileges. So far, practical cache-timing attacks have been developed against multiple cryptosystems, including but not limited to DSA [PGBY16], ECDSA [PGB17], DH [GVY17] and RSA [YGH16]. As a countermeasure against this type of attack, cryptography library developers such as OpenSSL and forks integrate algorithms in their codebase that execute in constant-time independently of the input values. During recent years, several researchers discovered and exploited flaws in these mitigations.

SCA research focuses mainly on cryptographic operations such as encryption, decryption, key exchange, and signature generation. All of them have in common the repeated use of the private key as input during some step of the algorithm execution, thus being able to observe and capture the leakage over several runs. In contrast, SCA research targeting key generation seems to be neglected—we speculate due to two assumptions: (1) keys are only generated once during the initial stage in a secure environment, isolated from any

possible threats; and (2) single trace attacks pose too many challenges, e.g. noise, and are not feasible.

We motivate our work with a scenario where a malicious attacker is co-located with a victim process generating RSA keys. In services such as Let’s Encrypt¹, RSA key generation is a common, regular and semi-predictable operation for web server automated certificate renewal, often performed in shared cloud environments such as Amazon Web Services (AWS), and Microsoft Azure. Recent numbers reported by Censys² suggest Let’s Encrypt is now the largest certificate issuer, therefore generating thousands of keys, and with an adoption rate of more than 60% from websites using SSL/TLS, thus highlighting the need for SCA-hardened key generation.

In this work, we present a methodology developed to identify the use of known side-channel vulnerable functions in cryptography libraries such as OpenSSL. Using our methodology, we disclose several vulnerabilities affecting the OpenSSL RSA key generation implementation. Due to the impact of our attack, the OpenSSL security team issued a security advisory and CVE-2018-0737. Moreover, we present the first practical single trace cache-timing attack against the binary GCD step used during RSA key generation leading to complete RSA private key recovery. The root cause of the vulnerability is the GCD callee function not supporting the constant-time flag, compounded by the parent function’s failure to enable it. More precisely, our attack focuses on the execution of the non constant-time binary GCD algorithm to test the coprimality between the integers $p - 1$ and $q - 1$, and the public exponent e . Finally, this work serves as a reminder that cryptography libraries should strive for a secure by default approach, thus avoiding several side-channel attacks that still might be lurking in the codebase.

Our contributions in this work are the following: (1) We develop a methodology to identify insecure code paths through known side-channel vulnerable functions still in use by cryptography libraries, and use it to identify and exploit a flaw in OpenSSL that allows a practical single trace cache-timing attack against RSA key generation (Section 3.1); (2) We combine several techniques from cache-timing attacks and power analysis to capture traces during binary GCD execution and process them in order to obtain a sequence of shift and subtraction operations, i.e. algorithm state, related with prime values p and q (Section 3.4); (3) Building on existing RSA key recovery work, we propose a novel error correction algorithm for noisy RSA primes that allows us to recover roughly 50% of bits for each prime (Section 4); (4) We implement a lattice attack that factors RSA-2048 keys knowing 522 bits of one prime. We perform an end-to-end attack for 10K independent keys achieving roughly a 27% success rate, with room for improvement (Section 5).

2 Background

2.1 The RSA Cryptosystem

RSA is a public key cryptosystem invented in 1978 [RSA78]. An RSA public key is a tuple of integers (N, e) where p and q are primes and $N = pq$ holds, and furthermore $ed = 1 \pmod{(p - 1)(q - 1)}$ holds, implying both e and d are odd. For the remainder of this paper, we restrict to standardized RSA- n that mandates for n -bit N both p and q have bit-length $n/2$ and furthermore $2^{16} < e < 2^{256}$ holds [Fip]. For efficiency reasons, $e = 65537$ is the most common choice.

The private key is the tuple $\mathbf{sk} = (p, q, d, dp, dq, iq)$ where the latter three are Chinese Remainder Theorem (CRT) values not relevant to this work. For well-chosen parameters, recovering the private key from the public key is believed to be as hard as factoring

¹<https://letsencrypt.org/>

²https://censys.io/certificates/report?q=tags%3Atrusted&field=parsed.issuer.organization.raw&max_buckets=50

N [Hin10]. Regarding security, the current minimum recommended RSA key size is 2048 bits, implying that p and q are 1024-bit primes. Applications of RSA in cryptography include public key encryption and digital signatures.

The secrecy of p and q is essential for RSA, moreover, partial knowledge of either value can lead to polynomial-time factoring algorithms. In his groundbreaking work, Coppersmith [Cop96] proved that knowing half of the bits from one prime suffices to factor N in polynomial time, a critical point in our attack (see Section 5).

Algorithm 1: OpenSSL RSA key generation

Input: Key size n and public exponent e .

Output: Public and private key pair.

```

1 begin
2   while gcd( $p - 1, e$ )  $\neq 1$  do
3      $p \leftarrow \text{rand } n/2\text{-bit prime}$            /* Generate  $p$  */
4   while gcd( $q - 1, e$ )  $\neq 1$  do
5      $q \leftarrow \text{rand } n/2\text{-bit prime}$            /* Generate  $q$  */
6    $d \leftarrow e^{-1} \bmod (p - 1)(q - 1)$          /* Priv exp */
7    $dp \leftarrow d \bmod (p - 1)$ 
8    $dq \leftarrow d \bmod (q - 1)$                  /* CRT parameters */
9    $iq \leftarrow q^{-1} \bmod p$ 
10  return ( $N, e$ ), ( $d, p, q, dp, dq, iq$ )

```

OpenSSL's RSA key generation closely resembles Algorithm 1. The first steps aim at generating random secret primes p and q , during which two loops ensure that $(p - 1)$ and $(q - 1)$ are coprime with e . Steps 7-9 are not relevant to this work.

Algorithm 1 involves computing (at least) two GCDs and two modular inversions. Binary GCD algorithms are a common implementation choice for both of these operations—a description follows.

2.2 Binary GCD Algorithms

Stein [Ste67] proposed the binary greatest common divisor algorithm (binary GCD) in 1967. This algorithm computes the GCD of two integers a and b employing only right-shift operations and subtractions (Algorithm 2). This approach is very attractive in cryptography as it performs very well, especially with large inputs.

In OpenSSL, GCD computations use the function `BN_gcd`, a high level wrapper to the function `euclid` that is one implementation of the binary GCD. Note however, that it does not follow the *classic* algorithm structure (c.f. Algorithm 2). Regardless, its flow can be analyzed using the classic variant since their equivalence can be easily verified. If an adversary can distinguish a right-shift from a subtraction operation, the algorithm state can be recovered [AGS07, ACSS17, PGB17]. We expand on this concept later in this section.

Finally, it is worth noting that with respect to RSA key generation, Step 4 never executes since one of the inputs is always odd.

2.3 Binary GCD: Side-Channel Analysis

The execution flow of Algorithm 2 is highly dependent of its inputs. In Algorithm 2 some execution flow relevant steps are highlighted. The *u-loop* and *v-loop* are the loops that remove all power-of-two divisors in variables u and v at each iteration. The *sub-step* executes when both variables are odd, consisting of a single subtraction.

Algorithm 2: Binary GCD**Input:** Integers a and b such that $0 < a < b$.**Output:** Greatest common divisor of a and b .

```

1 begin
2    $u \leftarrow a, v \leftarrow b, i \leftarrow 0$ 
3   while even( $u$ ) and even( $v$ ) do
4      $u \leftarrow u/2, v \leftarrow v/2, i \leftarrow i + 1$ 
5   while  $u \neq 0$  do
6     while even( $u$ ) do
7        $u \leftarrow u/2$                                 /* u-loop */
8     while even( $v$ ) do
9        $v \leftarrow v/2$                                 /* v-loop */
10    if  $u \geq v$  then
11       $u \leftarrow u - v$                                 /* sub-step */
12    else
13       $v \leftarrow v - u$ 
14  return  $v \cdot 2^i$ 

```

Consistent with the existing literature [ACSS17, PGB17], we encode the execution flow sequence of this algorithm with two symbols ‘L’ and ‘S’ representing right-shift and subtraction, respectively. Another representation uses two variables Z_i and X_i defined³ in [ACSS17] as follows: (1) Z_i stores the number of right-shifts at iteration i . (2) X_i stores a binary value to represent the result of the condition (Step 10 in Algorithm 2) at iteration i . $X_i = ‘u’$ means the condition was true while $X_i = ‘v’$ the opposite.

Figure 1 shows an LS-sequence example of an execution flow. The sequence reads from left to right: $Z_0 = 3, Z_1 = 2, Z_2 = 1, Z_3 = 5$, etc.

LLLSLLSLSLLLLLSL...LS

Figure 1: LS-encoded binary GCD execution flow example.

Regarding SCA, there are three different models for analyzing Algorithm 2 leakage. Each model originally targets the Binary Extended Euclidean Algorithm (BEEA) for computing modular inverses. However, they also apply to Algorithm 2 because the models exploit the execution flow leakage w.r.t. variables u and v , and said flow is the same for both algorithms when executed with the same input pair.

All-or-nothing. Aciçmez, Gueron, and Seifert [AGS07] and Aravamuthan and Thumparthy [AT07] independently proposed this model in 2007. It requires that the adversary knows *all* Z_i and X_i to recover algorithm inputs.

Partial. Aldaya, Cabrera Sarmiento, and Sánchez-Solano [ACSS17] recently proposed this model, an algebraic approach that relates the number of known Z_i, X_i with the number of input bits that can be recovered. In comparison with the previous model, this approach is more flexible as it can extract information from partial knowledge of the execution flow. In this model, the number of recovered bits grows with the number of Z_i and X_i

³Equivalent to SHIFTS[i] and SUBS[i] definition by Aciçmez, Gueron, and Seifert [AGS07]

that an adversary knows. Our work employs this model (see Section 3.2 for this selection rationale).

Look-up. Pereida García and Brumley [PGB17] proposed a model that also allows partial recovery, but instead of an algebraic approach it employs a table look-up. The adversary generates a table that relates every LS-sequence of a given length with the corresponding partial input bits. This model performs better than the previous when the number of bits to recover is small as it captures some algebraic equivalences not previously modeled. However, it becomes impractical for recovering a large number of bits (i.e. longer LS-sequences). The computational complexity (time and storage) for creating a table containing all possible LS-sequences increases exponentially on the LS-sequence length.

2.4 The Flush+Reload Technique

This technique is a cache-based side-channel attack technique targeting the Last-Level Cache (LLC) and used during our attack. FLUSH+RELOAD is a high resolution, high accuracy and high signal-to-noise ratio technique that positively identifies accesses to specific memory lines. It relies on cache sharing between processes, typically achieved through the use of shared libraries or page de-duplication.

A round of attack consists of three phases: (1) The attacker evicts (i.e. flushes) the target memory line from the cache. (2) The attacker waits some time so the victim has an opportunity to access the memory line. (3) The attacker measures the time it takes to reload the memory line. The latency measured in the last step tells whether or not the memory line was accessed by the victim during the second step of the attack, i.e. identifies cache-hits and cache-misses, in addition to information about the cache activity for detecting spy preemptions.

Consult [YF14, All+16, PGBY16] for more information on the technique and discussions about the challenges during attack setup due to processor optimizations and different architectures.

2.5 Error Correction in RSA Keys

Following Section 2.1, an RSA private key is composed by a six element tuple $\mathbf{sk} = (p, q, d, dp, dq, iq)$. The knowledge of any of these elements directly allows breaking the system. In addition, these elements contain redundancy and are related through public information. For example, as $N = pq$, the relation $N \equiv pq \pmod{2^n}$ holds for every positive integer n . Therefore the knowledge of the first n bits of p directly reveals the same amount on q .

Motivated by some implementation attacks such as side-channel and cold-boot attacks, factoring N from a noisy version of \mathbf{sk} is an active research line since the pioneering works of Percival [Per05] and Heninger and Shacham [HS09].

The number of elements in a noisy version of \mathbf{sk} depends on the data leak. For example, Percival [Per05] assumes a noisy version of (dp, dq) whereas Heninger and Shacham [HS09] consider different versions of noisy \mathbf{sk} , such as (p, q, d, dp, dq) and (p, q) . The number of elements in the noisy \mathbf{sk} is often denoted in literature as m .

Regarding this work, the case of $m = 2$ is particularly interesting since a FLUSH+RELOAD attack allows an attacker to obtain a pair of noisy LS sequences related to p and q , therefore we focus further related work analysis on the $m = 2$ case.

Another implementation attack property is the nature of errors that produces the noisy \mathbf{sk} —termed *noise model* by Henecka, May, and Meurer [HMM10]. They defined the noise model of Percival [Per05] and Heninger and Shacham [HS09] works as the *erasure model*. In these works, the adversary knows which bit positions contain valid data, and likewise exactly at which bit position data is missing.

On the other hand, Henecka, May, and Meurer [HMM10] proposed an algorithm for correcting errors in RSA keys from noisy \mathbf{sk} where some bits are flipped. Correcting errors in this *bit-flip model* is more challenging than in the erasure model [HMM10], however the authors showed that for $m = 2$ it is possible to achieve a success rate of 24% if the probability of a single bit-flip is less than 0.084. Paterson, Polychroniadou, and Sibborn [PPS12] also studied this error model but considering that a flip $0 \rightarrow 1$ has a different probability than $1 \rightarrow 0$.

Kunihiro [Kun15] analyzed *erasure* and *bit-flip* models in a combined approach proposing an algorithm and a theoretical analysis of the bounds on erasure and bit-flip rates to succeed, improving the allowed bit-flip rate up to 0.11 in a scenario without erasures. For a detailed survey on these approaches we suggest the reader to consult [Kun18].

In addition to these error correction algorithms, some authors employ multiple traces to remove noise from \mathbf{sk} . For example, in very different attack scenarios and completely independent research, Irazoqui, Eisenbarth, and Sunar [IES16] and Schwarz et al. [Sch+17] obtained an error rate of no more than 0.04 from a single trace attack. Then, combining the same data leakage from five traces, they corrected all the errors in their respective setting. RSA key error correction using multiple traces is an approach that only applies when the attacker can capture multiple traces of the same operation leaking data.

2.6 Related Work

Attacks on RSA keys. Over the years, cryptanalysis of RSA keys has been performed due to its widespread usage, its mathematical structure (i.e. CRT-based methods) and the ease of generating low entropy keys. One classification of attacks against RSA keys is: (1) only public key knowledge; (2) partial private key knowledge [Hin10].

The first category assumes an attacker only has knowledge of the public key (N, e) , attempting to use factoring methods such as Pollard $p - 1$ [Pol74], Pollard Rho [Pol75] and sieving methods to recover the private factors p and q . This type of attack is bound by the often sub-exponential, yet intractable, time complexity of the factoring methods, requiring massive computation time and resources. Current research achieves factorization of 768-bit RSA keys [Kle+10], therefore it has limited practical applicability and interest for an attacker.

The second category exploits partial knowledge about the private *and* public keys to perform attacks such as low exponent attacks [BM03, Wie90], side-channel attacks [YGH16, Bau+14], and Coppersmith related attacks [Cop96, Cop97], considered a universal tool to attack RSA keys with poorly chosen parameters or keys generated with poor entropy, i.e. using a faulty implementation.

In 2012, two independent teams [Hen+12, Len+12] exploited poor entropy of RSA keys in SSL certificates, SSH host keys, and PGP keys, thus allowing them to trivially factorize keys by carrying out pairwise GCD computations to recover shared prime factors among other RSA keys. Similarly in 2013, Bernstein et al. [Ber+13] analyzed the public record of RSA keys in the “Citizen Digital Certificate” database of Taiwanese citizens. The authors recovered 265 private keys by running a batch GCD computation followed by Coppersmith’s method.

In 2017, Nemeč et al. [Nem+17] discovered a critical vulnerability in the library used to generate RSA keys for identity cards, passports and Trusted Platform Modules; allowing factorization of 1024 and 2048-bit keys. Once again, this exploit was possible due to poor entropy introduced by a special mathematical structure of the prime factors that not only allowed key recovery using Coppersmith’s method but also detection of keys with this special structure.

Microarchitecture attacks on RSA. In his seminal work, Percival [Per05] demonstrated a cache-timing attack against RSA by identifying access to precomputed multipliers stored

in memory when using the Sliding Window Exponentiation (SWE) algorithm implemented in OpenSSL version 0.9.7c. To mitigate this issue, the OpenSSL team added a “constant-time” implementation of the modular exponentiation algorithm combining a fixed-window exponentiation algorithm with a scatter-gather method [Bri+06], allowing to mask table access to the multipliers. The scatter-gather method ensures the same cache lines are always accessed, irrespective of the multiplier used.

In 2016, Yarom, Genkin, and Heninger [YGH16] showed that the previous scatter-gather method implemented in OpenSSL still leaked timing information. In their work, the authors exploited cache-bank conflicts by accessing the same offsets within a cache line, these offsets depend on the multipliers used which are decided based on the private key. The attack allows 4096-bit RSA key recovery after observing 16000 decryptions on a HyperThreading architecture.

More recently, Bernstein et al. [Ber+17] performed 1024 and 2048-bit key recovery in the Libgcrypt library when computing modular exponentiations using the left-to-right sliding window method. More precisely, the authors demonstrated that the direction of the sliding window matters since it leaks more or less information depending on the encoding direction. Applying the FLUSH+RELOAD technique, paired with the algorithm by Heninger and Shacham [HS09], the authors are able to efficiently reconstruct private keys using a side-channel leak after recovering roughly 50% of the secret bits.

Aciçmez, Gueron, and Seifert [AGS07] showed information leakage in OpenSSL 0.9.8a during the modular inversion operation. When using the BEEA for modular inversion during key generation, decryption, and blinding when employing the RSA-CRT variant, these algorithms compute on secret values. Developing Simple Branch Prediction Analysis (SBPA), the authors conjecture it is feasible to deduce the outcome of branch statements using timings, recovering critical BEEA algorithm state, therefore leading to secret key recovery.

Side-channel attacks on RSA key generation. The research available on SCA against RSA key generation is limited and mostly focuses on leakages in physical devices. Finke, Gebhardt, and Schindler [FGS09] performed an attack on a custom implementation of a prime generation algorithm used for RSA key generation, analyzed using Simple Power Analysis (SPA). In 2012, Vuillaume, Endo, and Wooderson [VEW12] presented a Differential Power Analysis (DPA) template attack and fault attack on the Fermat and Miller-Rabin tests on a secure microcontroller but the authors give no additional information regarding their setup. Later on, Bauer et al. [Bau+14] analyzed the security of prime generation algorithms and the sieving process. Targeting the divisibility phase, the authors obtained more than half of the bits from the prime number generated with their own implementation and then using Coppersmith’s technique they recovered 1024-bit RSA keys. More recently, Aldaya et al. [Ald+17] analyzed the modular inversion operation used during RSA private key generation, leading to full key recovery using SPA. This attack differs from previous works because it focuses on alternative routines invoked during key generation, instead of primality tests or prime number generation.

Moreover, recent independent work examines one of the three code paths analyzed in this work (i.e. the `BN_gcd` function). Weiser, Spreitzer, and Bodner [WSB18] target RSA key generation within an Intel SGX enclave by a noiseless controlled-channel page-fault attack. Controlled-channel attacks [XCP15] are privileged attacks originating from a malicious OS targeting SGX enclaves, aligned with the SGX threat model. The most important differences compared to our work are: (1) cache-timing attacks are unprivileged and do not require escalation to kernel space (i.e. a malicious OS); and (2) controlled-channels are error-free, while cache-timing channels are far from that.

3 RSA Key Generation: New Vulnerabilities

Originally introduced with OpenSSL 0.9.7 in 2005 following [Per05], the *constant-time flag* is a boolean for BIGNUM variables handling secret information such as private keys, secret prime values, nonces, and integer scalars. When the flag is set, and the executing algorithm supports the flag, the code takes an early exit to the constant-time version of the algorithm, otherwise continues executing the default insecure version. For the sake of performance, OpenSSL defaults to non constant-time functions, assuming most operations are not secret.

3.1 Insecure Code Paths: A Methodology

Two recent works exploit the insecure default behavior of OpenSSL’s constant-time flag. Pereida García, Brumley, and Yarom [PGBY16] exploit the fact that, by design, the flag does not propagate from the source to the destination during BIGNUM copy operations. As a result, modular exponentiations during DSA sign operations took a side-channel insecure modular exponentiation path. Pereida García and Brumley [PGB17] exploit the failure to set the flag during ECDSA sign operations. In that case, the resulting scalar multiplication function is oblivious to the flag and always followed a side-channel secure path; the modular inversion function, however, requires this flag to follow its side-channel secure path.

These examples demonstrate that constant-time flag handling is tricky, hence there could be other vulnerable code paths believed to be safe. For tackling the problem of detecting such potential vulnerable code paths we developed a semi-automated tool that revises, with a single execution, multiple code paths, producing a report about them. Our methodology consists of the following steps: (1) From existing work, we create a list of previously known side-channel vulnerable functions within a library. (Here, OpenSSL.) (2) The tool utilizes the debugger to automatically set break points at lines of code, identified in the previous step, which should not be reached during security-critical operations. (3) The tool runs several security-critical commands and generates a report for calls hitting said break points.

Cryptography libraries such as OpenSSL support multiple architectures, compilation options, and implementations of the same functionality. Therefore, our tooling allows to perform exhaustive testing on the library, trying several combinations for vulnerable paths and easing the workload for SCA.

Using our tooling, w.r.t. RSA key generation we identified the following subset of known side-channel vulnerable functions of interest: (1) The function `BN_gcd` contains highly input-dependent branches that can potentially be used as a side-channel attack vector. Since the code has no early exit to a side-channel secure code path, i.e. does not check the constant-time flag at all, we blacklist the function’s entry point. (2) The function `BN_mod_inverse` executes a check for the constant-time flag at the beginning of the function, and early exits to a side-channel secure path if it is set. If the flag is not set, it continues to a side-channel insecure path. We blacklist the line immediately following the early exit. (3) The function `BN_mod_exp_mont` is analogous to the above, yet for modular exponentiation. Similarly, we blacklist the line immediately following the early exit.

Figure 2 shows a visualization of our tooling, setting breakpoints on `bn_gcd.c` (Lines 120 and 238) and `bn_exp.c` (Line 418) based on the previously blacklisted lines. Our tooling executes OpenSSL `genpkey` command to generate an RSA key, hitting the three break points multiple times, and reporting their corresponding call stacks. Naturally, hitting the break points does not guarantee a vulnerability—a deeper analysis follows.


```

INFO: Parsing source code at: ./openssl-1.0.2k
...
INFO: Breakpoints file generated: triggers.gdb
...
INFO: Monitor target command line
TOOL: gdb --batch --command=triggers.gdb --args
      openssl-1.0.2k/apps/openssl genkey -algorithm RSA
      -out private_key.pem -pkeyopt rsa_keygen_bits:2048
...
INFO: Setting breakpoints...
Breakpoint 1 at ...: file bn_exp.c, line 418.
Breakpoint 2 at ...: file bn_gcd.c, line 120.
Breakpoint 3 at ...: file bn_gcd.c, line 238.
...
INFO: Insecure code executed!
Breakpoint 1, BN_mod_exp_mont (...) at bn_exp.c:418
418 bn_check_top(a);
#0 BN_mod_exp_mont (...) at bn_exp.c:418
#1 ... in witness (...) at bn_prime.c:356
...
#2 ... in BN_is_prime_fasttest_ex (...) at bn_prime.c:329
#3 ... in BN_generate_prime_ex (...) at bn_prime.c:199
#4 ... in rsa_builtin_keygen (...) at rsa_gen.c:150
...
INFO: Insecure code executed!
Breakpoint 2, BN_gcd (...) at bn_gcd.c:120
120 int ret = 0;
#0 BN_gcd (...) at bn_gcd.c:120
#1 ... in rsa_builtin_keygen (...) at rsa_gen.c:154
...
INFO: Insecure code executed!
Breakpoint 3, BN_mod_inverse (...) at bn_gcd.c:238
238 bn_check_top(a);
#0 BN_mod_inverse (...) at bn_gcd.c:238
#1 ... in BN_MONT_CTX_set (...) at bn_mont.c:450
#2 ... in BN_is_prime_fasttest_ex (...) at bn_prime.c:319
#3 ... in BN_generate_prime_ex (...) at bn_prime.c:199
#4 ... in rsa_builtin_keygen (...) at rsa_gen.c:171
...

```

Figure 2: Testing the proposed methodology tool.

Insecure exponentiation code path. The Miller-Rabin primality test [Rab80] is the most common implementation of Algorithm 1, Lines 3 and 5. It involves choosing a random “witness” base b then computing $b^x \bmod p$ where p is the candidate prime and the relation $2^k x = p - 1$ holds. Indeed, OpenSSL’s is a straightforward implementation of these steps. Looking at the call stack for the `BN_mod_exp_mont` break point, the function `BN_is_prime_fasttest_ex` implements iterating this test for different b values to obtain prime confidence after sufficient successful trials. It carries out each trial by calling the function `witness` that performs the modular exponentiation, unfortunately calling `BN_mod_exp_mont` without setting `BN_FLG_CONSTTIME`. The algorithm continues with a classical sliding window exponentiation, potentially leaking partial information on x hence p . Note that the sliding window code path is known to be vulnerable to cache-timing attacks and was first exploited by Percival [Per05], nevertheless this leak is not relevant for our attack.

Insecure inversion code path. Related to the previous code path, as the function name `BN_mod_exp_mont` suggests, the implementation uses Montgomery arithmetic for efficiency. The Montgomery setup phase occurs in `BN_MONT_CTX_set`, computing the inverse of 2^w modulo p for w -bit architectures. Examining the call stack, the function calls `BN_mod_inverse` without setting `BN_FLG_CONSTTIME`, potentially leaking critical binary GCD algorithm state. However, in this case our terse analysis reveals the operands are not $\{2^w, p\}$ but $\{2^w, p \bmod 2^w\}$, implemented by copying the least significant word of p to a temporary `BIGNUM`. While all leaks are bad, some are worse than others—this is a nominal leak on the least significant word of p .

Insecure GCD code path. The shallowest call stack is for `BN_gcd`, called directly by `rsa_builtin_keygen` (Line 154). The function computes the GCD of e and $p - 1$ to ensure that e is invertible $\bmod (p - 1)(q - 1)$. The value $p - 1$ should remain secret, hence hitting this break point represents a potential side-channel attack vector. This is the code path we target in the remainder of this paper due to its novelty compared to insecure exponentiation.

Root cause analysis. From these results, we deduce modular inversions in OpenSSL’s RSA key generation at Steps 6 and 9 of Algorithm 1 have side-channel mitigations in place, yet GCD computations in Steps 2 and 4 lack such protection, and likewise for primality testing. The work of Acimez, Gueron, and Seifert [AGS07] induced the secure path code change, yet the impact of the academic result did not fully propagate throughout the

entirety of the RSA key generation implementation. We speculate this is a result of a simplification in Aciözmez, Gueron, and Seifert [AGS07, Sec. 2.1]: the pseudocode for key generation abstracts away the prime generation loop, and assumes a priori coprimality of e with $p - 1$ and $q - 1$ to compute d at Step 6. This allowed the authors to focus theoretical analysis on the impact of modular inversion leaks across various cryptosystems, while undoubtedly being aware of GCD and modular inversion execution flows having essentially the same branching characteristics. Alas, typical engineers are less inclined to such cryptographic subtleties—evidenced by this code path remaining vulnerable to microarchitecture side-channel attacks.

The tool. Subsequent to our work, Gridin et al. [Gri+19] expanded our methodology and tooling into a full-fledged Continuous Integration (CI) tool named *Triggerflow*⁴. It offers a different approach compared to static program analysis tools [DK17, Doy+15, Ant+17], and dynamic program analysis tools [Wan+17, Wic+18, Wei+18]. Rather than automated detection of security vulnerabilities and leakage quantification, our tool works in a white-box model where it complements other tools and assists developers to find undesired execution flows—such as non constant-time algorithm executions—and reports them back to the developers for further analysis. See [Gri+19] for more information about the goals, uses, and limitations of the tool.

3.2 Theoretical Leakage Analysis

Pereida García and Brumley [PGB17] demonstrate it is possible to recover some Z_i from OpenSSL modular inversion operations (BEEA) with cache timings during ECDSA signature generations. We are left with the following open question: *Is it possible to similarly recover binary GCD algorithm state?*

Aldaya et al. [Ald+17] analyzed RSA key generation with respect to SCA of GCD-based algorithms. The analysis focuses on the modular inversion at Step 6 of Algorithm 1, exploiting the fact that BEEA inputs have very different bit-lengths. The product $(p - 1)(q - 1)$ has 2048 bits for modern RSA key sizes, while the other input e has only 17 bits commonly.

Our vulnerability similarly exploits a large bit-length difference between inputs: the same e , but instead $p - 1$ and $q - 1$ having 1024 bits. As processing p and q are very similar regarding GCD computation, we use the prime p to present our analysis. Furthermore, we select the *partial* bit-recovery model (see Section 2.3) because (1) we will be working with noisy LS-sequences later in our full attack, thus partial recovery reduces noise influence; (2) covered later in this section, we will utilize a factoring method that inputs incomplete p ; and (3) we need to recover hundreds of bits so the *look-up* model is intractable.

The large bit-length difference between $p - 1$ and e implies that during several binary GCD iterations the condition $u \geq v$ will be true, giving the adversary partial execution flow information a priori (i.e. $X_i = 'u'$ for some iterations i). This situation holds until u (initialized to $p - 1$) stores a value of roughly the same bit-length as v (initialized to e). Therefore it divides u by two roughly $\lg(N)/2 - \lg e$ times.

According to the Z_i definition, at each iteration u loses Z_i bits. Therefore the number of iterations t that should execute before u has roughly the same bit-length as v is the minimum t that satisfies (1).

$$n = \sum_{i=1}^t Z_i \geq \lg(N)/2 - \lg e \quad (1)$$

Hence, the following question arises: *how many bits can be recovered in this setting?*

⁴Freely available, open source: <https://gitlab.com/nisec/triggerflow>

Partial recovery. Applying the *partial* model, we obtain a bit-recovery equation as follows. Assume the adversary obtains all Z_i , and t is the first iteration for which $u < v$ (i.e. $X_i = u^i$; $0 < i < t$). The values of u and v just before the *sub-step* for iterations $i < t$ are the following:

$$u_1 = p - 1, \quad v_1 = v_i = e, \quad u_{i+1} = \frac{u_i - v_i}{2^{Z_{i+1}}}$$

The invariant $u_i - v_i \equiv 0 \pmod 2$ holds for all iterations, since both variables are odd just before the *sub-step*. Expanding for $i < t$:

$$u_t - v_t = \frac{\frac{p-1}{2^{Z_1}} - e}{\frac{2^{Z_2}}{2^{Z_3}} - e} - e \equiv 0 \pmod 2$$

thus solving for p yields (2) for bit recovery, where n from (1) is the number of recovered bits from p .

$$p \equiv e(2^{Z_1} + 2^{Z_1+Z_2} + \dots + 2^n) + 1 \pmod{2^{n+1}} \tag{2}$$

In summary, for RSA-2048 n is roughly $1024 - 17 = 1007$ bits. However, due to Coppersmith [Cop96] an adversary only needs $\lg(N)/4 = 512$ bits of one prime to factor an RSA-2048 N , depicted in Figure 3. For either NIST compliant value of e , the number of bits recovered is far beyond the Coppersmith bound.

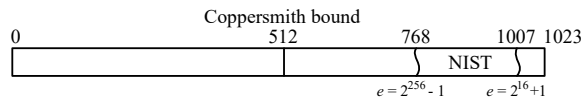


Figure 3: RSA-2048 bit-recovery bounds of n for different e .

We now have our requirements for a successful attack—the adversary must obtain (at least) the first t_c noise-free Z_i to factor N :

$$n = \sum_{i=1}^{t_c} Z_i \geq \lg(N)/4$$

where t_c is the minimum iteration for which n reaches the Coppersmith bound.

3.3 A Single Trace Attack: Roadmap

Previously in this section, we uncovered three side-channel insecure code paths traversed during RSA key generation. Subsequently focusing on the `BN_gcd` code path, we then gave a theoretical analysis on the GCD algorithm as implemented in OpenSSL to describe what kind of side-channel information we can extract, and rough bounds for how much (noise-free) information we need to leverage that to recover the private key by factoring N . The remainder of this paper is dedicated to describing the methods, techniques and problems faced when trying to recover the necessary information from the side-channel leakage in order to achieve full private RSA key recovery from a single trace. The roadmap for our end-to-end attack is as follows: (1) We capture cache-timing traces from `BN_gcd` executions during RSA key generation, then—leveraging signal processing techniques—extract the portions corresponding to $p-1$ and $q-1$, apply digital filters and extract their corresponding (noisy) LS-sequences (Section 3.4); (2) Building upon previous work, we design and

implement an error correction algorithm for these sequences—leveraging number theoretic constraints imposed by RSA—to extract partial bits of one factor of N (Section 4); (3) Said algorithm yields an ordered list of candidates for partial factors; we then derive lattice parameters for factoring with Coppersmith’s method, and create lattice instances with said candidates, iteratively executing them until the result yields complete factorization of N (Section 5).

Attack setup. Our attack setup consists of an Intel Core i5-2400 Sandy Bridge 3.10 GHz (32 nm) with 8 GB of memory running 64-bit Ubuntu 16.04 LTS “Xenial” with hardware prefetching and Turbo Boost disabled. All the cores share a 12-way 6 MB unified LLC. The system does not feature HyperThreading.

We tested our attack against OpenSSL 1.0.2k—the latest release and LTS version at the time of our experiments—with debugging symbols on the executable. We use the debugging symbols to map source code to memory addresses, allowing us to find the “hot” memory addresses for the degrading attack and probing accurately the sequence of operations mentioned previously. Note, however, debugging symbols are not a requirement for the attack as this information can be obtained through reverse engineering. We passed the `shared` configuration option to compile OpenSSL as a shared object.

3.4 From Timings to Sequence of Operations

The GCD algorithm implemented in OpenSSL is highly dependent on its inputs during execution, thus we use the well-known FLUSH+RELOAD technique to probe cache lines in code routines `BN_rshift1` and `BN_sub`. By probing these two routines, we are able to distinguish two branches executed by the GCD algorithm, namely right-shifts and subtractions. Unfortunately this is not enough to recover meaningful data, since we need to know the exact Z_i values (i.e. number of right-shifts executed between subtractions) in order to identify bits. Due to tight loop execution during these operations, our probe misses some of the accesses.

To that end, to get better resolution we pair the FLUSH+RELOAD technique with the performance degradation attack [All+16] which targets different cache lines in the same previous routines to slow down the execution. Moreover, we apply the profiling approach [PGB17] to easily identify the best memory addresses to probe and degrade. Adapted to our strategy, this provides a good starting point to recover a sequence of operations.

Granularity. Due to the nature of the GCD algorithm, the granularity of the `BN_rshift1` and `BN_sub` operations captured in a trace vary throughout the execution of the algorithm. As the input values to the function are processed, their bit length decreases and this behavior is reflected in the trace. Typically, when a GCD operation begins, a single right-shift operation spans over several data points (i.e. cache-hits) in the trace, while at the end of the execution the same right-shift operation registers fewer points, sometimes even only one point. This represents a challenge later in our attack during the horizontal analysis, when we need to extract the sequence of right-shift and subtraction operations from the p and q traces (i.e. Z_i), since operations can be easily misclassified due to high data point variation for each operation within a single GCD execution.

Traces. The contents of a typical trace (see Figure 4, top), from high to low abundance, is roughly: (1) noise and/or `BN_mod_exp_mont` executions during primality testing, not targeted by our probes but nonetheless consuming CPU cycles; (2) short `BN_mod_inverse` executions setting up Montgomery arithmetic, with the same underlying right-shifts and subtracts as `BN_gcd` that our probes target; (3) longer `BN_gcd` executions for testing

coprimality. We are only interested in the latter, yet manually isolating the part of the trace that corresponds to the real `BN_gcd` executions for $p - 1$ and $q - 1$ is time consuming and not feasible at a large scale.

Unlike other side-channel scenarios where attackers have oracle access to trigger the cryptographic operation, i.e. they can start side-channel signal acquisition (e.g. launch the spy) roughly at the same time as they start their query (e.g. initiate a protocol run or call an API), our case is very different. In our case, we have no control over when key generation takes place, leaving us with extremely long traces, and the challenging task of extracting a minuscule partial trace from abundant data—looking for a needle in a haystack. We turn to signal processing-based power analysis techniques to tackle this issue.

Templates. Inspired by SPA and template attacks [CRR02] and related (yet less statistical) cache-based techniques [BH09, GSM15], we create a template by manually adjusting `FLUSH+RELOAD` parameters in our spy process, then inspecting and trimming a trace that looks visually correct, i.e. no clear preemptions of the spy or the victim process.

Correlation and matching. Motivated by statistical methods such as Horizontal Correlation Analysis [Cla+10], we leverage the Pearson correlation coefficient to find the best match for these templates within full traces, automating the process. Once we create our template and acquire our trace, we compute the moving Pearson correlation between the template and full trace, then extract the peaks, i.e. discrete indices that exhibit highest correlation between the template and the side-channel data. This automates the GCD identification step, allowing us to extract the sequences for p and q .

Horizontal analysis. Finally, to overcome the granularity issues previously mentioned, we use a dynamic horizontal analysis approach to recover the sequence of operations executed by the GCD algorithm. Our dynamic horizontal analysis works as follows: first we take as input the processed trace which has been aligned to the first subtraction operation and trimmed to a specific length, avoiding noise as much as possible. Then, we take small chunks of the trace containing n windows of subtraction and right-shift operations, recall that each subtraction is followed by at least one right-shift operation. After that, we compute the Euclidean distance between the subtraction operations in those n windows. We sort the resulting distances from shortest to longest and then we consider the shortest distance as a single right-shift operation (i.e. $Z_i = 1$), thus using it as basis to determine the number of right-shift operations in the rest of windows. After calculating all the right-shift counts in those n windows, we proceed to the next chunk and repeat the process. We continue until all the operations in the trace have been calculated, resulting in a tentative and noisy *LS sequence* of operations (i.e. Z_i candidates).

Figure 4 illustrates our method in action using a real trace and template. The top most plot is the filtered partial trace, containing two GCD runs for p and q —note the narrower peaks corresponding to executions of `BN_mod_inverse` during the primality test, also leaking secret information on the prime values due to yet another flag not set (see Section 3.1). The third plot is the moving Pearson correlation coefficient between the template (second plot) and the trace (first plot), with two extremely distinguishable peaks that identify the locations of the two GCD operations. The second plot aligns the template at maximum correlation for visualization purposes; and finally, the bottom plot shows a closer view of the sequence of operations performed during a single GCD operation. As seen, our technique is remarkably effective.

As mentioned previously, the accuracy of the operations in the trace decreases dramatically by the end of the GCD execution. The trace contains errors introduced by several factors such as victim preemptions, spy preemptions, as well as noise created by other

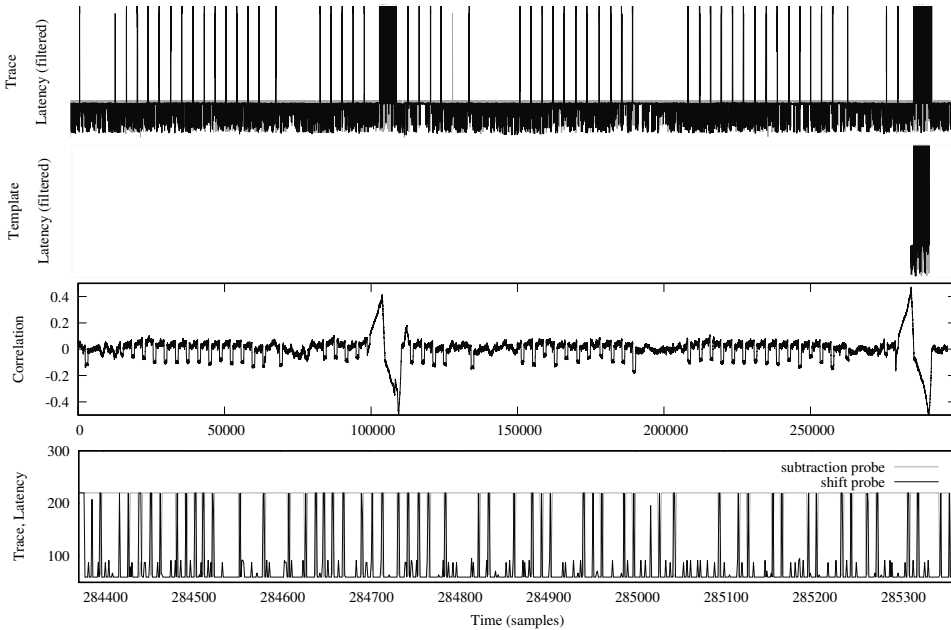


Figure 4: Visualization of the moving Pearson correlation in action. From top to bottom: filtered trace, aligned template trace, Pearson correlation, and raw trace (zoomed).

processes and microarchitecture components. To overcome this, Section 4 details an error correction algorithm developed to find potential correct LS sequence candidates that later are converted to bits and used as input values to perform the lattice attack explained in Section 5.

4 Error Correction in noisy LS Sequences

In order to design an algorithm for correcting the errors in an LS sequence, we characterize the nature of them. As discussed in Section 3.3, cache-timing attacks like FLUSH+RELOAD provide noisy data due to variances in the execution environment, interruptions, preemptions, task scheduling, etc. Therefore LS sequence extraction from the raw traces is not error free and contains errors with overwhelming probability.

After analyzing many of the traces with known inputs, we identified the following classes of errors: (1) Wrong number of ‘L’ symbols between two ‘S’ symbols (due to Z_i estimation error). (2) Missing ‘S’ symbols (less frequent). (3) Extra ‘S’ symbols (much less frequent). (4) Victim preemption: observed as a small gap (i.e. window of cache misses) in the middle of operations but fixable by removing this window during trace processing. (5) Spy preemptions: observed as a *hole* in the trace exhibited by the timing information from the FLUSH+RELOAD attack. They are detectable but unfortunately operations during the preemption window are completely lost.

4.1 Leakage Data: Error Modeling

From the FLUSH+RELOAD attack, we obtain pairs (Z_i^p, Z_i^q) for p and q respectively. With this information, we obtain two recovery equations according to (2), where wlog. n_1 and n_2 are greater than some n . Thus, we have sufficient (noisy) Z_i for both primes to recover

their first n bits.

$$\begin{aligned} p &\equiv ex_1 + 1 \pmod{2^{n_1+1}}, & x_1 &= (2^{Z_1^p} + 2^{Z_1^p+Z_2^p} + \dots + 2^{n_1}) \\ q &\equiv ex_2 + 1 \pmod{2^{n_2+1}}, & x_2 &= (2^{Z_1^q} + 2^{Z_1^q+Z_2^q} + \dots + 2^{n_2}) \end{aligned} \quad (3)$$

Therefore, the Z_i for a prime p (resp. q) defines set bits in the binary representation of x_1 (resp. x_2). Hence for every value of n we can check if (4) holds.

$$N \equiv (ex_1 + 1)(ex_2 + 1) \pmod{2^n} \quad (4)$$

This relation is very similar to that employed by Heninger and Shacham [HS09] for the $m = 2$ case. In their work, errors are in the bit positions of p and q directly while in our case they are instead in the bit positions of a divisor of $p - 1$ and $q - 1$. Irrespective of this difference, it leads to a similar error correction algorithm constructed for correcting some errors.

Regarding existing noise models, our data might have some form of erasures due to spy preemption. However, while *erasure model* considers missing data, at the same time it requires knowing where the missing bits are and they should be distributed at random. Spy preemption fulfills the first condition but it implies a consecutive missing bits/symbols instead of random. On the other hand this model does not consider insertions and deletions.

At the same time, the bit-flip model by Henecka, May, and Meurer [HMM10] does not apply to our case as the largest error source is due to insertions and deletions of zeros between consecutive ones in the binary representation of x_1 and x_2 . It is worth noting that these insertions and deletions generate some bit-flips on p and q . However, we verified that even a small insertion/deletion rate of 0.08 implies a bit-flip rate on p and q of about 0.5 due to an avalanche effect. Hence, obtained p and q from their noisy Z_i look like random data.

In this regard, the solution to bit-flip noise model by Henecka, May, and Meurer [HMM10] is tightly coupled to the Hamming distance as a metric for filtering out wrong solutions. The algorithm proposed in [HMM10] could be an option to address our noise model, but requires selecting a proper distance metric. We identified the weighted Levenshtein distance as a possible good distance candidate, as it allows assigning different weights for each operations per symbol. Then, it is possible to assign operation per symbol weights to fit a specific noise model. While this is a plausible approach, changing the distance metric inhibits us from using the theoretical and experimental results from [HMM10] to get an estimate on expected success rates. For this reason, we defer this approach to future work and implement an error correction algorithm that does not depend on an specific distance metric.

Inci et al. [Inc+16] use another interesting approach to correct errors in RSA keys. They developed an algorithm that fixes some errors in a noisy version of dp and dq (i.e. $m = 2$ case). Their noise model has some similarities with ours, however they considered that dp is almost error-free, while in our case we cannot make this assumption. In addition, not much is said about the error rate supported by this algorithm nor its success probability under any error rate.

However, despite that *erasure* and *bit-flip* noise models do not apply directly to our scenario, we borrow the core ideas from Heninger and Shacham [HS09] and Henecka, May, and Meurer [HMM10] to build an error correction algorithm that handles the most common errors in our noise model: *insertions and deletions of zeros in the binary representation of x_1 and x_2* .

This relaxed noise model selection is not arbitrary. Our intuition is to use this starting approach as a building block for fixing other error sources. Also, it allows getting a first lower bound on the success rate of the attack and then scaling it (if needed) to support other errors (for example errors in ‘S’). In addition, avoiding some specific error handling

like spy preemption allows using this correction algorithm in other scenarios for correcting these types of errors in other elements of **sk**.

4.2 Error Correction Algorithm

Our algorithm follows the Expand-and-Prune approach [HMM10] as shown in Algorithm 3. The algorithm iterates over all bits from $i \leq n$. It processes a set of candidates \mathcal{C}_i at every iteration i , starting from a single candidate with the noisy x_1, x_2 . At each iteration, each candidate resulting from the previous iteration expands to several candidates. Then, to avoid candidate space explosion, it prunes these candidates based on rules controlled by the algorithm parameters. Therefore, the configuration parameters of the **Prune** procedure manage the search space growth rate while aiming to increase the probability that the correct solution survives through iterations.

Algorithm 3: Error correction algorithm

Input: N, e, Z_i^p, Z_i^q, n , config parameters

Result: Set of n -bit candidates for x_1 and x_2

```

1 begin
2    $x_1, x_2 \leftarrow$  Using  $Z_i^p, Z_i^q$  according to (3)
3    $\mathcal{C}_0 \leftarrow \{(x_1, x_2)\}$ 
4   for  $i = 1$  to  $n - 1$  do
5      $\mathcal{E}_i \leftarrow \emptyset$ 
6     foreach  $c \in \mathcal{C}_{i-1}$  do
7        $\mathcal{E}_i = \mathcal{E}_i \cup \mathbf{Expand}(c, i)$ 
8      $\mathcal{C}_i \leftarrow \mathbf{Prune}(\mathcal{E}_i, \mathit{params})$ 
9   return  $\mathcal{C}_{n-1}$ 

```

Expand. To expand a given candidate c at some bit i (i.e. $c[i]$), consider the selected bit as a branch in a tree. If we construct a search tree, then the possibilities for the bits at any level give rise to new branches in said tree. The tree at any level i contains all the partial solutions x_1, x_2 up to the i -th LSB. At any level i there are at most six possible branching candidates that can fulfill (4), listed in Table 1.

Table 1: Possible branching candidates.

Possibilities	Description
(x_1, x_2)	(4) holds without changes to x_1 or x_2
$(x_1 - 0, x_2)$	Remove a zero at position i from x_1 ; no changes to x_2
$(x_1 + 0, x_2)$	Insert a zero at position i in x_1 ; no changes to x_2
$(x_1, x_2 - 0)$	Remove a zero at position i from x_2 ; no changes to x_1
$(x_1, x_2 + 0)$	Insert a zero at position i in x_2 ; no changes to x_1
mult	A combination of changes in both x_1 and x_2

One interesting feature of this algorithm is that even if (4) holds without changing x_1 and x_2 , it still tests the remaining possibilities, including errors that occur at the same index i in x_1 and x_2 —a situation not detected using (4).

Figure 5 illustrates the expansion and pruning procedure for three consecutive iterations of a candidate c starting at bit i showing the possible candidates. Here we used \emptyset to represent the candidates that did not generate valid solutions. Note how in the first

expansion, four possible branching candidates fulfilled (4), however, some of them did not generate viable solutions afterwards or were pruned.

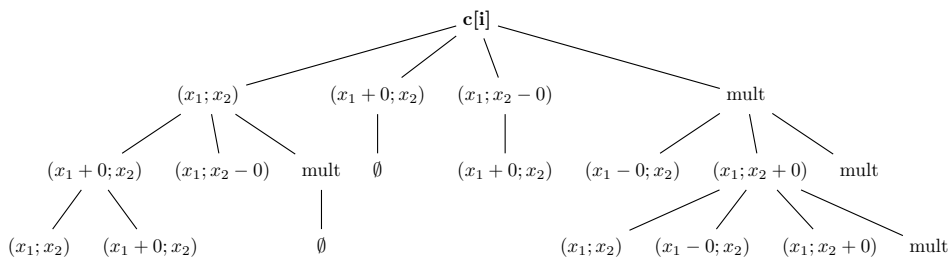


Figure 5: Expansion process for candidate $c[i]$. Pruned solutions are in red.

The candidate pool grows exponentially. We now turn to restricting the number of potential candidates (i.e. the partial solutions) at any level so that finding the correct one is possible through exhaustive search among all solutions within a certain feasible limit, examining situations that control the branching behavior of the tree.

Prune. The pruning process applies a set of filters to the expanded candidates \mathcal{E}_i . The filter spectrum is very wide and selecting the best for a given noise model is a challenging task. In this regard, contrary to [HMM10] we implemented a set of filters in a combined approach—the most novel feature of this algorithm.

Candidates in \mathcal{E}_i are grouped based on the total amount of *changes* (i.e. potential errors) made in both x_1 and x_2 until iteration i (see Table 1 for the list of possible changes), allowing us to sort the potential candidates based on this parameter.

The most important filters relate to the number of groups (g) to keep and the maximum number of candidates in each group (G). Denote e_{\min} as the minimum number of changes made in all candidates in \mathcal{E}_i , e_x the number of changes in x_1 or x_2 , and e_{mult} the number of multiple changes up to iteration i . We define the filters as follows: (1) *Max changes over minimum*: Keep the candidate if $e_{x_1} + e_{x_2} + e_{\text{mult}} \leq e_{\min} + g$ holds. (2) *Max candidates*: Sort each group based on $(e_{x_1} + e_{x_2} + e_{\text{mult}}, e_{\text{mult}})$, and keep the first G candidates. (3) *Consecutive changes*: Discard sequences having more than a fixed number of consecutive changes, following the heuristic that higher change densities should be less probable than lower ones—therefore it should be more likely that the correct solution has a smaller change density. (4) *Max changes (hard threshold)*: Candidates that exceed a maximum number of changes threshold are discarded ($e_{x_1} + e_{x_2} + e_{\text{mult}} \geq e_{\text{th}}$), helping to detect very unlikely solutions—those with an extremely high number of changes.

We selected the parameters of these filters to keep the probability of pruning the correct solution low, while at the same time keeping the computational requirements affordable for the attacker. In general terms, the adversary can profile the target environment—generating a set of known RSA keys and collecting information about the number of errors, their distribution, etc. for selecting these parameters. We followed this approach for 100 independent RSA-2048 keys and tuned these parameters for our attack environment. We analyzed the number of groups between 5 and 10 and the number of candidates in each group between 5000 and 15000. We set the number of consecutive changes filter to three and based on the observed error rates it is very unlikely that a candidate has more than 150 errors at any iteration, therefore we set the hard threshold to this value.

After this characterization, we observed that 37 traces of 100 fit our reduced noise model: only errors in the number of zeros between ones of x_1 and x_2 . We recovered at least 512 bits from 30 of the test traces, therefore our correction algorithm worked for

80% of the traces it can handle (reduced noise model). However, as we used these same traces to tune the filter parameters, this value should not be taken as a measure of the success rate of our algorithm as it is biased. Section 5.1 shows the experimental results for 10K independent traces, and from this large set we extracted the estimate that our error correction algorithm recovers at least 512 bits for 73% of the traces that met our reduced noise model (see Section 5.1 for more details).

It is worth noting that these success rates and handled error rates are incompatible with other works (e.g. [HMM10]) due to different noise models with respect to previous work. However, we point out that the multiple filter approach that we follow in our algorithm could be an interesting option for addressing other noise models, where initial experiments suggest that some previous works would be improved.

Candidates enumeration. One important feature of our algorithm is the way it enumerates candidates for checking factors of N (i.e. next stage of our end-to-end attack). As described above, each \mathcal{C}_i consists of a fixed number of groups g with at most G possible candidates in each group. The naïve approach would be to search all possible candidates in each group until finding the solution. However, based on empirical data, we found that the real solution tends towards the first position of a group. In this case, it makes more sense to consume the candidates using a round robin approach, giving a higher priority to the highest ranked candidates in each group.

5 Factoring with Partial Information: Endgame

In his groundbreaking work, Coppersmith [Cop96] proposed a method to find small solutions of univariate modular equations with modulus having unknown factorization. This result finds many uses in cryptography (mainly in cryptanalysis) as several times in real-world applications an attacker has access to an oracle that gives partial information of a secret and the problem of recovering the remaining part is modeled as a univariate modular equation.

Side-channel attacks play very nicely the oracle role as they often only reveal a (minority) fraction of secret bits. Also, as in our scenario even if it is theoretically possible to fully recover the primes from side-channel traces, it is preferable to only partially recover them to reduce noise influence.

Coppersmith’s result has several implications on RSA security. For excellent surveys about its impact, we refer readers to [NV10, Hin10]. One of these applications is factoring N when half the bits of one prime are known—either the most or the least significant half. The lattice-based solution to this application has been extensively covered in literature [NV10, Hin10, Nem+17]. However, for the sake of completeness Appendix A contains a full description of this procedure and its parametrization in terms of how many bits of p are needed to factor an RSA-2048 modulus. To summarize, we estimate that 522 bits of p are sufficient to compromise RSA-2048 with high probability.

5.1 Results: End-to-End Attack

To consolidate the attack and validate the successfulness of our techniques and the attack overall, we used the following setup: a core executing 10K RSA key generations using OpenSSL `genpkey` command, while degrading the performance in two additional cores and finally executing the spy process on the last core, thus collecting 10K traces. Once we collected the traces, and using templates and the Pearson correlation coefficient, we extracted and aligned the GCD operations for $p - 1$ and $q - 1$. Out of those 10K traces, 566 traces were useless due to two main reasons: (1) key generation execution took more time than expected due to failed primality tests; or (2) spy/victim were preempted for a

long period of time; thus the spy missed capturing one or both of the GCD operations in any of those cases. We were able to perform horizontal analysis on the remaining 9434 traces to extract a tentative LS sequence of operations, i.e. Z_i values, aiming to recover a minimum of 522 bits per prime value.

We then offloaded the data for these 9434 trials to a cluster for analysis, containing roughly 1500 nodes mixed between Intel E5-2680 (Sandy Bridge), E5-2670 (Sandy Bridge), and E5-2630 (Haswell) cores. In all the stages that follow, we limited per-job execution times to 4 CPU hours. Table 3 shows computation effort statistics, where a single attack run takes less than 2 CPU hours on average, making it very affordable in practice.

The LS sequences contain errors that the subsequent lattice attack cannot tolerate—error correction is required to recover sufficient bits. Our lattice attack can factor RSA-2048 knowing 522 bits of a prime, hence we configured our error correction algorithm to recover the same number of bits. Following the algorithm tuning process (see Section 4.2) we selected the set of pruning filter parameters shown in Table 2.

Table 2: Parameters for error correction algorithm.

Parameter	Value
Max changes over minimum	10
Max candidates per group	15000
Consecutive changes	3
Max changes (hard threshold)	150

On the cluster, we launched the algorithm for the 9434 traces that contained tentative LS sequences using Table 2 parameters. For each of the 9434 traces, we also analyzed the ground truth correct sequence to collect data about the probability of recovering a given amount of bits up to 552. Figure 6 (Left) shows the resulting survival probability curve.

This survival curve gives an idea about the error correcting algorithm behavior for our large data set. One of the most relevant results is the probability of recovering at least 522 bits: 27.89% (2632 of 9434). At the same time, the probability is quite close to that of 512 bits: 29.17%. This small difference confirms the estimation made during the lattice attack parameter optimization: correcting errors up to 522 bits is not significantly more challenging than the 512 bits case. Therefore in our setting, improving lattice attack parameters to Coppersmith’s bound (512 bits) will not significantly increase the error correction success rate.

Of interest, Figure 6 (Left) shows an abrupt probability drop at the start of the curve. We analyzed it closely and confirmed that roughly 30% of the traces have a spy preemption just at the beginning, resulting in incomplete traces. It seems there is a bias in our environment that increases the probability of spy preemption at the start of a GCD execution, yet not in the middle. We are investigating the reasons behind this bias, but the fact that the Figure 6 (Left) curve does not contain another abrupt drop confirms our bias hypothesis.

After this analysis, our error correction algorithm was able to recover 522 bits for 2632 traces. One interesting metric is the number of errors considering both LS sequences (i.e. p and q) that it handled per key, and Figure 6 (Middle) shows the boxplots of the aggregate number of errors in both LS sequences for recovering various bit quantities. The data suggest the number of errors successfully handled is diverse—for example, recovering 522 bits (rightmost boxplot), this metric ranges from 24 to 108. It implies that our error correction algorithm with Table 2 parameters recovered 522 bits in 2632 traces with error rates that range from $24/(2 \cdot 522) = 0.02$ to $108/(2 \cdot 522) = 0.10$, since we require 522 bits of each prime (i.e. p and q).

Of these 9434 instances, we successfully recovered 2285 private keys after 12875 lattice trials; Figure 6 (Right) “Computed” depicts these data points. This represents just over

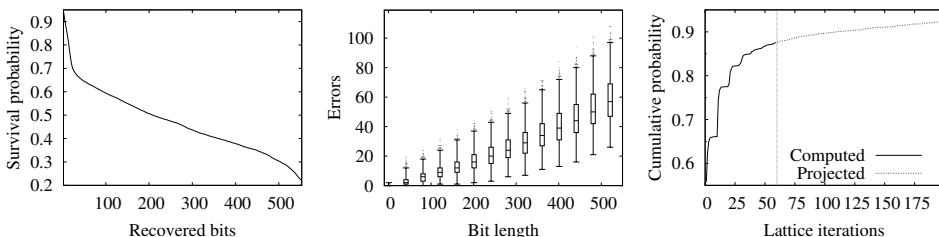


Figure 6: Left: bit recovery survival plot. Middle: number of errors successfully handled by our error correction algorithm. Right: Lattice iterations for successful instances.

Table 3: Cluster computation effort in CPU minutes across phases and CPUs.

CPU	Error correction phase			Lattice phase		
	Median	Mean	Dev.	Median	Mean	Dev.
E5-2670	103.0	103.7	41.2	6.2	6.7	1.7
E5-2680	92.3	90.4	36.1	4.3	4.3	0.3
E5-2630	100.0	95.0	40.1	6.8	7.3	1.8

45 days of CPU time. The remaining 7149 instances break down as follows, which we analyzed using the ground truth private keys. For 347 instances, the partial prime factor remained amongst the candidates, yet had a poor ranking, hence the number of lattice iterations needed exceeding our fixed allowed time (4 h); Figure 6 (Right) “Projected” depicts these data points. In these cases, we verified the lattice output at that future iteration indeed yields the intended factor. The remaining 6802 instances failed to retain the correct candidate. To find a solution, the error correction algorithm with round-robin enumeration achieved an impressive median of one lattice iteration for successful instances.

End-to-end attack summary. From a large data set of 9434 independent traces, our end-to-end attack achieves a success rate of 27.89%. The error correction algorithm was able to fix/recover 522 bits for 2632 traces. At the same time, the lattice attack succeeded for all these 2632 traces, showing the robustness of our lattice attack parameters for 522 bits (Table 4).

6 Conclusion

In this work, we proposed a methodology to analyze cryptographic software for traversal of known side-channel insecure code paths. Applying our methodology to RSA key generation in OpenSSL uncovered three new vulnerabilities, one of which we designed an end-to-end cache-timing attack around, leading to key recovery with good probability and modest computational effort. The attack chain consisted of (1) gathering timings with a combination of FLUSH+RELOAD and performance degradation; (2) locating the trace segments of interest (two specific GCD executions) within abundant data; (3) transforming these traces into noisy LS-sequences representing GCD algorithm state; (4) executing our error correction algorithm, resulting in a ranked list of partial prime factor candidates; (5) formulating lattice problems for these candidates that recover the unknown portion; (6) testing if the result yields a prime factor of the RSA modulus N , hence the private key. Executing 10K trials and moving the analysis to a cluster, we achieved roughly a 27% success rate for full key recovery. We close with lessons learned from our work.

Lesson 1: Secure by default. Similar to two recent works [PGBY16, PGB17], two of the vulnerabilities our methodology uncovered are due to insecure default behavior—failure to set a particular flag that, by early exit, diverts the code through algorithms with SCA mitigations. Had the logic been inverted, taking the secure paths by default would have prevented these vulnerabilities. For OpenSSL, these new vulnerabilities continue an unfortunate trend of insecure by default failures that went undetected during unit testing.

Lesson 2: Knowledge transfer. Our end-to-end attack exploits only one of the three vulnerabilities our methodology uncovered. The function we targeted is oblivious to the constant-time flag, hence having it set or clear has no effect on our attack. Our root cause analysis (Section 3.1) suggests that the mitigations mainlined as a result of pioneering academic work [AGS07] failed to consider RSA key generation as a whole, and the similarities between GCD computation (which we exploited) and modular inversion with respect to branching behavior went unnoticed when these mitigations were independently developed. This disconnect demonstrates the critical importance of engineers working side-by-side with cryptographers to ensure that academic results reach their intended impact on real-world products.

Affected versions and responsible disclosure. The issues presented in this paper affected OpenSSL versions 1.1.0-1.1.0h and 1.0.2-1.0.2o. Following responsible disclosure procedures, we reported these issues to OpenSSL and provided fixes, subsequently merged into the 1.0.2 and 1.1.0 branches after the embargo lifted. OpenSSL 1.1.1 did not exist at that time, thus it was never impacted. OpenSSL assigned CVE-2018-0737 based on our work.

Acknowledgments

We would like to thank to Matúš Nemeč for sharing his results. We thank Tampere Center for Scientific Computing (TCSC) for generously granting us access to computing cluster resources. Supported in part by Academy of Finland grant 303814. The second author was supported in part by a Nokia Foundation Scholarship and by the Pekka Ahonen Fund through the Industrial Research Fund of Tampere University of Technology. This article is based in part upon work from COST Action IC1403 CRYPTACUS, supported by COST (European Cooperation in Science and Technology). This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No 804476).

References

- [ACSS17] Alejandro Cabrera Aldaya, Alejandro J. Cabrera Sarmiento, and Santiago Sánchez-Solano. “SPA vulnerabilities of the binary extended Euclidean algorithm”. In: *J. Cryptographic Engineering* 7.4 (2017), pp. 273–285. DOI: 10.1007/s13389-016-0135-4. URL: <https://doi.org/10.1007/s13389-016-0135-4>.
- [AGS07] Onur Aciğmez, Shay Gueron, and Jean-Pierre Seifert. “New Branch Prediction Vulnerabilities in OpenSSL and Necessary Software Countermeasures”. In: *Cryptography and Coding, 11th IMA International Conference, Cirencester, UK, December 18-20, 2007, Proceedings*. Ed. by Steven D. Galbraith. Vol. 4887. Lecture Notes in Computer Science. Springer, 2007, pp. 185–203. DOI: 10.1007/978-3-540-77272-9_12. URL: https://doi.org/10.1007/978-3-540-77272-9_12.

- [Ald+17] Alejandro Cabrera Aldaya et al. “Side-channel analysis of the modular inversion step in the RSA key generation algorithm”. In: *I. J. Circuit Theory and Applications* 45.2 (2017), pp. 199–213. DOI: 10.1002/cta.2283. URL: <https://doi.org/10.1002/cta.2283>.
- [All+16] Thomas Allan et al. “Amplifying side channels through performance degradation”. In: *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016*. Ed. by Stephen Schwab, William K. Robertson, and Davide Balzarotti. ACM, 2016, pp. 422–435. DOI: 10.1145/2991079.2991084. URL: <http://doi.acm.org/10.1145/2991079.2991084>.
- [Ant+17] Timos Antonopoulos et al. “Decomposition instead of self-composition for proving the absence of timing channels”. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. Ed. by Albert Cohen and Martin T. Vechev. ACM, 2017, pp. 362–375. DOI: 10.1145/3062341.3062378. URL: <https://doi.org/10.1145/3062341.3062378>.
- [AT07] Sarang Aravamuthan and Viswanatha Rao Thumparthy. “A Parallelization of ECDSA Resistant to Simple Power Analysis Attacks”. In: *Proceedings of the Second International Conference on COMMunication System softWare and MiddlewaRE (COMSWARE 2007), January 7-12, 2007, Bangalore, India*. Ed. by Sanjoy Paul, Henning Schulzrinne, and G. Venkatesh. IEEE, 2007. DOI: 10.1109/COMSWA.2007.382592. URL: <https://doi.org/10.1109/COMSWA.2007.382592>.
- [Bau+14] Aurélie Bauer et al. “Side-Channel Attack against RSA Key Generation Algorithms”. In: *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*. Ed. by Lejla Batina and Matthew Robshaw. Vol. 8731. Lecture Notes in Computer Science. Springer, 2014, pp. 223–241. DOI: 10.1007/978-3-662-44709-3_13. URL: https://doi.org/10.1007/978-3-662-44709-3_13.
- [Ber+13] Daniel J. Bernstein et al. “Factoring RSA Keys from Certified Smart Cards: Coppersmith in the Wild”. In: *Advances in Cryptology - ASIACRYPT 2013 - 19th International Conference on the Theory and Application of Cryptology and Information Security, Bengaluru, India, December 1-5, 2013, Proceedings, Part II*. Ed. by Kazue Sako and Palash Sarkar. Vol. 8270. Lecture Notes in Computer Science. Springer, 2013, pp. 341–360. DOI: 10.1007/978-3-642-42045-0_18. URL: https://doi.org/10.1007/978-3-642-42045-0_18.
- [Ber+17] Daniel J. Bernstein et al. “Sliding Right into Disaster: Left-to-Right Sliding Windows Leak”. In: *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*. Ed. by Wieland Fischer and Naofumi Homma. Vol. 10529. Lecture Notes in Computer Science. Springer, 2017, pp. 555–576. DOI: 10.1007/978-3-319-66787-4_27. URL: https://doi.org/10.1007/978-3-319-66787-4_27.
- [BH09] Billy Bob Brumley and Risto M. Hakala. “Cache-Timing Template Attacks”. In: *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings*. Ed. by Mitsuru Matsui. Vol. 5912. Lecture Notes in Computer Science. Springer, 2009, pp. 667–684. DOI: 10.1007/978-3-642-10366-7_39. URL: https://doi.org/10.1007/978-3-642-10366-7_39.

- [BM03] Johannes Blömer and Alexander May. “New Partial Key Exposure Attacks on RSA”. In: *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*. Ed. by Dan Boneh. Vol. 2729. Lecture Notes in Computer Science. Springer, 2003, pp. 27–43. DOI: 10.1007/978-3-540-45146-4_2. URL: https://doi.org/10.1007/978-3-540-45146-4_2.
- [Bri+06] Ernie Brickell et al. “Software mitigations to hedge AES against cache-based software side channel vulnerabilities”. In: *IACR Cryptology ePrint Archive 2006.52* (2006). URL: <http://eprint.iacr.org/2006/052>.
- [Cla+10] Christophe Clavier et al. “Horizontal Correlation Analysis on Exponentiation”. In: *Information and Communications Security - 12th International Conference, ICICS 2010, Barcelona, Spain, December 15-17, 2010. Proceedings*. Ed. by Miguel Soriano, Sihang Qing, and Javier López. Vol. 6476. Lecture Notes in Computer Science. Springer, 2010, pp. 46–61. DOI: 10.1007/978-3-642-17650-0_5. URL: https://doi.org/10.1007/978-3-642-17650-0_5.
- [Cop96] Don Coppersmith. “Finding a Small Root of a Univariate Modular Equation”. In: *Advances in Cryptology - EUROCRYPT ’96, International Conference on the Theory and Application of Cryptographic Techniques, Saragossa, Spain, May 12-16, 1996, Proceeding*. Ed. by Ueli M. Maurer. Vol. 1070. Lecture Notes in Computer Science. Springer, 1996, pp. 155–165. DOI: 10.1007/3-540-68339-9_14. URL: https://doi.org/10.1007/3-540-68339-9_14.
- [Cop97] Don Coppersmith. “Small Solutions to Polynomial Equations, and Low Exponent RSA Vulnerabilities”. In: *J. Cryptology* 10.4 (1997), pp. 233–260. DOI: 10.1007/s001459900030. URL: <https://doi.org/10.1007/s001459900030>.
- [CRR02] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. “Template Attacks”. In: *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*. Ed. by Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar. Vol. 2523. Lecture Notes in Computer Science. Springer, 2002, pp. 13–28. DOI: 10.1007/3-540-36400-5_3. URL: https://doi.org/10.1007/3-540-36400-5_3.
- [DK17] Goran Doychev and Boris Köpf. “Rigorous analysis of software countermeasures against cache attacks”. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. Ed. by Albert Cohen and Martin T. Vechev. ACM, 2017, pp. 406–421. DOI: 10.1145/3062341.3062388. URL: <https://doi.org/10.1145/3062341.3062388>.
- [Doy+15] Goran Doychev et al. “CacheAudit: A Tool for the Static Analysis of Cache Side Channels”. In: *ACM Trans. Inf. Syst. Secur.* 18.1 (2015), 4:1–4:32. DOI: 10.1145/2756550. URL: <https://doi.org/10.1145/2756550>.
- [FGS09] Thomas Finke, Max Gebhardt, and Werner Schindler. “A New Side-Channel Attack on RSA Prime Generation”. In: *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*. Ed. by Christophe Clavier and Kris Gaj. Vol. 5747. Lecture Notes in Computer Science. Springer, 2009, pp. 141–155. DOI: 10.1007/978-3-642-04138-9_11. URL: https://doi.org/10.1007/978-3-642-04138-9_11.
- [Fip] *Digital Signature Standard (DSS)*. FIPS PUB 186-4. National Institute of Standards and Technology, 2013. DOI: 10.6028/NIST.FIPS.186-4. URL: <https://doi.org/10.6028/NIST.FIPS.186-4>.

- [Gri+19] Iaroslav Gridin et al. “Triggerflow: Regression Testing by Advanced Execution Path Inspection”. In: *Detection of Intrusions and Malware, and Vulnerability Assessment - 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19-20, 2019, Proceedings*. Ed. by Roberto Perdisci et al. Vol. 11543. Lecture Notes in Computer Science. Springer, 2019, pp. 330–350. DOI: 10.1007/978-3-030-22038-9_16. URL: https://doi.org/10.1007/978-3-030-22038-9_16.
- [GSM15] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. “Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches”. In: *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*. Ed. by Jaeyeon Jung and Thorsten Holz. USENIX Association, 2015, pp. 897–912. URL: <https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/gruss>.
- [GVY17] Daniel Genkin, Luke Valenta, and Yuval Yarom. “May the Fourth Be With You: A Microarchitectural Side Channel Attack on Several Real-World Applications of Curve25519”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. Ed. by Bhavani M. Thuraisingham et al. ACM, 2017, pp. 845–858. DOI: 10.1145/3133956.3134029. URL: <http://doi.acm.org/10.1145/3133956.3134029>.
- [Hen+12] Nadia Heninger et al. “Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices”. In: *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*. Ed. by Tadayoshi Kohno. USENIX Association, 2012, pp. 205–220. URL: <https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/heninger>.
- [Hin10] M. Jason Hinek. *Cryptanalysis of RSA and its variants*. Chapman & Hall/CRC Cryptography and Network Security. CRC Press, 2010. ISBN: 978-1-4200-7518-2. DOI: 10.1201/9781420075199. URL: <https://doi.org/10.1201/9781420075199>.
- [HMM10] Wilko Henecka, Alexander May, and Alexander Meurer. “Correcting Errors in RSA Private Keys”. In: *Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings*. Ed. by Tal Rabin. Vol. 6223. Lecture Notes in Computer Science. Springer, 2010, pp. 351–369. DOI: 10.1007/978-3-642-14623-7_19. URL: https://doi.org/10.1007/978-3-642-14623-7_19.
- [How97] Nick Howgrave-Graham. “Finding Small Roots of Univariate Modular Equations Revisited”. In: *Cryptography and Coding, 6th IMA International Conference, Cirencester, UK, December 17-19, 1997, Proceedings*. Ed. by Michael Darnell. Vol. 1355. Lecture Notes in Computer Science. Springer, 1997, pp. 131–142. DOI: 10.1007/BFb0024458. URL: <https://doi.org/10.1007/BFb0024458>.
- [HS09] Nadia Heninger and Hovav Shacham. “Reconstructing RSA Private Keys from Random Key Bits”. In: *Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings*. Ed. by Shai Halevi. Vol. 5677. Lecture Notes in Computer Science. Springer, 2009, pp. 1–17. DOI: 10.1007/978-3-642-03356-8_1. URL: https://doi.org/10.1007/978-3-642-03356-8_1.

- [IES16] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. “Cross Processor Cache Attacks”. In: *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi’an, China, May 30 - June 3, 2016*. Ed. by Xiaofeng Chen, XiaoFeng Wang, and Xinyi Huang. ACM, 2016, pp. 353–364. DOI: 10.1145/2897845.2897867. URL: <http://doi.acm.org/10.1145/2897845.2897867>.
- [Inc+16] Mehmet Sinan Inci et al. “Cache Attacks Enable Bulk Key Recovery on the Cloud”. In: *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*. Ed. by Benedikt Gierlichs and Axel Y. Poschmann. Vol. 9813. Lecture Notes in Computer Science. Springer, 2016, pp. 368–388. DOI: 10.1007/978-3-662-53140-2_18. URL: https://doi.org/10.1007/978-3-662-53140-2_18.
- [Kle+10] Thorsten Kleinjung et al. “Factorization of a 768-Bit RSA Modulus”. In: *Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings*. Ed. by Tal Rabin. Vol. 6223. Lecture Notes in Computer Science. Springer, 2010, pp. 333–350. DOI: 10.1007/978-3-642-14623-7_18. URL: https://doi.org/10.1007/978-3-642-14623-7_18.
- [Koc+19] Paul Kocher et al. “Spectre Attacks: Exploiting Speculative Execution”. In: *2019 IEEE Symposium on Security and Privacy, SP 2019, Proceedings, 20-29 May 2019, San Francisco, California, USA*. IEEE, 2019, pp. 19–37. DOI: 10.1109/SP.2019.00002. URL: <https://doi.org/10.1109/SP.2019.00002>.
- [Kun15] Noboru Kunihiro. “An Improved Attack for Recovering Noisy RSA Secret Keys and Its Countermeasure”. In: *Provable Security - 9th International Conference, ProvSec 2015, Kanazawa, Japan, November 24-26, 2015, Proceedings*. Ed. by Man Ho Au and Atsuko Miyaji. Vol. 9451. Lecture Notes in Computer Science. Springer, 2015, pp. 61–81. DOI: 10.1007/978-3-319-26059-4_4. URL: https://doi.org/10.1007/978-3-319-26059-4_4.
- [Kun18] Noboru Kunihiro. “Mathematical Approach for Recovering Secret Key from Its Noisy Version”. In: *Mathematical Modelling for Next-Generation Cryptography*. Vol. 29. Math. Ind. (Tokyo). Springer, Singapore, 2018, pp. 199–217. DOI: 10.1007/978-981-10-5065-7. URL: <https://doi.org/10.1007/978-981-10-5065-7>.
- [Len+12] Arjen K. Lenstra et al. “Public Keys”. In: *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*. Ed. by Reihaneh Safavi-Naini and Ran Canetti. Vol. 7417. Lecture Notes in Computer Science. Springer, 2012, pp. 626–642. DOI: 10.1007/978-3-642-32009-5_37. URL: https://doi.org/10.1007/978-3-642-32009-5_37.
- [Lip+18] Moritz Lipp et al. “Meltdown: Reading Kernel Memory from User Space”. In: *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. Ed. by William Enck and Adrienne Porter Felt. USENIX Association, 2018, pp. 973–990. URL: <https://www.usenix.org/conference/usenixsecurity18/presentation/lipp>.
- [LLL82] A. K. Lenstra, H. W. Lenstra Jr., and L. Lovász. “Factoring polynomials with rational coefficients”. In: *Math. Ann.* 261.4 (1982), pp. 515–534. ISSN: 0025-5831. DOI: 10.1007/BF01457454. URL: <https://doi.org/10.1007/BF01457454>.

- [Nem+17] Matús Nemeč et al. “The Return of Coppersmith’s Attack: Practical Factorization of Widely Used RSA Moduli”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. Ed. by Bhavani M. Thuraisingham et al. ACM, 2017, pp. 1631–1648. DOI: 10.1145/3133956.3133969. URL: <http://doi.acm.org/10.1145/3133956.3133969>.
- [NV10] Phong Q. Nguyen and Brigitte Vallée, eds. *The LLL Algorithm - Survey and Applications*. Information Security and Cryptography. Springer, 2010. ISBN: 978-3-642-02294-4. DOI: 10.1007/978-3-642-02295-1. URL: <https://doi.org/10.1007/978-3-642-02295-1>.
- [Per05] Colin Percival. “Cache Missing for Fun and Profit”. In: *BSDCan 2005, Ottawa, Canada, May 13-14, 2005, Proceedings*. 2005. URL: <http://www.daemonology.net/papers/cachemissing.pdf>.
- [PGB17] Cesar Pereida García and Billy Bob Brumley. “Constant-Time Callees with Variable-Time Callers”. In: *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. Ed. by Engin Kirda and Thomas Ristenpart. USENIX Association, 2017, pp. 83–98. ISBN: 978-1-931971-40-9. URL: <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/garcia>.
- [PGBY16] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. ““Make Sure DSA Signing Exponentiations Really are Constant-Time””. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. Ed. by Edgar R. Weippl et al. ACM, 2016, pp. 1639–1650. DOI: 10.1145/2976749.2978420. URL: <http://doi.acm.org/10.1145/2976749.2978420>.
- [Pol74] J. M. Pollard. “Theorems on factorization and primality testing”. In: *Mathematical Proceedings of the Cambridge Philosophical Society* 76.3 (1974), pp. 521–528. DOI: 10.1017/S0305004100049252.
- [Pol75] J. M. Pollard. “A Monte Carlo method for factorization”. In: *BIT Numerical Mathematics* 15.3 (1975), pp. 331–334. ISSN: 1572-9125. DOI: 10.1007/BF01933667. URL: <https://doi.org/10.1007/BF01933667>.
- [PPS12] Kenneth G. Paterson, Antigoni Polychroniadou, and Dale L. Sibborn. “A Coding-Theoretic Approach to Recovering Noisy RSA Keys”. In: *Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings*. Ed. by Xiaoyun Wang and Kazue Sako. Vol. 7658. Lecture Notes in Computer Science. Springer, 2012, pp. 386–403. DOI: 10.1007/978-3-642-34961-4_24. URL: https://doi.org/10.1007/978-3-642-34961-4_24.
- [Rab80] Michael O. Rabin. “Probabilistic algorithm for testing primality”. In: *J. Number Theory* 12.1 (1980), pp. 128–138. ISSN: 0022-314X. DOI: 10.1016/0022-314X(80)90084-0. URL: [https://doi.org/10.1016/0022-314x\(80\)90084-0](https://doi.org/10.1016/0022-314x(80)90084-0).
- [RSA78] R. L. Rivest, A. Shamir, and L. Adleman. “A method for obtaining digital signatures and public-key cryptosystems”. In: *Comm. ACM* 21.2 (1978), pp. 120–126. ISSN: 0001-0782. DOI: 10.1145/359340.359342. URL: <https://doi.org/10.1145/359340.359342>.

- [Sch+17] Michael Schwarz et al. “Malware Guard Extension: Using SGX to Conceal Cache Attacks”. In: *Detection of Intrusions and Malware, and Vulnerability Assessment - 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings*. Ed. by Michalis Polychronakis and Michael Meier. Vol. 10327. Lecture Notes in Computer Science. Springer, 2017, pp. 3–24. DOI: 10.1007/978-3-319-60876-1_1. URL: https://doi.org/10.1007/978-3-319-60876-1_1.
- [Ste67] Josef Stein. “Computational problems associated with Racah algebra”. In: *Journal of Computational Physics* 1.3 (1967), pp. 397–405. ISSN: 00219991. DOI: 10.1016/0021-9991(67)90047-2. URL: [https://doi.org/10.1016/0021-9991\(67\)90047-2](https://doi.org/10.1016/0021-9991(67)90047-2).
- [VEW12] Camille Vuillaume, Takashi Endo, and Paul Wooderson. “RSA Key Generation: New Attacks”. In: *Constructive Side-Channel Analysis and Secure Design - Third International Workshop, COSADE 2012, Darmstadt, Germany, May 3-4, 2012. Proceedings*. Ed. by Werner Schindler and Sorin A. Huss. Vol. 7275. Lecture Notes in Computer Science. Springer, 2012, pp. 105–119. DOI: 10.1007/978-3-642-29912-4_9. URL: https://doi.org/10.1007/978-3-642-29912-4_9.
- [Wan+17] Shuai Wang et al. “CacheD: Identifying Cache-Based Timing Channels in Production Software”. In: *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. Ed. by Engin Kirda and Thomas Ristenpart. USENIX Association, 2017, pp. 235–252. URL: <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/wang-shuai>.
- [Wei+18] Samuel Weiser et al. “DATA - Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries”. In: *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. Ed. by William Enck and Adrienne Porter Felt. USENIX Association, 2018, pp. 603–620. URL: <https://www.usenix.org/conference/usenixsecurity18/presentation/weiser>.
- [Wic+18] Jan Wichelmann et al. “MicroWalk: A Framework for Finding Side Channels in Binaries”. In: *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. ACM, 2018, pp. 161–173. DOI: 10.1145/3274694.3274741. URL: <https://doi.org/10.1145/3274694.3274741>.
- [Wie90] Michael J. Wiener. “Cryptanalysis of short RSA secret exponents”. In: *IEEE Trans. Information Theory* 36.3 (1990), pp. 553–558. DOI: 10.1109/18.54902. URL: <https://doi.org/10.1109/18.54902>.
- [WSB18] Samuel Weiser, Raphael Spreitzer, and Lukas Bodner. “Single Trace Attack Against RSA Key Generation in Intel SGX SSL”. In: *Proceedings of the 2018 on Asia Conference on Computer and Communications Security, AsiaCCS 2018, Incheon, Republic of Korea, June 04-08, 2018*. Ed. by Jong Kim et al. ACM, 2018, pp. 575–586. DOI: 10.1145/3196494.3196524. URL: <http://doi.acm.org/10.1145/3196494.3196524>.
- [XCP15] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. “Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems”. In: *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 2015, pp. 640–656. DOI: 10.1109/SP.2015.45. URL: <https://doi.org/10.1109/SP.2015.45>.

- [YF14] Yuval Yarom and Katrina Falkner. “FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack”. In: *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. USENIX Association, 2014, pp. 719–732. ISBN: 978-1-931971-15-7. URL: <https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom>.
- [YGH16] Yuval Yarom, Daniel Genkin, and Nadia Heninger. “CacheBleed: A Timing Attack on OpenSSL Constant Time RSA”. In: *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*. Ed. by Benedikt Gierlichs and Axel Y. Poschmann. Vol. 9813. Lecture Notes in Computer Science. Springer, 2016, pp. 346–367. DOI: 10.1007/978-3-662-53140-2_17. URL: https://doi.org/10.1007/978-3-662-53140-2_17.

A Lattice Construction and Parametrization

Coppersmith’s method reduces the modular equation problem i.e. $f(x) = 0 \pmod p$ to an equation over the integers i.e. $g(x) = 0$ with roots easily found by algorithms like Berlekamp-Zassenhaus. This transformation employs lattice reduction algorithms such as LLL [LLL82] assuming the original modular equation root is small [Cop96].

Following Coppersmith’s approach, Howgrave-Graham [How97] revisited the lattice construction and proposed a new method to build a lattice that allows obtaining a $g(x) = 0$ from the original modular equation $f(x) = 0 \pmod p$. Howgrave-Graham lattice construction is often preferred due to its simplicity and numerous practical advantages [How97, NV10].

Factoring N knowing LSBs of p . Assume we know wlog. the n LSBs of a prime p that is a factor of N , i.e. p is expressed as

$$p = \tilde{p}2^n + p_0$$

where p_0 is the known portion and \tilde{p} the only unknown. Hence \tilde{p} is a *small* root of the polynomial

$$f(x) = x2^n + p_0 \pmod p \quad |\tilde{p}| \leq X$$

and in this case, *small* meaning that \tilde{p} is bound by some known constant X . Coppersmith approach requires $f(x)$ to be monic. To achieve that, define $b = 2^{-n} \pmod N$, where $b2^n = 1 + kN$ for some integer k , then express $f(x)$ as follows.

$$f(x) = x + bp_0 \pmod p \quad |\tilde{p}| \leq X \tag{5}$$

Coppersmith-Howgrave-Graham approach aims to solve (5) by reducing this univariate modular equation to an equation over the integers, visualized below.

$$\underbrace{f(x_0) = 0 \pmod p \Rightarrow f_i(x_0) = 0 \pmod p^m \Rightarrow B \xrightarrow{\text{LLL}} g(x_0) = 0}_{\text{Coppersmith-Howgrave-Graham}}$$

From the monic polynomial $f(x)$, build a set of $d = m + t$ polynomials $f_i(x)$ over p^m according to the Howgrave-Graham approach, such that these $f_i(x)$ have the same root $x_0 = \tilde{p}$ modulo p^m as $f(x)$ modulo p [How97]. Said polynomials are as follows.

$$\begin{aligned} f_i(x) &= N^i f^{m-i}(x) & i &= 0, 1, \dots, m-1 \\ f_{m+i}(x) &= x^i f^m(x) & i &= 0, 1, \dots, t-1 \end{aligned}$$

The next step builds a lattice B from the $f_i(x)$ for $0 \leq i < d$. Following Howgrave-Graham [How97] the basis vectors of B are the coefficient vectors of $f_i(xX)$. Then, lattice-reduced B should yield a $g(x)$ over the integers if the Coppersmith-Howgrave-Graham conditions are respected or heuristically relaxed—we expand later. The small root bound X defines these conditions and the lattice dimension $d = m + t$, therefore we select them such that we can factor N knowing n bits of p .

A.1 Lattice Parametrization

Three parameters control the effectiveness and efficiency of this method: X , m and t , where the latter two control the lattice dimension ($d = m + t$) hence the amount of information in it. This dimension dictates the running time of LLL, therefore the goal is to minimize m and t considering that the attacker should run it for each candidate resulting from the error correction phase.

To validate the Coppersmith-Howgrave-Graham approach, we used a public SageMath implementation⁵. The objective of this validation is to obtain—for n , a given number of LSBs—which parameter set (X, m, t) yields the right solution with very high probability while minimizing the runtime as much as possible. This approach is very similar to that of Nemeč et al. [Nem+17], where the authors fixed the bound X and optimize m and t —however we also tweak X to get some runtime improvements.

One of the main tasks for using Coppersmith-Howgrave-Graham method is selecting the bound X of the unknown root. Recalling Section 2.1, N of an RSA-2048 key has exactly 2048 bits by forcing the two MSBs of p and q to be set, i.e. they have exactly 1024 bits. Hence for RSA-2048, the inequality (6) holds, where the ordering of p and q is arbitrary.

$$q < \sqrt{N} < p < 2^{1024} < 2\sqrt{N} < N \quad (6)$$

Considering that we know the n LSBs of p , we divide (6) by 2^n to obtain bounds for \tilde{p} .

$$\frac{q}{2^n} < \frac{\sqrt{N}}{2^n} < \tilde{p} < \frac{2^{1024}}{2^n} < \frac{2\sqrt{N}}{2^n} < \frac{N}{2^n}$$

This results in the bound $X = \frac{2\sqrt{N}}{2^n}$ that should work for both primes. However, in practice the Coppersmith conditions are slightly pessimistic, hence in our analysis we also consider $X = \frac{\sqrt{N}}{2^n}$.

Optimizing parameters. We aim at finding the parameters that solve the partial factorization problem given n LSBs of a prime p . We are interested in obtaining this parametrization for different values of n , starting from Coppersmith bound for RSA-2048: 512 to 552 bits. The optimization process is as follows: (1) for each n , X , m and t we generate 100 RSA-2048 keys using OpenSSL and try to recover the remaining bits of both primes; (2) filter out those sets (X, m, t) that do not achieve 100% success rate; (3) choose the set that minimizes $m + t$ for each n .

It is worth noting that each lattice test implies recovering the same key with p and with q . This is due to the fact that in our attack scenario (see Section 3.3), the adversary is unaware if the known LSBs correspond to the larger or smaller prime, hence does not know if the bound is respected.

For all values of n , the bound $X = \frac{\sqrt{N}}{2^n}$ provides highly probable solutions and sometimes the lattice dimension shrinks by one. At first glance, this lattice reduction might seem insignificant. But, for example, with $n = 522$ it implies a runtime reduction of roughly 40 s. Indeed this is quick for a single lattice run, but when the number of candidates to test is high (i.e. after error correction) every second counts.

⁵D. Wong, function `coppersmith_howgrave_univariate` [link]

Table 4 summarizes the results of this characterization process for $X = \frac{\sqrt{N}}{2^m}$. One important aspect is that, for either value of X , we could not achieve the Coppersmith bound ($n = 512$). However, as pointed out in Section 5.1 our simulations suggest that the probability of recovering/correcting 512 bits is roughly the same as 522 bits, i.e. $n = 522$ is adequate in our setting.

We executed this parametrization on Sage 8.1 running on Ubuntu 16.04 on an Intel i7-3770 3.4 GHz. The running times in Table 4 correspond to the average of 100 lattice runs, dominated by the execution of LLL (Sage 8.1 default).

Table 4: Lattice attack characterization.

n	m	t	time (s)
522	26	27	133.0
532	13	14	3.0
542	9	10	0.7
552	6	7	0.2

PUBLICATION

IV

Triggerflow: Regression Testing by Advanced Execution Path Inspection

I. Gridin, C. Pereida García, N. Tuveri and B. B. Brumley

*Detection of Intrusions and Malware, and Vulnerability Assessment - 16th
International Conference, DIMVA 2019, Gothenburg, Sweden, June 19-20, 2019,
Proceedings.* Ed. by R. Perdisci, C. Maurice, G. Giacinto and M. Almgren. 2019,
330–350

DOI: 10.1007/978-3-030-22038-9_16

Publication reprinted with the permission of the copyright holders

Triggerflow: Regression Testing by Advanced Execution Path Inspection

Iaroslav Gridin, Cesar Pereida García, Nicola Tuveri, and Billy Bob Brumley

Tampere University, Tampere, Finland

{iaroslav.gridin,cesar.pereidagarcia,nicola.tuveri,billy.brumley}@tuni.fi

Abstract. Cryptographic libraries often feature multiple implementations of primitives to meet both the security needs of handling private information and the performance requirements of modern services when the handled information is public. OpenSSL, the de-facto standard free and open source cryptographic library, includes mechanisms to differentiate the confidential data and its control flow, including run-time flags, designed for hardening against timing side-channels, but repeatedly accidentally mishandled in the past. To analyze and prevent these accidents, we introduce Triggerflow, a tool for tracking execution paths that, assisted by source annotations, dynamically analyzes the binary through the debugger. We validate this approach with case studies demonstrating how adopting our method in the development pipeline would have promptly detected such accidents. We further show-case the value of the tooling by presenting two novel discoveries facilitated by Triggerflow: one leak and one defect.

Keywords: software testing · regression testing · continuous integration · dynamic program analysis · applied cryptography · side-channel analysis · OpenSSL

1 Introduction

Attacks based on Side-Channel Analysis (SCA) are ubiquitous in microarchitectures and recent research [22, 20] suggest that they are much harder to mitigate than originally believed due to flawed system microarchitectures. Constant-time programming techniques are arguably the most effective and cheapest countermeasure against SCA. Functions implemented following this approach, execute and compute results time-independent from the secret inputs, thus avoiding information leakage.

Implementing constant-time code requires a highly specialized and ever growing skill set such as SCA techniques, operating systems, compilers, signal processing, and even hardware architecture; thus it is a difficult and error-prone task. Unfortunately, code is not always easily testable for SCA flaws due to code complexity and the difficulty of creating the tests themselves. Moreover, cryptography libraries tend to offer several versions of a single algorithm to be used in particular cases depending on the users' needs, thus amplifying the confusion and the possibility of using SCA vulnerable functions.

To that end, we present Triggerflow, a tool that allows to selectively track code paths during program execution. The approach used by Triggerflow is elegant in its simplicity: it reports code paths taken by a given program according to the annotations defined by the user. This enables designing simple regression tests to track control flow skew. Moreover, the tool is extendable and can be integrated in the Continuous Integration (CI) development pipeline, to automatically test code paths in new builds. Triggerflow can be used both as a stand-alone tool to continuously test for known flaws, and as a support tool for other SCA tools when the source code is available. It easily allows examining code execution paths to pinpoint code flaws and regressions.

We motivate our work and demonstrate Triggerflow’s effectiveness by adapting it to work with OpenSSL due to its rich history of known SCA attacks, its wide usage in the Internet, and its rapid and constant development stage. We start by back-testing OpenSSL’s previously known and exploited code flaws, where our tool is able to easily find and corroborate the vulnerabilities. Additionally, using Triggerflow we identify new bugs and SCA vulnerabilities affecting the most recent OpenSSL 1.1.1a version.

In summary, Section 2 discusses previous problems and pitfalls in OpenSSL that led to side-channel attacks. Section 3 describes the Triggerflow tool and Section 4 its application in a CI setting. We analyze in Section 5 the new bugs and vulnerabilities affecting OpenSSL, and in Section 6 we back-test known OpenSSL SCA vulnerabilities to validate the tool’s effectiveness. Section 7 looks at related work. In Section 8 we discuss the limitations of our tool, and finally we conclude in Section 9.

2 Background

2.1 The OpenSSL BN_FLG_CONSTTIME Flag

In 2005, OpenSSL started considering SCA in their threat model, introducing code changes in OpenSSL version 0.9.7. The (then new) RSA cache-timing attack by Percival [25] allowed an attacker to recover secret exponent bits during the sliding-window exponentiation algorithm on systems supporting simultaneous multi-threading (SMT). As a countermeasure to this attack, the OpenSSL team adopted two important changes: Commit 3 introduced the constant-time exponentiation flag and `BN_mod_exp_mont_consttime`, a fixed-window modular exponentiation function; and Commit 4 implemented exponent padding. By combining these countermeasures, OpenSSL aimed for SCA resistant code path execution when performing secret key operations during DSA, RSA, and Diffie-Hellman (DH) key exchange, with the goal of performing exponentiation reasonably independent of the exponent weight or length.

The concept is to set the `BN_FLG_EXP_CONSTTIME` flag on `BIGNUM` variables containing secret information: e.g. private keys, secret prime values, nonces, and integer scalars. Once set, the flag drives access to the constant-time security critical modular exponentiation function supporting the flag. Due to performance

reasons, OpenSSL kept both functions: the constant-time version and the non constant-time version of the modular exponentiation operation. The library defaults to the non constant-time function since it assumes most operations are not secure critical, thus they can be done faster, but upon entry to the non constant-time function the input BN variables are checked for the flag and if the program detects the flag is set, it takes an early exit to the constant-time function, otherwise it continues the insecure code path.

As research and attacks on SCA improved, Acıçmez et al. [1] demonstrated new SCA vulnerabilities in OpenSSL. More precisely, the authors showed that the default BN division function, and the Binary Extended Euclidean algorithm (BEEA) function—used in OpenSSL to perform modular inversion operations—are highly dependent on their input values, therefore they leak enough information to perform a cache-timing attack. This discovery forced the introduction of Commit 14, implementing the `BN_div_no_branch` and `BN_mod_inverse_no_branch` functions, offering a constant-time implementation for the respective operations. Moreover, `BN_FLG_EXP_CONSTTIME` was renamed to `BN_FLG_CONSTTIME` to reflect the fact that it offered protection not only to the modular exponentiation function, but to other functions as well.

2.2 Flag Exploitation

During the last three years, the `BN_FLG_CONSTTIME` flag has received a fair amount of attention due to its flawed effectiveness as an SCA countermeasure in OpenSSL. Pereida García et al. [27] showed the issues of having an insecure-by-default approach in OpenSSL by exploiting a flaw during DSA signature generation due to a flag propagation issue. Performing a `FLUSH+RELOAD` [39] attack, the authors fully recover DSA private keys.

Following the previous work, Pereida García and Brumley [26] identified yet another flaw in OpenSSL, this time involving the `BN_mod_inverse` function. Failure to set the flag allowed the authors to successfully perform a cache-timing attack using `FLUSH+RELOAD` to recover secret keys during ECDSA P-256 signature generation in SSH and TLS protocols.

Building on top of the previous works, two research teams [35, 3] discovered independently several SCA flaws in OpenSSL. On the one hand, Aldaya et al. [3] developed and used a simple but effective methodology to find vulnerable code paths in OpenSSL. The authors tracked SCA vulnerable functions in OpenSSL using GDB by placing breakpoints on them. They executed the RSA key generation command, hitting the breakpoints and thus revealing flaws in OpenSSL's RSA key generation implementation. On the other hand, [35] analyzed the RSA key generation implementation and also discovered calls to the SCA vulnerable GCD function. In both cases, the authors noticed a combination of non constant-time functions in use, failure to set flags, and flags not propagated to `BIGNUM` variables caused OpenSSL to leak key bits. Moreover, both works demonstrate that it is possible to retrieve enough key bits to fully recover an RSA key after a single SCA trace using different cache techniques and threat models (page-level or `FLUSH+RELOAD`).

The previous works highlight a clear and serious issue surrounding the constant-time flag. The developers need to identify all the possible security critical cases in OpenSSL where the flag must be set in order to prevent SCA attacks, which has proven to be a laborious and clearly error-prone task. Even if done thoroughly and correctly, the developers must still ensure code changes do not introduce regressions surrounding the flag.

3 Tracking Execution Paths with Triggerflow

OpenSSL’s regression-testing framework has significantly improved over time, notably following the HeartBleed vulnerability. Nevertheless, the framework has its limitations, with real-world constraints largely imposed by portability requirements weighed against engineering effort. With respect to the `BN_FLG_CONSTTIME` flag, the testing framework does not provide a mechanism to track function calls or examine the call stack. This largely contributes to the root cause of the previously discussed vulnerabilities surrounding the `BN_FLG_CONSTTIME` flag: the testing framework cannot accommodate a reasonable regression test in these instances.

With this motivation, our work began by designing Triggerflow¹: a tool for tracking execution paths. After marking up the source code with special comments, its purpose is to detect when code hits paths of interest. We wrote Triggerflow in Ruby² and it uses GDB³ for inspecting code execution. In support of Open Science [18], Triggerflow is free and open source, distributed under MIT license.

We chose GDB since it provides all the required functionality: an established interface for choosing trace points and inspecting the program execution, as well as a machine-readable interface⁴. Additionally, GDB supports a wide variety of platforms, architectures, and languages.

Architecture. The high level concept of Triggerflow is as follows.

1. The inputs to Triggerflow are: a directory with annotated source code, instructions to build it, commands to run and debug, and optionally patches to apply before building.
2. Triggerflow scans the source code for special keywords, which are typically placed in comments near related lines of code, and builds a database of annotations.
3. Triggerflow commences the build, then runs the given commands (*triggers*) under GDB, instructed to set breakpoints at all points of interest.
4. When GDB reports hitting a breakpoint, Triggerflow inspects the backtrace supplied by GDB, makes decisions based on the backtrace and stored annotations, and possibly logs the code path that led to it.

¹ <https://gitlab.com/nisec/triggerflow>

² <https://www.ruby-lang.org/en/>

³ <https://www.gnu.org/software/gdb/>

⁴ https://sourceware.org/gdb/onlinedocs/gdb/GDB_002fMI.html

In addition to verbose raw logging, Triggerflow provides output in Graphviz DOT format, allowing easy conversion to PDF, image, and other formats.

Annotations. Using marked up source code allows leveraging existing tools for merging code changes to (semi)automatically update annotations to reflect codebase changes. It is best when annotations are maintained in the original code, and updated by the author of related changes, but for the purposes of code analysis by a third party, Triggerflow also supports storing annotations separately, in form of patches that define annotation context. Our tool currently supports four different annotations, described below and illustrated in Figure 1.

1. `TRIGGERFLOW_POI` is a point of interest and it is always tracked. The Triggerflow tool reports back every time the executing code steps into it.
2. `TRIGGERFLOW_POI_IF` is a conditional point of interest, thus it is conditionally tracked. The Triggerflow tool reports back every time the code annotated is stepped into and the given expression evaluates to true.
3. `TRIGGERFLOW_IGNORE` is an ignore annotation that allows to safely ignore specific code lines resulting in code execution paths that are not interesting (false positives).
4. `TRIGGERFLOW_IGNORE_GROUP` is a group ignore annotation that allows to safely ignore a specific code execution path if and only if every line marked with the same group ID is stepped into.

<pre> 1 /* code before */ 2 if(a % 2 == 0) // TRIGGERFLOW_POI 3 /* code after */ </pre> <hr/> <pre> 1 if(something) { 2 a = publickey; // ↳ TRIGGERFLOW_IGNORE_GROUP ↳ ec_publickey 3 } 4 call_suspicious_code(a) // ↳ TRIGGERFLOW_IGNORE_GROUP ↳ ec_publickey </pre>	<pre> 1 /* code before */ 2 call_suspicious_code(a) // ↳ TRIGGERFLOW_POI_IF a.private() 3 /* code after */ </pre> <hr/> <pre> 1 int call_suspicious_code(int a) { 2 // TRIGGERFLOW_POI 3 /* something interesting with a */ 4 } 5 call_suspicious_code(public_key) // ↳ TRIGGERFLOW_IGNORE </pre>
--	---

Fig. 1. Annotations currently supported by Triggerflow.

3.1 Annotating OpenSSL

Using the known vulnerable code paths previously discussed in Section 2.2, we created a set of annotations for OpenSSL with the intention to track potential leakage during secure critical operations in different public key cryptosystems such as DSA, ECDSA, RSA, as well as high-level CMS routines.

Following a direct approach, as Figure 2 illustrates we placed `TRIGGERFLOW_POI` annotations to track the code path execution of the most prominent information-leaking functions previously exploited. We placed an annotation in the

`BN_mod_exp_mont` function immediately after the early exit to its constant-time counterpart. In the `BN_mod_inverse` function, we placed a similar annotation after the early exit. We added an annotation at the top of the non constant-time `BN_gcd` function since it is known for being previously used during security critical operations but this function does not have an exit to a constant-time implementation, i.e., it is oblivious to the `BN_FLG_CONSTTIME`.

On the ECC code we annotated the `ec_wNAF_mul` function. This function implements *wNAF* scalar multiplication, a known SCA vulnerable function exploited several times in the past [12, 8, 28, 4, 2]. Similar to the previous cases, upon entry to this function, an early exit is available to a more SCA secure Montgomery ladder scalar multiplication `ec_scalar_mul_ladder`, thus we added the annotation immediately after the early exit.

The strategy to annotate `BN_div` varies depending on the OpenSSL branch. For branches up to and including 1.1.0, the function checks the flag on BN operands and assigns `no_branch = 1` if it detects the flag. Hence we annotate with a `no_branch != 1` conditional breakpoint. The master and 1.1.1 branches recently applied SCA hardening to its callee `bn_div_fixed_top` to make it oblivious to the flag. The corner case is when the number of words in BN operands are not equal, and inside the resulting data-dependent control flow we add an unconditional point of interest annotation.

Ideally, the previous annotations should never be reached, since we assume OpenSSL follows a constant-time code path during the execution of these secure critical operations. Yet one of the most security-critical parts of the process is marking false positive annotations. To give an idea of the scope of such marking, with the above described point of interest annotations applied to the OpenSSL 1.1.0 branch, and no ignore annotations, Triggerflow identifies 84 potentially errant code paths, provided with only a basic set of 25 triggers.

4 Continuous Integration

As previously discussed, our main motivation for Triggerflow is the need to test for regressions in OpenSSL surrounding the `BN_FLG_CONSTTIME` flag. From the software quality perspective, and given the previously exploited vulnerabilities discussed later in Section 6, there is a clear need for an automated approach that accounts for the time dimension and a rapidly changing codebase. Seemingly small and insignificant changes can suddenly shift codepaths, and when PRs are proposed and merged we want to be automatically informed. Using code marked up for Triggerflow allows establishing CI, automatically testing code for introducing unsafe codepaths. We propose (and deploy) the following approach to establish an automatic CI pipeline using Triggerflow and GitLab’s infrastructure, illustrated in Figure 3.

- Create a special Git repository containing Triggerflow configuration, trigger list, annotations in form of Quilt⁵ patch queue, and a submodule containing

⁵ <https://savannah.nongnu.org/projects/quilt>

```

1  int ec_wNAF_mul(const EC_GROUP *group, EC_POINT *r,
↪ const BIGNUM *scalar,
2      size_t num, const EC_POINT *points[],
↪ const BIGNUM *scalars[],
      BN_CTX *ctx)
3  {
4      /* ... */
5      if ((scalar == NULL) && (num == 1)) {
6          return ec_scalar_mul_ladder(group, r,
↪ scalars[0], points[0], ctx);
7      }
8  }
9
10 if (scalar != NULL) { /* TRIGGERFLOW_POI */
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000

```

Fig. 2. Top left: a TRIGGERFLOW_POI annotation in the *wNAF* scalar multiplication function after the early exit. Middle left: a TRIGGERFLOW_POI annotation during *BN_div* execution. Bottom left: a TRIGGERFLOW_POI annotation in OpenSSL’s insecure *BN_gcd* function. Top right: a TRIGGERFLOW_POI annotation in OpenSSL’s *BN_mod_inverse* function after the early exit. Bottom right: a TRIGGERFLOW_POI annotation in *BN_mod_exp_mont* after the early exit.

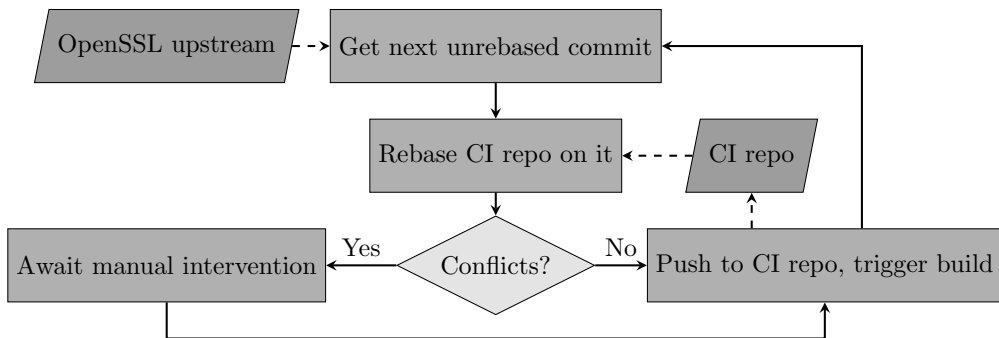


Fig. 3. CI flow illustrated.

code to test (in our case, OpenSSL). This repository is hosted on a GitLab instance and includes the description of the testing process in GitLab format, `.gitlab.yml`.

- Two *runners* are established on separate machines, connected to the GitLab instance. A runner is automated testing software which creates a container and runs testing routines according to rules in `.gitlab.yml`. We maintain two runners with different architectures, `x86_64` and `aarch64`. The runners are based in our infrastructure. When new code is pushed into the GitLab repository and `.gitlab.yml` is present, runners execute the tests and report status back to GitLab, where results are then reviewed.
- A separate software (*repatcher*) is continuously monitoring main OpenSSL code repository for updates and adapting annotations to changed code. If changes can be applied automatically, `repatcher`⁶ pushes updated code to GitLab where it is tested. Otherwise, a human is notified to resolve conflicts and update the patches manually. After that, `repatcher`'s work automatically continues. `Repatcher` is based in our infrastructure.

This process is independent of any support from the original developers. Of course, a better approach is to have developers themselves integrate and maintain Triggerflow annotations upstream, or potentially enforce them at compile time.

Unfortunately, successful deployment of such a CI pipeline depends on code being buildable on every upstream commit, which is sometimes not the case with OpenSSL. Still, with minimal manual inspection it makes a great automatic testing setup: Figure 4 illustrates our CI testing OpenSSL's master branch using Triggerflow. The results of our CI system instance are public⁷, monitoring `master`, `1.1.1` and `1.1.0` branches of OpenSSL.

Average build of OpenSSL on our runners takes 85 s on `x86_64` (440 s on `aarch64`), and Triggerflow takes average of 26 s to run our set of triggers on `x86_64` (92 s on `aarch64`).

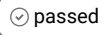


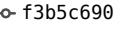



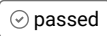


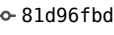



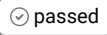


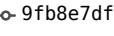



Status	Pipeline	Commit	Stages
 passed	#1495 by  latest	 patched/mas...   [master:c8147d37ccaaf28c...	 00:07:42  1 hour ago
 passed	#1494 by 	 patched/mas...   [master:fe16ae5f95fa86ddb...	 00:07:52  1 hour ago
 passed	#1493 by 	 patched/mas...   [master:0b76ce99aaa5678b...	 00:07:46  1 hour ago

Fig. 4. GitLab CI running: Triggerflow testing OpenSSL code.

⁶ <https://gitlab.com/nisec/repatcher>

⁷ <https://gitlab.com/nisec/openssl-triggerflow-ci>

5 New Bugs and Vulnerabilities

With the tooling in place, our first task was to examine functionality issues that could arise with applying the annotation patches to a shifting codebase. The EC module recently underwent a quite heavy overhaul regarding SCA security [33]. We used that as a case study, and in this section we present two discoveries facilitated by Triggerflow: one leak and one software defect.

5.1 A New Leak

We started from Commit 1 and the Triggerflow unit test in question is ECDSA signing in `ecdsa_oss1.c`. The test passed at that commit, hence the tooling proceeded with subsequent commits. They all passed unit testing, until reaching Commit 2. The purpose of said commit was to fix a regression in the padding of secret scalar inputs in the timing-resistant elliptic curve scalar multiplication, using the group cardinality rather than the generator order, supporting cryptosystems where the distinction is relevant (e.g., ECDH and cofactor variants). Figure 5 illustrates the failed unit test.

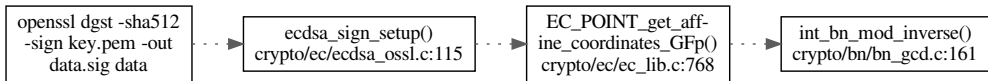


Fig. 5. Insecure flow: projective to affine point conversion (abridged).

The fix. In this case, what the tooling is telling us is that the code is traversing the insecure modular inversion path when converting from projective to affine coordinates. Examining this function, it has always been oblivious to the constant-time flag, yet academic results suggest that said conversion should be protected [24, 23]. Put another way, Commit 2 is not the culprit—the function is insecure by design. Instead of simply enabling the flag, we chose⁸ to add a `field_inv` function pointer inside the `EC_METHOD` structure, alongside existing pointers for other finite field operations such as `field_mul` and `field_sqr`. This allowed us to unify the finite field inversion across the EC module, instead of each function meticulously enabling the constant-time flag when calling `BN_mod_inverse`. Once unified, we can ensure default SCA hardening through a single interface. We provided three different implementations for this pointer for three different `EC_METHOD` instances:

1. `EC_GFp_mont_method` is the default for prime curves and pre-computes a Montgomery arithmetic structure for finite field arithmetic. This is convenient for inversion via FLT, which is modular exponentiation with a fixed exponent and variable base—benefiting generously from the Montgomery

⁸ <https://github.com/openssl/openssl/pull/8254>

arithmetic. Hence our `field_inv` implementation is a straightforward version of FLT in this case.

2. `EC_GFp_simple_method` is a fallback method that contains much of the boilerplate code pointed to by several other `EC_METHOD` implementations. For example, those that implement their own custom arithmetic, such as NIST curves that use Mersenne-like primes. Here, no Montgomery structure is guaranteed to exist. Hence our `field_inv` implementation is blinding, computing $a^{-1} = b/(ab)$ with b chosen uniformly at random and the ab term inverted via `BN_mod_inverse`.
3. `EC_GF2m_simple_method` is the only method for binary curves present in the OpenSSL codebase. Here `field_inv` is a simple wrapper around `BN_GF2m_mod_inv`, which is already SCA-hardened with blinding.

With these SCA-hardened `field_inv` function pointers in place, we then transitioned all finite field inversions in the EC module from `BN_mod_inverse` and `BN_GF2m_mod_inv` to our new pointer, including that of the projective to affine conversion. After these changes, Triggerflow unit tests were successful.

5.2 A New Defect

The previous unit test failure is curious in the sense that Commit 2 was essentially unrelated to projective to affine conversion. As stated above, that conversion has always been oblivious to the constant-time flag. We were left with the question of how such a change could trigger an insecure behavior in an unrelated function.

Using the debugger to compare the internal state when executing `EC_POINT_get_affine_coordinates_GFp` in Commit 2 and its parent, we discovered that, until the latter, a temporary variable storing one of the inputs to `BN_mod_inverse` was flagged as constant-time even if the flag was not explicitly set with the dedicated function. The temporary variable in question was obtained through a `BN_CTX` object, a buffer shared among various functions that simulates a hardware stack to store `BIGNUM` variables, minimizing costly memory allocations—we defer to [13] for more details on the internals of the `BN_CTX` object.

In this case, the `BN_CTX` object is created in the top level function implementing signature generation for the ECDSA cryptosystem, and is shared among most of its callees and descendants; the analysis led to discover that the `BN_CTX` buffer retained the state of `BN_FLG_CONSTTIME` for each stored `BIGNUM` variable, allowing functions to alter the value of `BN_FLG_CONSTTIME`, and thus occasionally the execution flow, of subsequently called functions sharing the same `BN_CTX`.

The fix. This long-standing defect raises several concerns:

- as in the case that led to its discovery, retrieving a `BIGNUM` variable from the `BN_CTX` with `BN_FLG_CONSTTIME` unexpectedly set, might lead to unintentional execution of a timing-resistant code-path. This could be perceived as a benign effect, but hides unexpected risks as it generates false negatives

during security analysis. Moreover, changes as trivial as getting one more temporary variable from the shared `BN_CTX`—or even just changing the order by which temporary variables are retrieved—can influence the execution flow of seemingly unrelated functions, eluding manual analysis and defying developer expectations;

- a `BIGNUM` variable with `BN_FLG_CONSTTIME` unexpectedly set could reach function implementations that execute in variable time and should never be called with confidential inputs marked with `BN_FLG_CONSTTIME`. Such functions diligently check for API abuse and raise exceptions at run time: this defect can then result in unexpected application crashes or potentially expose to bug attacks;
- automated testing is made fragile, in part for the false negatives already mentioned, but additionally because the test suite becomes not representative of external application usage of the library, as different usage patterns of a shared `BN_CTX` in unrelated functions lead to different execution paths. Finally, the generated failure reports could be misleading as changes in unrelated functions might end up triggering errors in other modules.

The fix itself was relatively straightforward, and consisted in unconditionally clearing `BN_FLG_CONSTTIME` every time a `BIGNUM` variable is retrieved from a `BN_CTX`⁹.

What is remarkable is how Triggerflow assisted in the discovery of a defect that had been unnoticed for over a decade, automating the interaction with the debugger to pinpoint which revisions triggered the anomalous behavior.

6 Validation

In order to validate our work, we present next a study of the known flaws briefly discussed in Section 2.2 that led to several SCA attacks, security advisories, and significant manpower downstream to address these issues. We present these flaws as case studies, briefly discussing the root cause, security implications, and the results of running our tooling against an annotated OpenSSL. We separate the cases by cryptosystem and at the same we (mostly) follow the chronological discovery of these flaws.

As part of the validation, we used the same OpenSSL versions as in the original attacks. To that end, we forked OpenSSL branches on the respective versions and then, we applied the set of annotations previously discussed in Section 3.1. This approach allowed us to quickly back test and validate the effectiveness of our tooling to detect potential leakage in OpenSSL.

The list of cases presented here is not exhaustive but serves three purposes:

1. it gives insight to the types of flaws that our Triggerflow is able to find;
2. it shows it is not a trivial task to do, let alone automate; and

⁹ <https://github.com/openssl/openssl/pull/8253>

3. it demonstrates the fragility of the `BN_FLG_CONSTTIME` countermeasure introduced 14 years ago and the need of a secure-by-default approach in cryptography libraries such as OpenSSL.

Moreover, the flaws and vulnerabilities presented in this section and in Section 5 demonstrate the effectiveness and efficiency of integrating Triggerflow to the development pipeline. Maintaining annotations, either as separate patches or integrated in the code base, might be seen as tedious or error-prone but the automation benefits outweigh the disadvantages. On the one hand, maintaining annotations does not require deep and specialized understanding of the code, compared to manually finding and triggering all the possible vulnerable code paths across several platforms, CPUs, and versions. On the other hand, a misplaced annotation does not introduce flaws nor vulnerabilities, since they are used only for testing and reporting purposes.

6.1 DSA

The DSA signature generation implementation in OpenSSL has arguably the longest and most troubled history of SCA issues. In 2016, a decade after `BN_FLG_CONSTTIME` and the constant-time exponentiation function countermeasures were introduced, Pereida García et al. [27] discovered that the constant-time path was not taken due to a flag propagation issue. The authors noticed that `BN_copy` effectively copies the content from a `BIGNUM` variable to another but it fails to copy the existing flags, thus flags are not propagated and the constant-time flag must be set again. This issue left the DSA signature generation vulnerable to cache-timing attacks for more than a decade. To test this issue, we pointed Triggerflow at our annotated `OpenSSL_1_0_2k` branch, resulting in Figure 6 and therefore correctly reporting the flaw.

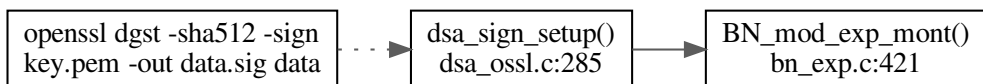


Fig. 6. Triggerflow detecting CVE-2016-2178, the flawed CVE-2005-0109 fix (abridged).

The authors provided a fix for this issue in Commit 5, but at the same time they introduced a new flaw in the modular inversion operation during DSA signature generation. This new vulnerability was enabled due to a missing constant-time flag in one of the input values to the `BN_mod_inverse` function. At that time, the flaw was confined to the development branch, subsequently promptly fixed in Commit 6, thus it did not affect users. Figure 7 shows the result of pointing Triggerflow to OpenSSL in Commit 5, detecting the flawed fix.

Later in 2018, Weiser et al. [36] found additional SCA vulnerabilities in DSA. The authors exploited a timing variation due to the `BIGNUM` structure to re-

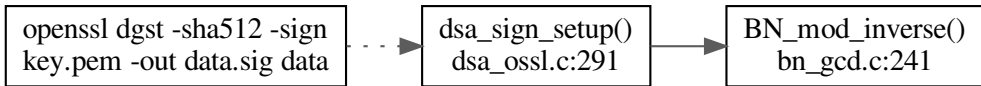


Fig. 7. Triggerflow detecting the flawed CVE-2016-2178 fix (abridged).

cover DSA private keys, an unrelated issue to the `BN_FLG_CONSTTIME` flag. However, the fix provided for this issue in Commit 8 was incomplete, and moreover it introduced a new SCA flaw, once again due to not setting a flag properly. Triggerflow detected this flaw (see Figure 8) in the `OpenSSL_1_1_1` branch, later fixed in Commit 9 but again only present briefly in development branches.

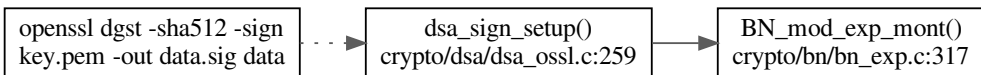


Fig. 8. Triggerflow detecting the flawed CVE-2018-0734 fix (abridged).

In the same work, the authors discovered that every time the library loads a DSA private key, it calculates the corresponding public key following a non constant-time code path due to a missing flag, and therefore is also vulnerable to SCA attacks. In fact, Triggerflow previously detected this vulnerability while back-testing Commit 5, suggesting that this issue was long present in the codebase and could have been detected earlier. This issue was recently fixed in Commit 7.

6.2 ECDSA

OpenSSL’s ECDSA implementation has also been affected by SCA leakage. Pereida García and Brumley [26] discovered that the `BN_FLG_CONSTTIME` flag was not set at all during ECDSA P-256 signature generation. More specifically, the modular inversion operation was performed using the non constant-time path in the `BN_mod_inverse` function, thus leaving the scalar k vulnerable to SCA attacks.

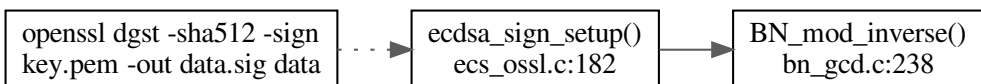


Fig. 9. Triggerflow detecting CVE-2016-7056 (abridged).

Similar to the previous case and in order to back-test this issue, we pointed Triggerflow to the annotated `OpenSSL_1_0_1u` branch and then we generated

ECDSA signatures, triggering the breakpoints. The tool reported back an insecure usage of the modular inversion function as shown in Figure 9. The flag was not set in the nonce k prior to the modular inversion operation. Surprisingly, this issue is still present in the OpenSSL 1.0.1 branch although the authors provided a patch for it, mainlined by the vast majority of vendors. It is worth mentioning the OpenSSL 1.0.1 branch reached EOL around the same time as the work—we assume that is the reason the OpenSSL team did not integrate it.

6.3 RSA

In 2018, two independent works [35, 3] discovered several SCA flaws during RSA key generation in OpenSSL. OpenSSL’s RSA key generation is a fairly complex implementation due to the use of several different algorithms during the process. It requires the generation of random integers; testing the values for primality; computing the greatest common divisor and the least common multiple, using secret values as input. For all of the previous reasons, it is not trivial to implement a constant-time RSA key generation algorithm. Both research works identified missing flags, flags set in the wrong variable, and a direct call to the non constant-time function `BN_gcd` as the culprits enabling the attacks.

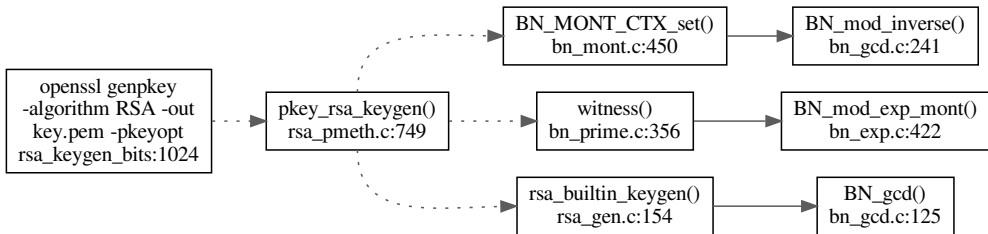


Fig. 10. Triggerflow detecting CVE-2018-0737 (abridged).

During back testing we used an annotated `OpenSSL_1_0_2k` branch, and we pointed the Triggerflow tool at it. It successfully reported all the vulnerabilities discovered by the authors. The authors submitted a total of four commits to OpenSSL codebase to fully mitigate this issue—see Commit 10, Commit 11, Commit 12, and Commit 13 for more details.

7 Related Work

The Triggerflow framework differs from other existing tools in being a tool to assist the development process rather than a system for automated detection and quantification of security vulnerabilities, and aims at being more general purpose and not restricted to the field of cryptographic applications. As such, it should be viewed as complementary rather than alternative to the approaches listed below.

Programming languages. Various works propose and analyze the option of using specialized programming languages to achieve constant-time code generation and verification [10, 14], while others analyze the challenges [7] or opportunities [31] of translating human-readable code into machine instructions through compilers when dealing with cryptographic software and the need for SCA resistant implementations. They differ from this work in the goal: our evaluation is not based on a lack of timing-resistant implementations, but rather in assisting the development process and making sure that insecure paths are not executed, by mistake, with confidential inputs.

Black box testing. These practices are based on statistical analysis to estimate the SCA leakage. *dudect* [29] applies this methodology measuring the timing of the system under test for different inputs.

Static program analysis. These techniques refers to the analysis of the source code [5, 38, 30] (building on the capabilities of the LLVM project to perform the analysis) or annotated machine code [9] of a program to quantify leakages. An alternative to this approach is represented by *CacheAudit* [17, 16] based on symbolic execution, which is usually applied to smaller software or individual algorithms as it requires more resources. *BLAZER* [6] and *THEMIS* [15] employ static analysis to detect side-channels in Java bytecode programs. *BLAZER* introduces a *decomposition* technique associated with *taint tracking* to discover timing channels (or prove their absence) in execution branches tainted by secret inputs. *THEMIS* combines lightweight static taint analysis with precise relational verification to verify the absence of timing or response size side-channels. Similar in spirit as it uses lightweight taint tracking, *Catalyzer* [32] is a closed-source, commercial tool to detect potential leakage by filtering conditional branches and array accesses after marking sensitive inputs; the authors apply their tooling to the C-language MbedTLS library. All of these methods share with Triggerflow the requirement of access to the source code of the tested software (either direct or reasonably decompiled).

Dynamic program analysis. These techniques detect, measure, and accurately locate microarchitecture leakage during the execution of the code in the system. *ctgrind* [21], based on *Valgrind memcheck*, monitors control flow and memory accesses for dependencies on secret data. Previous work [37, 36] uses *Dynamic Binary Instrumentation*, adding instrumentation at run-time to collect metadata and measurements directly to the binary code without altering the execution flow of the program, independently providing extensible frameworks with high accuracy and supporting leakage models for the most relevant microarchitecture attacks. Relevant recent works employ symbolic execution to detect side-channel leaks. *CacheD* [34] is a hybrid approach that combines DBI, symbolic execution, taint tracking, and constraint solving, while the more recent *CaSym* [11] employs cache-aware IR symbolic execution; both works then combine different cache models to detect cache-based timing channels. *SPECTECTOR* [19] uses similar symbolic execution techniques in combination with *speculative non-interference* models to detect speculative execution leaks and

optimization opportunities in the strategies used by compilers to implement hardening measures.

Triggerflow is similar to Dynamic Program Analysis techniques with respect to performing the evaluation when the software is actively running on the target system. Although limited by requiring access to the source code, Triggerflow can leverage this property and avoid any instrumentation: the tested binary is exactly the one generated by the build process of the target, with the only requirement of not stripping the debug symbols, to aid GDB in mapping function names and the memory addresses of the routines included in the target software.

8 Limitations

Triggerflow requires access to the sources of the target software, and to annotate it with markup comments as described in Section 3. Preferably, Triggerflow annotations should be maintained directly in the codebase of the upstream target project, but Triggerflow includes support for versioning of annotation patches for the analysis of third-party projects. Additionally, it is worth stressing that Triggerflow does not automatically detect where to annotate the target code—this goes beyond the tool capabilities. Instead, it relies on developer expertise to annotate the execution paths of interest. As such, source code access is a limit only for the analysis of closed-source third-party projects, which fall out of the immediate scope of Triggerflow as an aid tool for the development process.

Triggerflow depends on the availability of GDB and Ruby on the target platform, and is limited to the executables that can be debugged through GDB. This is arguably a minor concern, with the only remarkable exception that debugging through GDB inside a virtualized container usually requires overriding the default set of system call restrictions that is meant to isolate the supervisor from the container, raising security concerns when running Triggerflow for third-party CI and partially limiting the selection of available CI platforms.

The tools developed during this work can also be applied to other software projects, not just OpenSSL. Triggerflow can work with any language GDB supports and is useful for analyzing and testing execution paths through any complex project that meets the minimal requirements.

A case study. To substantiate the above claims and demonstrate the flexibility of Triggerflow, we annotated the ECC portion of `golang`¹⁰. The documentation states the P384 (pseudo-)class for NIST P-384 curve operations is not constant-time. Indeed, the `ScalarMult` method is textbook double-and-add scalar multiplication. We placed a `TRIGGERFLOW_POI` annotation inside this method, and used a `golang` ECDSA signing application as a trigger. Figure 11 shows the result, confirming Triggerflow is not restricted to OpenSSL or the C language.

¹⁰ <https://golang.org/pkg/crypto/elliptic/>

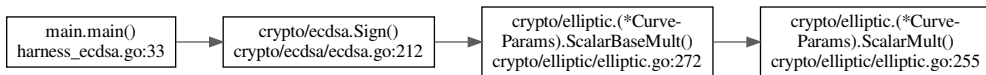


Fig. 11. Triggerflow detecting an insecure scalar multiplication path in golang.

9 Conclusion

Triggerflow complements the results offered by any of the analysis techniques described in Section 7: in large software projects like OpenSSL, pinpointing the location of a detected leak might not be sufficient. Similarly to other cryptographic libraries, OpenSSL often includes several implementations of the same primitive, many of which are designed for performance and safe to use only when all the inputs are public. When a leak is detected in one of these functions, developers are challenged with the task of discovering why and how secret data reached the insecure code path, rather than altering the location where the leakage is reported. As demonstrated in Sections 5 and 6, Triggerflow can be successfully and efficiently used to aid developers in these situations and, through CI, prevent regressions in the handling of secret data.

Considering the high number of valid combinations of supported platforms and build-time options for OpenSSL, and that the available implementations and control flow depend on these specific combinations, Triggerflow is a good solution to aid developers by exhaustively automating the `BN_FLG_CONSTTIME` tests and prevent future regressions similar to the ones described in this work.

In the context of using Triggerflow with OpenSSL to monitor `BN_FLG_CONSTTIME`, it should be mentioned that, security-wise, a secure-by-default approach would be desirable: i.e., all `BIGNUM` are considered *constant-time* unless the programmer explicitly marks them as public, so that when alternatives exist, the default implementation of each algorithm is the timing-resistant one, and insecure but more efficient ones need to be enabled explicitly and after careful examination. On the other hand, such change has the potential for being disruptive for existing applications, and is therefore likely to be rejected or implemented over a long period of time to meet the project release strategy.

Future work. On top of continued development of the tool as discussed, we plan to expand on this work in the future to widen the coverage of the OpenSSL library and of the project *apps* and their options, by setting more triggers and point of interest across multiple architectures and build-time options. In parallel, to further demonstrate the capabilities of the tool we plan to apply a similar methodology to other security libraries and cryptographic software, aiming at uncovering, fixing, and testing related timing leaks.

Responsible disclosure. All PRs submitted as a result of this work were coordinated with the OpenSSL security team. Following the GitHub PR URLs, readers will find more extensive discussions of the security implications of the identified leak and defect. To briefly summarize: (1) the leakage during projective to affine conversion does not appear to be exploitable with recent SCA

hardening to the EC module—we speculate it can only be utilized in combination with some other novel leak, by which time the larger additional leak would likely be enough independently; (2) while we were able to implement a straw man application to demonstrate the BN_CTX defect (reaching unintended code paths and inducing function failures), we were unable to locate a real-world OpenSSL-linking application matching our PoC characteristics, nor any technique to exploit the defect *within* the OpenSSL library itself. We also filed a report with CERT, summarizing our security findings.

Acknowledgments. This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No 804476).

References

1. Aciğmez, O., Gueron, S., Seifert, J.: New branch prediction vulnerabilities in OpenSSL and necessary software countermeasures. In: Cryptography and Coding, 11th IMA International Conference, Cirencester, UK, December 18-20, 2007, Proc. LNCS, vol. 4887, pp. 185–203. Springer (2007), https://doi.org/10.1007/978-3-540-77272-9_12
2. Aldaya, A.C., Brumley, B.B., ul Hassan, S., Pereida García, C., Tuveri, N.: Port contention for fun and profit. In: 2019 IEEE Symposium on Security and Privacy, SP 2019, Proc., 20-22 May 2019, San Francisco, California, USA. pp. 1037–1054. IEEE (2019), <https://doi.org/10.1109/SP.2019.00066>
3. Aldaya, A.C., Pereida García, C., Alvarez Tapia, L.M., Brumley, B.B.: Cache-timing attacks on RSA key generation. IACR Cryptology ePrint Archive 2018(367) (2018), <https://eprint.iacr.org/2018/367>
4. Allan, T., Brumley, B.B., Falkner, K.E., van de Pol, J., Yarom, Y.: Amplifying side channels through performance degradation. In: Proc., 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016. pp. 422–435. ACM (2016), <http://doi.acm.org/10.1145/2991079.2991084>
5. Almeida, J.B., Barbosa, M., Barthe, G., Dupressoir, F., Emmi, M.: Verifying constant-time implementations. In: 25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016. pp. 53–70. USENIX Association (2016), <https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida>
6. Antonopoulos, T., Gazzillo, P., Hicks, M., Koskinen, E., Terauchi, T., Wei, S.: Decomposition instead of self-composition for proving the absence of timing channels. In: Proc., 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017. pp. 362–375. ACM (2017), <https://doi.org/10.1145/3062341.3062378>
7. Balakrishnan, G., Reps, T.W.: WYSINWYX: what you see is not what you execute. ACM Trans. Program. Lang. Syst. 32(6), 23:1–23:84 (2010), <https://doi.org/10.1145/1749608.1749612>
8. Benger, N., van de Pol, J., Smart, N.P., Yarom, Y.: “Ooh Aah... Just a Little Bit”: A small amount of side channel can go a long way. In: Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proc. LNCS, vol. 8731, pp. 75–92. Springer (2014), https://doi.org/10.1007/978-3-662-44709-3_5

9. Blazy, S., Pichardie, D., Trieu, A.: Verifying constant-time implementations by abstract interpretation. In: Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proc., Part I. LNCS, vol. 10492, pp. 260–277. Springer (2017), https://doi.org/10.1007/978-3-319-66402-6_16
10. Bond, B., Hawblitzel, C., Kapritsos, M., Leino, K.R.M., Lorch, J.R., Parno, B., Rane, A., Setty, S.T.V., Thompson, L.: Vale: Verifying high-performance cryptographic assembly code. In: 26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017. pp. 917–934. USENIX Association (2017), <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/bond>
11. Brotzman, R., Liu, S., Zhang, D., Tan, G., Kandemir, M.: CaSym: Cache aware symbolic execution for side channel detection and mitigation. In: 2019 IEEE Symposium on Security and Privacy, SP 2019, Proc., 20-22 May 2019, San Francisco, California, USA. pp. 364–380. IEEE (2019), <https://doi.org/10.1109/SP.2019.00022>
12. Brumley, B.B., Hakala, R.M.: Cache-timing template attacks. In: Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proc. LNCS, vol. 5912, pp. 667–684. Springer (2009), https://doi.org/10.1007/978-3-642-10366-7_39
13. Brumley, B.B., Tuveri, N.: Cache-timing attacks and shared contexts. In: Constructive Side-Channel Analysis and Secure Design - 2nd International Workshop, COSADE 2011, Darmstadt, Germany, February 24-25, 2011. Proc. pp. 233–242 (2011), <https://tutcris.tut.fi/portal/files/15671512/cosade2011.pdf>
14. Cauligi, S., Soeller, G., Brown, F., Johannesmeyer, B., Huang, Y., Jhala, R., Stefan, D.: Fact: A flexible, constant-time programming language. In: IEEE Cybersecurity Development, SecDev 2017, Cambridge, MA, USA, September 24-26, 2017. pp. 69–76. IEEE Computer Society (2017), <https://doi.org/10.1109/SecDev.2017.24>
15. Chen, J., Feng, Y., Dillig, I.: Precise detection of side-channel vulnerabilities using Quantitative Cartesian Hoare Logic. In: Proc., 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017. pp. 875–890. ACM (2017), <https://doi.org/10.1145/3133956.3134058>
16. Doychev, G., Köpf, B.: Rigorous analysis of software countermeasures against cache attacks. In: Proc., 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017. pp. 406–421. ACM (2017), <https://doi.org/10.1145/3062341.3062388>
17. Doychev, G., Köpf, B., Mauborgne, L., Reineke, J.: Cacheaudit: A tool for the static analysis of cache side channels. ACM Trans. Inf. Syst. Secur. 18(1), 4:1–4:32 (2015), <https://doi.org/10.1145/2756550>
18. Gridin, I., Pereida García, C., Tuveri, N., Brumley, B.B.: Triggerflow. Zenodo (Apr 2019), <https://doi.org/10.5281/zenodo.2645805>
19. Guarnieri, M., Köpf, B., Morales, J.F., Reineke, J., Sánchez, A.: SPECTECTOR: principled detection of speculative information flows. CoRR abs/1812.08639 (2018), <http://arxiv.org/abs/1812.08639>
20. Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre attacks: Exploiting speculative execution. In: 2019 IEEE Symposium on Security and Privacy, SP 2019, Proc., 20-22 May 2019, San Francisco, California, USA. pp. 19–37. IEEE (2019), <https://doi.org/10.1109/SP.2019.00002>

21. Langley, A.: ctgrind—checking that functions are constant time with Valgrind. <https://github.com/agl/ctgrind> (2010)
22. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M.: Meltdown: Reading kernel memory from user space. In: 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018. pp. 973–990. USENIX Association (2018), <https://www.usenix.org/conference/usenixsecurity18/presentation/lipp>
23. Maimut, D., Murdica, C., Naccache, D., Tibouchi, M.: Fault attacks on projective-to-affine coordinates conversion. In: Constructive Side-Channel Analysis and Secure Design - 4th International Workshop, COSADE 2013, Paris, France, March 6-8, 2013, Revised Selected Papers. LNCS, vol. 7864, pp. 46–61. Springer (2013), https://doi.org/10.1007/978-3-642-40026-1_4
24. Naccache, D., Smart, N.P., Stern, J.: Projective coordinates leak. In: Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004, Proc. LNCS, vol. 3027, pp. 257–267. Springer (2004), https://doi.org/10.1007/978-3-540-24676-3_16
25. Percival, C.: Cache missing for fun and profit. In: BSDCan 2005, Ottawa, Canada, May 13-14, 2005, Proc. (2005), <http://www.daemonology.net/papers/cachemissing.pdf>
26. Pereida García, C., Brumley, B.B.: Constant-time callees with variable-time callers. In: 26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017. pp. 83–98. USENIX Association (2017), <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/garcia>
27. Pereida García, C., Brumley, B.B., Yarom, Y.: “Make sure DSA signing exponentiations really are constant-time”. In: Proc., 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016. pp. 1639–1650. ACM (2016), <http://doi.acm.org/10.1145/2976749.2978420>
28. van de Pol, J., Smart, N.P., Yarom, Y.: Just a little bit more. In: Topics in Cryptology - CT-RSA 2015, The Cryptographer’s Track at the RSA Conference 2015, San Francisco, CA, USA, April 20-24, 2015. Proc. LNCS, vol. 9048, pp. 3–21. Springer (2015), https://doi.org/10.1007/978-3-319-16715-2_1
29. Reparaz, O., Balasch, J., Verbauwhede, I.: Dude, is my code constant time? In: Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017. pp. 1697–1702. IEEE (2017), <https://doi.org/10.23919/DATE.2017.7927267>
30. Rodrigues, B., Pereira, F.M.Q., Aranha, D.F.: Sparse representation of implicit flows with applications to side-channel detection. In: Proc., 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016. pp. 110–120. ACM (2016), <http://doi.acm.org/10.1145/2892208.2892230>
31. Simon, L., Chisnall, D., Anderson, R.J.: What you get is what you C: controlling side effects in mainstream C compilers. In: 2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018. pp. 1–15. IEEE (2018), <https://doi.org/10.1109/EuroSP.2018.00009>
32. Takarabt, S., Schaub, A., Facon, A., Guilley, S., Sauvage, L., Souissi, Y., Mathieu, Y.: Cache-timing attacks still threaten IoT devices. In: Codes, Cryptology and Information Security - Third International Conference, C2SI 2019, Rabat, Morocco, April 22-24, 2019, Proc. - In Honor of Said El Hajji. LNCS, vol. 11445, pp. 13–30. Springer (2019), https://doi.org/10.1007/978-3-030-16458-4_2

33. Tuveri, N., ul Hassan, S., Pereida García, C., Brumley, B.B.: Side-channel analysis of SM2: A late-stage featurization case study. In: Proc., 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018. pp. 147–160. ACM (2018), <https://doi.org/10.1145/3274694.3274725>
34. Wang, S., Wang, P., Liu, X., Zhang, D., Wu, D.: Cached: Identifying cache-based timing channels in production software. In: 26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017. pp. 235–252. USENIX Association (2017), <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/wang-shuai>
35. Weiser, S., Spreitzer, R., Bodner, L.: Single trace attack against RSA key generation in Intel SGX SSL. In: Proc., 2018 on Asia Conference on Computer and Communications Security, AsiaCCS 2018, Incheon, Republic of Korea, June 04-08, 2018. pp. 575–586. ACM (2018), <http://doi.acm.org/10.1145/3196494.3196524>
36. Weiser, S., Zankl, A., Spreitzer, R., Miller, K., Mangard, S., Sigl, G.: DATA - differential address trace analysis: Finding address-based side-channels in binaries. In: 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018. pp. 603–620. USENIX Association (2018), <https://www.usenix.org/conference/usenixsecurity18/presentation/weiser>
37. Wichelmann, J., Moghimi, A., Eisenbarth, T., Sunar, B.: MicroWalk: A framework for finding side channels in binaries. In: Proc., 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018. pp. 161–173. ACM (2018), <https://doi.org/10.1145/3274694.3274741>
38. Wu, M., Guo, S., Schaumont, P., Wang, C.: Eliminating timing side-channel leaks using program repair. In: Proc., 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018. pp. 15–26. ACM (2018), <https://doi.org/10.1145/3213846.3213851>
39. Yarom, Y., Falkner, K.: FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In: Proc., 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014. pp. 719–732. USENIX Association (2014), <https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom>

A OpenSSL Commits

- | | |
|--|--|
| 1. fe2d3975880e6a89702f18ec58881307bf862542 | 8. a9cfb8c2aa7254a4aa6a1716909e3f8cb78049b6 |
| 2. a766aab93a282774e63ba918d0bb1c6680a5f292 | 9. 00496b6423605391864fbbd1693f23631a1c5239 |
| 3. 46a643763de6d8e39ecf6f76fa79b4d04885aa59 | 10. e913d11f444e0b46ec1ebbf3340813693f4d869d |
| 4. 0ebfcc8f92736c900bae4066040b67f6e5db8edb | 11. 8db7946ee879ce483f4c81141926e1357aa6b941 |
| 5. 621eaf49a289bfac26d4cbcbdb7396e796784c534 | 12. 54f007af94b8924a46786b34665223c127c19081 |
| 6. b7d0f2834e139a20560d64c73e2565e93715ce2b | 13. 6939eab03a6e23d2bd2c3f5e34fe1d48e542e787 |
| 7. 6364475a990449ef33fc270ac00472f7210220f2 | 14. bd31fb21454609b125ade1ad569ebcc2a2b9b73c |

PUBLICATION

V

Side-Channel Analysis of SM2: A Late-Stage Featurization Case Study

N. Tuveri, S. ul Hassan, C. Pereida García and B. B. Brumley

Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018. 2018, 147–160

DOI: 10.1145/3274694.3274725

Publication reprinted with the permission of the copyright holders



Side-Channel Analysis of SM2: A Late-Stage Featurization Case Study

Nicola Tuveri
Tampere University of Technology
Tampere, Finland
nicola.tuveri@tut.fi

Cesar Pereida García
Tampere University of Technology
Tampere, Finland
cesar.pereidagarcia@tut.fi

Sohaib ul Hassan
Tampere University of Technology
Tampere, Finland
sohaibulhassan@tut.fi

Billy Bob Brumley
Tampere University of Technology
Tampere, Finland
billy.brumley@tut.fi

ABSTRACT

SM2 is a public key cryptography suite originating from Chinese standards, including digital signatures and public key encryption. Ahead of schedule, code for this functionality was recently mainlined in OpenSSL, marked for the upcoming 1.1.1 release. We perform a security review of this implementation, uncovering various deficiencies ranging from traditional software quality issues to side-channel risks. To assess the latter, we carry out a side-channel security evaluation and discover that the implementation hits every pitfall seen for OpenSSL's ECDSA code in the past decade. We carry out remote timings, cache timings, and EM analysis, with accompanying empirical data to demonstrate secret information leakage during execution of both digital signature generation and public key decryption. Finally, we propose, implement, and empirically evaluate countermeasures.

KEYWORDS

software engineering; applied cryptography; public key cryptography; side-channel analysis; timing attacks; cache-timing attacks; power analysis; TVLA; SM2; OpenSSL

ACM Reference Format:

Nicola Tuveri, Sohaib ul Hassan, Cesar Pereida García, and Billy Bob Brumley. 2018. Side-Channel Analysis of SM2: A Late-Stage Featurization Case Study. In *2018 Annual Computer Security Applications Conference (ACSAC '18), December 3–7, 2018, San Juan, PR, USA*. ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/3274694.3274725>

1 INTRODUCTION

SM2¹ is a suite of elliptic curve public key cryptosystems, standardized as a part of Chinese commercial cryptography mandates. Support for SM2 in OpenSSL landed in the public GitHub repository through pull request (PR) #4793,² created in November 2017

¹<https://tools.ietf.org/html/draft-shen-sm2-ecdsa-02>

²<https://github.com/openssl/openssl/pull/4793>

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ACSAC '18, December 3–7, 2018, San Juan, PR, USA

© 2018 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-6569-7/18/12.

<https://doi.org/10.1145/3274694.3274725>

by external contributors. During the review process, in January 2018, the OpenSSL team assigned the PR to the Post-1.1.1 milestone, marking functionality intended to be merged *after* the upcoming 1.1.1 release of OpenSSL.

Due to this, SM2 support was excluded from the two alpha releases for OpenSSL 1.1.1. But in March 2018, just *before* the release of the first 1.1.1 beta—and the associated feature freeze—the OpenSSL development team decided to merge the PR into the 1.1.1 beta development cycle, to have a chance to work on it and possibly include SM2 support as part of the upcoming minor release rather than waiting for the next one.³ Considering that new features can only be added with a new minor release and that the current one (OpenSSL 1.1.0) was released on August 2016, it is likely that a similar—if not longer—development cycle might be required before the SM2 functionality could be added to OpenSSL. The SM2 functionality has thus been part of the beta development cycle since the release of OpenSSL 1.1.1-pre3 (beta 1).

At the time of beta 1 release, the release timetable⁴ for OpenSSL 1.1.1 envisioned four beta releases, aiming at 15th May 2018 as the first possible final release date. As such, the addition of SM2 support into the active development branch occurred at an extremely late stage to be included in the upcoming release cycle, giving a remarkably short window for public review before the final release.

The original release timeline was later⁵ updated, waiting for the final publication of TLS 1.3 as RFC 8446, adding more beta releases, and eventually shifting the final release date for OpenSSL 1.1.1 to September 11, 2018.

Motivation and goal. The first contribution of our work, our initial security review revealed that the late-stage featurization process resulted in various deficiencies, ranging from code quality issues to traditional software defects, and hinted at significant side-channel analysis (SCA) risks based on previous SCA results targeting ECC within OpenSSL. The goal of this research consists in empirically verifying these SCA deficiencies, and then responsibly mitigate them, aiming at intersecting the upcoming OpenSSL 1.1.1 release to ensure these vulnerabilities do not affect released versions of the library.

³<https://github.com/openssl/openssl/pull/4793#pullrequestreview-104954310>

⁴<https://mta.openssl.org/pipermail/openssl-project/2018-March/000372.html>

⁵<https://github.com/openssl/web/pull/55>

Furthermore, taking SM2 as a case study, we criticize the current status of the project. It demonstrates that implementing new functionality without reintroducing previously fixed vulnerabilities proves to be unnecessarily challenging, requiring intimate familiarity with internal details of lower level library modules (e.g. where, when, and how constant-time flags *must* be re-/enabled, which codepaths in the lower EC and BIGNUM modules require to use implementations with SCA mitigations, etc.). Hence, as a secondary goal, we also aim at reviewing the abstraction level at which current SCA countermeasures are implemented, and push for a *secure-by-default* approach—within the boundaries the project enforces for a minor release—so that future implementations will by default benefit from them.

Structure and our contributions. Section 2 reviews relevant background and previous work. We present our security analysis related to the integration of the SM2 functionality in the OpenSSL codebase in Section 3, offering an overview of the issues uncovered. In Section 4, Section 5, and Section 6, respectively, we evaluate SCA defects in the SM2 implementation related to remote timings, cache timings and EM analysis. We propose, implement and empirically evaluate appropriate mitigations in Section 7. Finally, we conclude in Section 8.

2 BACKGROUND

This section describes SM2, various SCA techniques that potentially apply to SM2 implementations, and summarizes previous work on SM2 implementation attacks.

2.1 SM2: Chinese Cryptography Standards

SM2 consists of a digital signature scheme (SM2DSA), a public key encryption scheme (SM2PKE), and a key agreement protocol. In this work, we restrict to SM2DSA and SM2PKE.

Elliptic curves and SM2. While the RFC contains cryptosystem test vectors for several different curves in simplified Weierstrass form (over both prime and binary fields), one required curve⁶ consists of all the (x, y) points $(x, y \in GF(p))$ satisfying the equation

$$E : y^2 = x^3 + ax + b$$

over $GF(p)$ along with the point-at-infinity (group identity element). The domain parameters are consistent with legacy ECC, setting p a 256-bit Mersenne-like prime, $a = -3 \in GF(p)$, both $b \in GF(p)$ and generator point $G \in E$ seemingly random, and prime group order n (i.e. co-factor $h = 1$) slightly below 2^{256} .

SM2DSA digital signatures. The user's private-public keypair is (d_A, Q_A) where d_A is chosen uniformly from $[1..n-1]$ and $Q_A = [d_A]G$ holds. Denote ZA the personalization string (hash) and m the message. Digital signatures compute as follows.

- (1) Compute the digest $h = H(ZA \parallel m)$.
- (2) Select a secret nonce k uniformly from $[1..n]$.
- (3) Compute $(x, y) = [k]G$.
- (4) Compute $r = h + x \bmod n$.
- (5) Compute $s = (1 + d_A)^{-1}(k - rd_A) \bmod n$.
- (6) If any of $r = 0$, $s = 0$, or $s = k$ hold, retry.

⁶OID 1.2.156.10197.1.301

- (7) Return the SM2 digital signature (r, s) .

Hash function H can be any “approved” function, including SM3⁷ standardized in a parallel effort. Verification is not relevant to this work, hence we omit the description.

SM2PKE public key encryption. SM2PKE is roughly analogous to ECIES [2, Sec. 5.1]. Denote the ciphertext $C = C_1 \parallel C_2 \parallel C_3$ where, at a high level, C_1 represents the sender's ephemeral Diffie-Hellman public key (point), C_2 is the One-Time-Pad (OTP) ciphertext (with length $|C_2|$), and C_3 is the authentication tag. The recipient with private-public keypair (d_B, Q_B) recovers the plaintext from C as follows.

- (1) Convert C_1 to a point on E . If C_1 is not on the curve or does not have order n , return an error.
- (2) Compute $(x, y) = [d_B]C_1$, the shared ECDH point.
- (3) Compute $z = KDF(x \parallel y, |C_2|)$, the OTP key; $|z| = |C_2|$.
- (4) Compute $m' = z \oplus C_2$, i.e. OTP decryption.
- (5) Compute $t' = H(x \parallel m' \parallel y)$, the purported tag.
- (6) If $t' \neq C_3$ holds, return an error.
- (7) Return the plaintext m' .

Encryption is not relevant to this work, hence we omit the description.

2.2 Remote Timing Attacks

Timing attacks exploit differences in the time required by a specific implementation to perform an operation on different inputs. In the case of hardware or software cryptosystem implementations, if there is a correlation between the timing of an operation and some secret inputs, the leaked information might be used to mount an attack to recover secret material.

In his seminal work, Kocher [49] introduces a number of simple timing attacks on modular exponentiation and modular reduction implementations, affecting implementations of public key cryptosystems with a static key such as RSA and static Diffie-Hellman or DSA implementations that precompute the ephemeral part.

Brumley and Boneh [22, 23] demonstrate that timing attacks apply also to general software systems, defying contemporary common belief, by devising a timing attack against the OpenSSL implementation of RSA decryption—exploiting time dependencies introduced by the Montgomery reduction and the multiplication routines—and ultimately retrieving the complete factorization of the key pair modulus. Moreover, they demonstrate that such attacks are practical even in a remote scenario, mounting a real-world attack through a client timing RSA decryptions during SSL handshakes with an OpenSSL server. The attack is effective when performed between two processes running on the same host, across co-located virtual machines, and in local networks. They analyze three possible defenses, favoring RSA blinding, and as a consequence several cryptographic libraries, including OpenSSL, enable RSA blinding by default as a countermeasure.⁸

Aciüzmez et al. [5] further improve the original attack, by targeting Montgomery Multiplications in the *table initialization phase*

⁷<https://tools.ietf.org/html/draft-oscca-cfrg-sm3-02>

⁸The issue uncovered by their work was tracked in the public CVE dictionary with the id CVE-2003-0147, and, addressing it, OpenSSL issued a Security Advisory (17 March 2003), and CERT issued vulnerability note VU#997481.

of the sliding window algorithm used to perform the RSA exponentiation in OpenSSL, rather than the *exponentiation phase* itself, increasing the number of multiplications that leak timing information used to retrieve one of the secret prime factors of RSA moduli.

Chen et al. [26] build on these two attacks, improving the success rate through an error detection and correction strategy, thus reducing the number of queries required to mount a successful attack and affecting the total time of the attack and its detectability.

Brumley and Tuveri [21] present another end-to-end remote timing attack: it similarly demonstrates full key recovery in local and remote scenarios, and targets the OpenSSL Montgomery’s ladder implementation for scalar multiplication on elliptic curves over binary fields. The Montgomery ladder algorithm is often recommended as a countermeasure to side-channel attacks due to a fixed sequence of curve operations, that does not depend on the values of individual bits in the secret scalar, while still being computationally fast with no large memory overhead. Nonetheless, the attack exploits exactly the regularity feature of the algorithm, as it creates a direct linear correlation between the binary logarithm (i.e. the bit length) of the secret scalar and the number of iterations (and thus curve operations) in the ladder.

The authors exploit this vulnerability by mounting an attack that collects several measures of the wall-clock execution time of a partial TLS handshake, using an ECDHE_ECDSA ciphersuite over a binary curve. The collected measures are heavily dominated by the EC scalar multiplication of the ECDSA signature generation, implemented using the Montgomery ladder, and thus can be directly correlated with the bit length of the secret scalar (the ephemeral nonce of the ECDSA signature generation algorithm). A second, offline, post-processing phase then uses this partial knowledge to recover the full secret key through a lattice attack.

The proposed countermeasure, adopted by OpenSSL, is based on conditionally padding the nonce before the actual scalar multiplication, to always work on scalars of fixed length (i.e. adding once or twice the group order to the scalar yields an equivalent scalar with the topmost bit set) which in turn fixes the number of curve operations in the ladder and the associated execution time.⁹

Timing measurement noise heavily affects the success rate of the described attacks, usually resulting in the attacks being unfeasible over a wireless link and having severely limited feasibility over a WAN connection due to both decreased accuracy and the total time of the attacks (which is generally further increased to compensate the noise by collecting more samples). However, more recent results [32] address the latter scenario, studying the statistical distribution of latency over different network environments and designing specialized filters to significantly reduce the effect of *jitter* (i.e. the random noise on the latency introduced by additional hops in the route(s) of a network connection). These filters allow attackers to measure events with higher accuracy over the Internet, with potential effects on the feasibility of remote timing attacks over WAN connections.

Timing as a side-channel is not limited to the execution time of a whole cryptographic operation, and is often a gateway to retrieve

information from other resources shared between an attacker and a victim, including microarchitecture components, as in the cache-timing attacks covered below or, switching to the domain of web privacy, even virtual constructs in modern web browsers [71, 72].

Alternatively, the timing side-channel can be used to build reliable oracles, often circumventing trivial implementations of countermeasures to prevent other side-channel attacks. In 1998, Bleichenbacher [16] presented a famous adaptive chosen-ciphertext attack on SSL/TLS ciphersuites based on RSA and PKCS#1 v1.5 encryption padding, based on an oracle built on top of different error messages sent by servers in case of malformed ciphertexts during the SSL/TLS handshake. As a result of the work, subsequent specifications of the TLS protocol (starting from RFC 2246 [33] TLS 1.0, in the same year) recommend “to treat incorrectly formatted messages in a manner indistinguishable from correctly formatted RSA blocks”. But when implementations fail to extend this recommendation to the execution time of handling different events and conditions, the timing side-channel can be used to build an alternative oracle, effective for remote exploitation, as presented in 2014 by Meyer et al. [56]. Their work targeted, among others, the default Java Secure Socket Extension (JSSE) and OpenSSL implementations of the SSL/TLS protocol.

2.3 Cache Timing Attacks

Cache-timing attacks are a subset of microarchitecture attacks targeting specifically the cache hierarchy. Cache-timing attacks against implementations of cryptography primitives exploit two key features: (1) the timing variation introduced by the cache hierarchy; and (2) the non-constant time execution of algorithms handling confidential data used by cryptography primitives and algorithms, e.g. key generation [8, 73], digital signatures [11, 65], encryption [12] and key exchange [39]. Typically, the ultimate goal of a cache-timing attack is to recover confidential information from an algorithm execution and this is done by correlating cache timing data to either the execution time of the algorithm in use, its internal state during execution, or the output of the algorithm. Cache-timing attacks are enabled by several cache attack techniques proposed and used successfully in the past, e.g. EVICT+TIME [63], PRIME+PROBE [64], and FLUSH+RELOAD [75]. The choice of attack technique depends on the attack scenario since each technique has its own advantages and disadvantages.

Cache Architecture. Accessing data and instructions from main memory is not an instant operation since it takes time to locate and fetch the data, thus delaying the execution of the processor. To improve the efficiency of the processor, the memory hierarchy includes memory banks called caches, located between the CPU cores and the RAM. Caches are smaller and faster compared to RAM and main memory, helping to improve the performance by exploiting spacial and temporal locality during memory access.

Modern CPUs contain multiple cache levels, usually *L1* and *L2* caches are private to a specific core and the last level cache (LLC) is shared among all the cores. Typically, the LLC is said to be *inclusive*, meaning that it contains a superset of the data in the caches below it, thus it contains both instructions and data from *L1* and *L2*. The caches are organized into fixed size *cache lines* which are grouped in *cache sets*. The number of cache lines in a cache set is

⁹To track the issue uncovered by this work the id CVE-2011-1945 was assigned and CERT issued the vulnerability note VU#536044.

the *associativity*, i.e., a cache with W lines in each set is a W -way *set-associative* cache.

When the CPU needs to fetch data from memory, it first checks in the caches; if the data is there, a *cache hit* occurs and the load delay is short. On the other hand, when the data is not found in the caches, a *cache miss* occurs and the data must be fetched from a higher level memory, causing a longer delay. A copy of the data fetched from a higher level is cached, exploiting temporal locality. In addition, data close to the accessed data will be fetched and cached too, exploiting spatial locality. If a cache miss occurs and all the cache lines are in use, one of the cache lines is evicted, freeing space for the new data. In order to determine the cache line to evict, modern CPUs use variations of the least-recently-used (LRU) replacement policy.

FLUSH+RELOAD. Proposed by Yarom and Falkner [75], this powerful technique positively identifies accesses to specific memory lines with a high resolution, high accuracy, and high signal-to-noise ratio. Moreover, the technique relies on cache sharing between the CPU cores, typically achieved through the use of shared libraries.

A round of attack consists of three phases: (1) the attacker evicts the target memory line using the `clflush` instruction; (2) the attacker waits some time for the victim to access the memory line; (3) the attacker measures the time it takes to reload the memory line. The timing reveals whether or not the memory line was accessed by the victim during the waiting period, i.e. identifies cache hits and cache misses.

In addition to cache-timing attacks on cryptography, the FLUSH+RELOAD technique has been applied in clever ways targeting the kernel [45], web server function calls [77], user input [42, 51], covert channels [55], as well as more powerful microarchitecture attacks such as the Meltdown [52] and Spectre [48] attacks.

2.4 EM Analysis

Introduced by Kocher et al. [50], power analysis exploits the correlation between sensitive data and changing power leakages on the device. These power fluctuations are a result of transistor switching between the logic levels of CMOS circuits, and the current flow on data lines, as a result of processor activity and memory accesses.

Due to the tightly packaged components on modern devices, power analysis can be difficult to perform with limited or no access to power rails and only noisy global power consumption. As an alternative, Electromagnetic (EM) emanations—a by-product caused by the current flow on data lines and power rails—originally proposed as cryptographic side-channels by Quisquater and Samyde [66], provides a spatial dimension to perform side-channel analysis in isolation from unwanted leakage.

Various techniques exploit data dependent EM leakage such as Differential Power Analysis [50], Correlation Power Analysis [17], Template Attacks [25] and Horizontal attacks [30, 34]. Identifying data dependent EM leakage can be challenging due to additional noise and other unwanted artifacts, thus in addition to simple frequency analysis, additional leakage detection statistical tools are required such as Mutual Information Analysis [29], χ^2 -test [58] and Test Vector Leakage Assessment (TVLA) [41, 67].

Originally developed by Cryptography Research, Inc. for AES [41] and later adapted for public-key cryptography [47], TVLA is a

preferred choice for applying black-box leakage detection testing to identify side-channel weaknesses [28, 61]. TVLA is based on Welch's T-test [74], which computes a statistical value, i.e. confidence interval (CI) to accept or reject the null hypothesis. More specifically, the test validates whether two sets of samples are taken from similar data by comparing the averages of the two data sets. Formally, for two sets S_1 and S_2 , the T-test computes as

$$t = \frac{\mu_1 - \mu_2}{\sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}}$$

where μ_1 , σ_1 , and n_1 are the mean, standard deviation, and cardinality of S_1 , respectively, and similarly for S_2 . The T-test will fail at some discrete sample point if the value is greater than some threshold $C\tau$. In the context of side-channel data, usually fixed vs random test samples are compared to identify points with data dependent leakage [41].

Contemporary works demonstrate the effectiveness of EM analysis on modern PCs, embedded and mobile devices on various open source libraries such as GnuPG and OpenSSL, for attacking cryptosystems like AES [54], RSA [38], ECDH [36], and ECDSA [37]. Moreover, e.g. Goller and Sigl [40] successfully demonstrate the viability of EM attacks over varying distances from mobile devices on ECC and RSA.

Longo et al. [54] performed localized EM analysis on a modern embedded device running software based OpenSSL AES, a bit-sliced optimized implementation for SIMD NEON core, and an AES hardware engine. They applied TVLA to identify EM leakages and subsequently carry out template attacks. Genkin et al. [37] were able to filter out EM emanations from a mobile device at very low frequencies using inexpensive equipment and additional signal processing steps. Their attack successfully recovered a few bits of ECDSA nonces, targeting the OpenSSL wNAF implementation. With roughly 100 signatures, they then successfully mounted a lattice attack for full key recovery.

2.5 SM2 Implementation Attacks: Previous Work

Due to only recently being standardized and coupled with lack of sufficient public implementations and deployments, academic results on attacking SM2 implementations are limited in number. Nevertheless, existing results suggest that implementation attacks on ECDSA generally extend—with slight modification—to SM2DSA. A brief review follows.

Liu et al. [53] were the first to construct an SM2DSA analogue of existing lattice-based ECDSA key recovery with partially known nonces. The authors model exposure of three LSBs, and with 256-bit p and n recover a private key from 100 signatures with reasonable probability and modest computation time.

Chen et al. [27] were the first to implement an SM2DSA lattice attack with real traces. They target an SM2DSA smartcard implementation and distinguish least significant byte collisions by detecting Hamming weight with PCA-based techniques. Restricting to byte values `0x00` and `0xFF`, the authors obtain 120K signatures with power traces, filter them to 48 pairs, and iteratively construct lattice problem instances to recover a private key. Interestingly, the target is not the underlying ECC itself, but data moves by the RNG

during nonce generation. In that respect, their attack is independent of the underlying ECC arithmetic.

Building on [11, 20] that focus on the LSDs of the wNAF for ECDSA nonces, Zhang et al. [76] extend the analysis to SM2DSA. With their own implementation of ECC including traditional wNAF scalar multiplication paired with SM2DSA, they demonstrate it is possible to reliably capture the sequence of ECC doubles and adds through SPA on an Atmega128. Subsequently modeling the filtered nonces with sufficient zeros in the LSDs and constructing lattice problem instances, they recover private keys with high probability. Since they target least significant zeros in the wNAF expansion, their attack is largely independent of the scalar representation—for example, it immediately applies to binary, sliding window, and fixed window expansions. Their work provides even further evidence that ECDSA-type leaks are similarly detrimental to SM2DSA.

While no English version is available, the abstract of [68] suggests a CPA attack to recover the SM2PKE session key exploiting potential leakage from the SM3 compression function execution. That is, the target is not the ECC but the subsequent KDF.

3 SM2 IN OPENSSL

Refer to Section 1 for the detailed timeline of the SM2 feature within OpenSSL. With the narrow review window induced by the release milestone shift, several security (and functionality) issues were mainlined into the OpenSSL codebase. We give an overview of these issue in this section. Listing 1 includes an extract of the SM2DSA signature generation implementation and Listing 2 for SM2PKE public key decryption, as of OpenSSL 1.1.1-pre5 (beta 3).

Code review. Due to the hasty review process, the code implementing SM2 in the beta releases is evidently not in line with the quality standards of analogous components of `libcrypto`,¹⁰ lacking test coverage, including critical bugs (e.g. double frees and wrong return values), a lack of return values checking and poor error handling. These defects are particularly evident in the integration with the `EVP_PKEY` (and `EVP_DigestSign`) API, which is the main entry point for `libssl` and internal and external applications for using the cryptographic functionality included in `libcrypto`.

SCA review. Beyond these traditional software issues, we preformed an SCA evaluation of both SM2DSA and SM2PKE in OpenSSL. This integration provides a rare opportunity to see how a straightforward implementation of an EC cryptosystem mixes with the underlying EC module for arithmetic. Our review resulted in the following observations, leveraging existing SCA results (Section 2) on the OpenSSL EC module.

- (1) For SM2DSA, in Listing 1 there is no scalar padding before calling `EC_POINT_mul`, suggesting an SM2DSA analogue of CVE-2011-1945 for remote timing attacks; see Section 4 for our empirical evaluation.
- (2) For SM2DSA, since there is no custom `EC_METHOD` for the SM2 curve, `EC_POINT_mul` is a wrapper to `ec_wNAF_mul`, suggesting an SM2DSA analogue for cache timing attacks

¹⁰The OpenSSL binaries can be roughly split in three blocks: `libcrypto`, providing the cryptographic and abstraction layer; `libssl`, providing the networking layer; `apps`, consisting in a CLI toolkit using the two libraries to perform various tasks.

```

84 k = BN_CTX_get(ctx);
85 rk = BN_CTX_get(ctx);
86 x1 = BN_CTX_get(ctx);
87 tmp = BN_CTX_get(ctx);
88
89 if (tmp == NULL)
90     goto done;
91
92 /* These values are returned and so should not be allocated out of the
93    ← context */
94 r = BN_new();
95 s = BN_new();
96
97 if (r == NULL || s == NULL)
98     goto done;
99
100 for (;;) {
101     BN_priv_rand_range(k, order);
102
103     if (EC_POINT_mul(group, kG, k, NULL, NULL, ctx) == 0)
104         goto done;
105
106     if (EC_POINT_get_affine_coordinates_GFp(group, kG, x1, NULL, ctx) == 0)
107         goto done;
108
109     if (BN_mod_add(r, e, x1, order, ctx) == 0)
110         goto done;
111
112     /* try again if r == 0 or r+k == n */
113     if (BN_is_zero(r))
114         continue;
115
116     BN_add(rk, r, k);
117
118     if (BN_cmp(rk, order) == 0)
119         continue;
120
121     BN_add(s, dA, BN_value_one());
122     BN_mod_inverse(s, s, order, ctx);
123
124     BN_mod_mul(tmp, dA, r, order, ctx);
125     BN_sub(tmp, k, tmp);
126
127     BN_mod_mul(s, s, tmp, order, ctx);
128
129     sig = ECDSA_SIG_new();
130
131     if (sig == NULL)
132         goto done;
133
134     /* takes ownership of r and s */
135     ECDSA_SIG_set0(sig, r, s);
136     break;
137 }

```

Listing 1: Source code from `crypto/sm2/sm2_sign.c` in OpenSSL 1.1.1-pre5 for SM2DSA signature generation.

```

270 C1 = EC_POINT_new(group);
271 if (C1 == NULL)
272     goto done;
273
274 if (EC_POINT_set_affine_coordinates_GFp
275     (group, C1, sm2_cctx->C1x, sm2_cctx->C1y, ctx) == 0)
276     goto done;
277
278 if (EC_POINT_mul(group, C1, NULL, C1, EC_KEY_get0_private_key(key), ctx) ==
279     0)
280     goto done;
281
282 if (EC_POINT_get_affine_coordinates_GFp(group, C1, x2, y2, ctx) == 0)
283     goto done;

```

Listing 2: Source code from `crypto/sm2/sm2_crypt.c` in OpenSSL 1.1.1-pre5 for SM2PKE decryption.

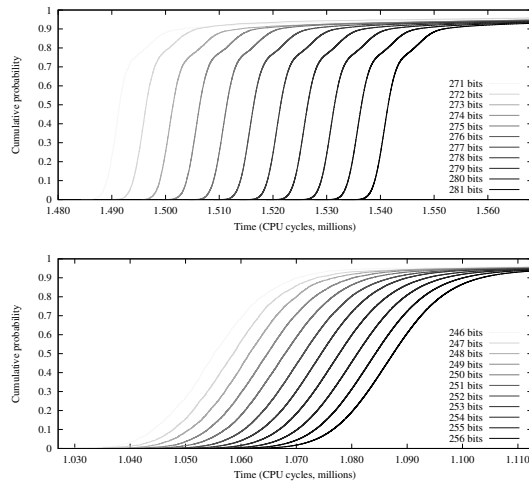


Figure 1: SM2DSA latency dependency on the nonce length on amd64 architecture in OpenSSL 1.1.1-pre3. Top: K-283 binary curve. Bottom: Recommended SM2 prime curve.

targeting scalar multiplication; see Section 5.1 for our empirical evaluation.

- (3) The SM2DSA implementation uses `BN_mod_inverse` without setting `BN_FLG_CONSTTIME`, suggesting an SM2DSA analogue for cache timing attacks targeting inversion via BEEA; see Section 5.2 for our empirical evaluation.
- (4) For SM2PKE, in Listing 2 there are no SCA considerations, suggesting (at least) DPA-style attacks on `EC_POINT_mul` during decryption; see Section 6 for our empirical evaluation.

The remainder of this paper is dedicated to evaluating these SCA leaks, proposing and implementing mitigations (Section 7), and empirical SCA evaluation of the mitigations (Section 7.3).

4 SM2DSA: REMOTE TIMINGS

We note the lack of scalar padding before calling `EC_POINT_mul`, suggesting an SM2DSA analogue of CVE-2011-1945. To evaluate the impact of this vulnerability, we correlate nonce lengths and the execution time of signature generations, adopting a process similar to the one presented by Brumley and Taveri [21].

We wrote an OpenSSL client application which repeatedly generates SM2DSA signatures for a given plaintext, under the same private key. For each generated signature, the program measures the execution time of the operation (in CPU cycles) and retrieves the associated nonce by monitoring the PRNG. We repeated the experiment using both the recommended SM2 prime curve and the standardized K-283 binary (Koblitz) curve [3], as the library executes two different code paths for `EC_POINT_mul` over prime and binary curves. We then analyzed the captured data to correlate the timings with the binary logarithm (bit-length) of the nonces.

We ran these experiments on a 4-cores/4-threads Intel Core i5-6500 CPU (Skylake) running at 3.2GHz, with Enhanced Intel

```

lfence
rdtsc
lfence
mov %rax, %r10
shr $31, %rdx
mov (%rsi, %rdx), %rax
lfence
rdtsc
shl $32, %rax
or %r10, %rax
mov %rax, (%rdi)
shr $31, %rdx
clflush (%rsi, %rdx)

```

Listing 3: FLUSH+RELOAD (left) and performance degradation (right) implemented for our cache-timing attacks.

SpeedStep Technology and Intel Turbo Boost Technology disabled. Figure 1 shows cumulative distribution functions (CDF) for different nonce bit-lengths for the two curves, collating 4 million samples for each curve. Both plots show a strong correlation between the bit-length of the nonce and the execution time of the signature generation, which in turn is distinctly dominated by the execution time of the underlying `EC_POINT_mul` operation. For no correlation, these curves should essentially be on top of each other, i.e. indistinguishable; see Section 7.3.

Generic binary curves. The top plot of Figure 1 shows that, using a generic binary curve as the underlying elliptic curve for SM2DSA, the timing correlation appears easily exploitable to mount a remote timing attack similar to [21]. For generic binary curves, OpenSSL implements the `EC_POINT_mul` operation through a Montgomery ladder algorithm, which due to its extreme regularity in the sequence of EC additions and doublings, results in an overall execution time directly proportional to the binary logarithm of the secret `EC_POINT_mul` scalar (i.e. the SM2DSA nonce). As a result, each nonce bit-length exhibits a clearly distinct CDF, and suggests simple thresholding on the execution time to filter signatures associated with a specific nonce length with high probability.

Recommended SM2 curve. When using the recommended SM2 prime curve, OpenSSL 1.1.1-pre3 implements the `EC_POINT_mul` operation using the generic prime curve codepath, using a wNAF algorithm (see Section 5.1). The bottom plot of Figure 1 shows that, similarly to the previous case, there is a strong correlation between the execution time of SM2DSA and the associated nonce length. We note that in this case, mounting a practical attack poses more challenges due to a less distinct separation between the different CDFs, likely compensated by collecting more samples.

5 SM2DSA: CACHE TIMINGS

As mentioned in Section 2.5, several previous works show SM2DSA vulnerable to ECDSA-type SCA attacks. For that reason, we explore and analyze the cryptosystem applying existing cache-timing attack techniques to code paths known for leaking information, and exploited successfully in the past for ECDSA [11, 65].

For our analysis, we use the FLUSH+RELOAD technique [75] paired with a performance degradation attack [9, 65]. Listing 3 shows code snippets used to implement both techniques. This combination of techniques allows us to accurately probe relevant memory addresses with enough granularity to confirm bit leakage on both scalar multiplication and modular inversion operations.

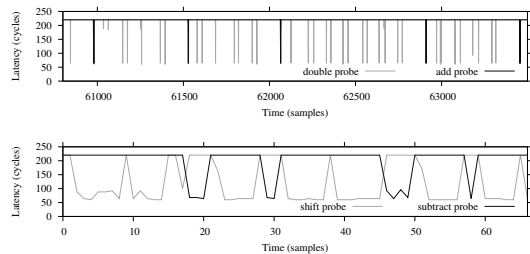


Figure 2: Partial raw cache-timing traces during SM2DSA. Top: Scalar multiplication. Bottom: Binary GCD modular inversion. Both traces reveal partial information on the secret scalar and the long-term private key, respectively.

5.1 Scalar Multiplication

SM2DSA in OpenSSL performs scalar multiplication operations by calling the `EC_POINT_mul` function in `SM2_sig_gen @ crypto-sm2/sm2_sign.c`, which is only a wrapper to the underlying `ec_wNAF_mul` function. The `ec_wNAF_mul` function is a generic code path performing scalar multiplication, i.e. $[k]G$ in SM2DSA, by executing a series of double and add operations based on the wNAF representation of k . This code path is vulnerable to cache-timing attacks due to its non constant-time execution, targeted previously using cache-timing techniques [9, 20, 70, 75]. Generally, the strategy is to trace the sequence of double and add operations, which leaks LSDs of k , leading to private key recovery.

Unlike previous attacks, during our analysis we do not probe memory lines directly used in functions `EC_POINT_add` and `EC_POINT_dbl`, but instead we focus in low level functions `BN_rshift1` and `BN_lshift`. The `BN_rshift1` function is one of several functions called during `EC_POINT_add` execution and, unlike the rest of the functions in the routine, `BN_rshift1` is a representative of the add operation. Similarly, `BN_lshift` is a representative of the double operation, allowing to identify add and double operations respectively during scalar multiplication. Therefore, these low level functions allow accurately detecting when add and double operations execute. By tracing the sequence of `BN_rshift1` and `BN_lshift` operations, we are able to determine with high accuracy the sequence of double and add operations, leaking LSDs of k . Top trace in Figure 2 shows a post-filtered cache-timing trace of a scalar multiplication with a random nonce k during SM2DSA. The probes detect the sequence of curve operations from left to right as follows: 1 double, 1 add, 4 doubles, 1 add, 4 doubles, 1 add, 7 doubles, 1 add, 4 doubles, and 1 add; thus revealing partial information on k .

5.2 Modular Inversion

Modular inversion is a common operation during digital signatures and in OpenSSL, SM2DSA uses the `BN_mod_inverse` function for this purpose. This function executes one of several GCD algorithm variants. Unfortunately, most of these variants are based on the Euclidean algorithm which executes in a non constant-time fashion. The Euclidean algorithm and variants are highly dependent on their

inputs and previous research exploits some of these variants [4, 8, 65].

During SM2DSA execution, none of the input values has the flag `BN_FLG_CONSTTIME` set when entering to the `BN_mod_inverse` function, therefore the function takes the default insecure path, calculating the modular inverse of $d_A + 1$ through the Binary Extended Euclidean Algorithm (BEEA). More importantly, this operation executes every time a signature is generated with the exact same input values, therefore an attacker has several opportunities to trace the BEEA execution on the private key.

Similar to the scalar multiplication case, we identify the low level operations leaking bits from the input values. In the BEEA case, this means functions `BN_rshift1` and `BN_sub`. By placing probes in memory lines in these routines, we are able to trace the sequence of shift and subtraction operations performed during modular inversion, leading to partial bit recovery of $d_A + 1$. Using algebraic methods [7], it is possible to recover a variable amount of private key LSDs from these sequences. Bottom trace in Figure 2 shows the end of a post-filtered cache-timing trace capturing the execution of the BEEA during SM2DSA. The trace matches the sequence 2 shifts, 1 subtract, 1 shift, 1 subtract, 2 shifts, 1 subtract, 1 shift, 1 subtract, 1 shift; obtained from a perfect trace computed by executing BEEA on the inputs taken from the SM2DSA signature test, demonstrating private key leakage.

6 SM2PKE: EM ANALYSIS

As discussed in Section 2.1, SM2PKE decryption computes the shared secret using the receiver’s private key d_B and the sender’s ECDHE public key C_1 . The point multiplication $[d_B]C_1$ can leak intermediate values which can be exploited using both vertical attacks [31, 35, 62] and horizontal attacks [10, 34] for key recovery.

To evaluate the side-channel leakage for SM2PKE, we applied Test Vector Leakage Analysis (TVLA) using Welch’s T-test [41, 67]. We took an approach similar to [28, 61] for ECC, with a reduced set of test vectors. We divided the test vectors into three different sets $\{S_i\}$ for $i = 1, 2, 3$. The sets S_1, S_2 and S_3 contained traces for fixed key d_B and fixed cipher text C_1 , fixed d_B and varying C_1 , fixed C_1 and varying d_B respectively. We performed the tests in pairs, such that $\{(S_1, S_2), (S_1, S_3)\}$ would fail the T-test if the resulting confidence threshold satisfies $|Cr| > 4.5$. We selected the value Cr (a function of number of samples) based upon empirical evidence from Jaffe et al. [47].

Experimental setup. We performed the experiments on an AM335x Sitara SoC¹¹ featuring a 32-bit ARM Cortex-A8 embedded on a BeagleBone Black¹² development board. We used the standard BeagleBone Debian distribution (“Wheezy” 7.8) while keeping all the default configurations intact. For capturing the EM traces, we used a Langer LF-U5 near-field probe (500kHz to 50MHz) and 30dB Langer PA-303 low noise amplifier. We positioned the probe head directly on the SoC, seeking to strengthen the acquisition quality. We procured the traces using a PicoScope 5244B digital oscilloscope at a sampling rate of 125 MSamples/sec with a 12-bit ADC resolution. Figure 3 shows our setup for the EM analysis.

¹¹<http://www.ti.com/processors/sitara/arm-cortex-a8/am335x/overview.html>

¹²<https://beagleboard.org/black>

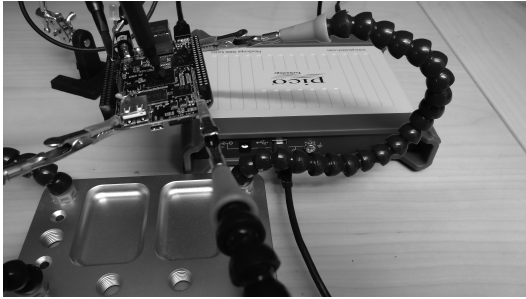


Figure 3: Capturing EM traces from the BeagleBone Black using a Langer probe positioned on the SoC and procured using the Picoscope USB oscilloscope.

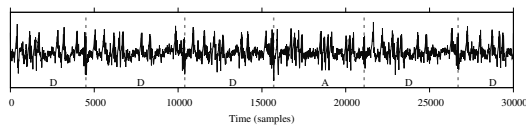


Figure 4: The filtered EM trace clearly reveals the sequence of ECC double and add operations during SM2PKE decryption.

EM acquisition. For the purpose of this analysis, we captured 1500 EM traces for each set (S_1, S_2, S_3) while performing the decryption operation. We fixed the clock frequency at 1GHz to avoid any bias in the captured traces. To acquire traces, we initially utilized the GPIO pin of the board to trigger the oscilloscope. However, this trigger proved unreliable as it encountered random delays. To improve this, we applied correlation based matching to locate the beginning of the trace. As most of the EM signal energy was concentrated at much lower frequencies, we also applied a Low Pass filter with a cut-off frequency at 15MHz.

Due to noise in the traces, we performed additional processing steps. For the envelope detection, we applied a Digital Hilbert Transform, followed by a Low Pass Filter to smooth out any high frequency noise. From the sets, we dropped traces containing noise due to preemptive interrupts and other unwanted signal features. In the end, we retained a total of 1000 traces per set. Since the T-test required averaging multiple traces, we aligned the traces at each point of interest (i.e. ECC operations). Figure 4 shows part of an actual processed EM trace, depicting a sequence of ECC double and add operations.

T-test. To validate the results, we divided each set into subsets $\{S_{1a}\}$ and $\{S_{1b}\}$ and performed an independent T-test between sets $\{(S_{1a}, S_{ka})\}$ and $\{(S_{1b}, S_{kb})\}$ for $k = 2, 3$. We performed a further test by combining an equal number of randomly selected traces from both $\{S_1\}$ and $\{S_k\}$ such that the two resulting subsets were disjoint. A correct T-test for the random sets $R_1 = \{(S_1 \cup S_k)\}$ and $R_k = \{(S_1 \cup S_k) - R_1\}$ should result in confidence threshold $|Cr| < 4.5$ for all the points in the traces.

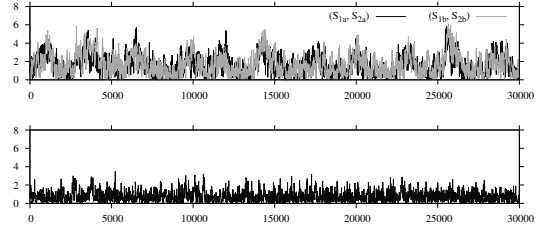


Figure 5: TVLA during SM2PKE decryption. Top: T-test results between sets S_1 and S_2 versus sample index; for fixed vs random k the test fails since many peaks exceed the 4.5 threshold for both sets. Bottom: T-test results between random sets R_1 and R_2 versus sample index; it shows no peaks exceeding 4.5 since the means are similar due to balanced and random selection of fixed and random k in both sets.

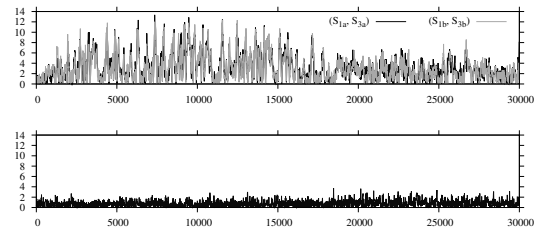


Figure 6: TVLA during SM2PKE decryption. Top: T-test results between sets S_1 and S_3 versus sample index; for fixed vs random C the test fails since many peaks exceed the 4.5 threshold for both sets. Bottom: T-test results between random sets R_1 and R_3 versus sample index; shows no peaks exceeding 4.5 since the means are similar due to equal and random selection of fixed and random C in both sets.

The experiments showed multiple points where the T-test failed for both $\{(S_1, S_2), (S_1, S_3)\}$. Figure 5 shows two T-test results for $\{(S_{1a}, S_{2a})\}$ and $\{(S_{1b}, S_{2b})\}$. It is clear from the figure that the T-test values have a significant number of peaks satisfying $|Cr| > 4.5$ for both tests, roughly at the same points. This demonstrates there is a strong leak at these points, since we performed both tests on different sets of traces. From the random sets $\{R_1, R_2\}$ the confidence threshold remains $|Cr| < 4.5$ which further validates our hypothesis. Similarly, Figure 6 shows the failed T-test for $\{(S_{1a}, S_{3a})\}$ and $\{(S_{1b}, S_{3b})\}$.

7 SCA MITIGATIONS

The attacker effort required to achieve full key recovery using the previously described leaks is very low. Taking the cache-timing leak in Section 5.1 as an example, SM2DSA lattice attacks discussed in Section 2.5 analogous to ECDSA would require roughly a mere 500 signatures with traces, immediately discarding roughly 75% of those that statistically will not reveal enough information about the LSDs (i.e. three or more bits are needed in practice for lattice attacks). In

this section, we describe our results on mitigating the discovered leaks. We claim no novelty for the mitigations themselves, only their application and implementation within the OpenSSL library; they are standard techniques known since at least the 90s.

We stress that the focus of our mitigation effort is not on SM2 nor any individual cryptosystem, but rather on the EC module itself, to provide transparent secure-by-default behavior to cryptosystems at the architecture level. That is, conceptually it should be completely reasonable to drop in a cryptosystem implementation like it was done with SM2DSA or SM2PKE and have it resist SCA, with absolutely no esoteric knowledge of OpenSSL internals that control SCA features such as constant-time flags.

7.1 Scalar Multiplication: SCA Mitigations

Ladder. While it is indeed feasible to reduce leakage in OpenSSL’s wNAF scalar multiplication code path [18], tediously straightlining conditions and making table lookups regular adds significant code complexity, increases the probability of defects, and generally results in low maintainability code. Even then, there is no guarantee that all leakage issues are addressed: the code path was not initially intended to resist SCA, and retrofitting mitigations becomes awkward.

We instead implemented an early exit from `ec_wNAF_mu1` that—irrespective of the constant time flag—diverts to a new single scalar multiplication function for all instances of $[k]G$ (fixed point, e.g. ECC key generation, SM2DSA signing, ECDSA signing, first half of ECDH) or $[k]P$ (variable point, e.g. SM2PKE decryption, last half of ECDH), and falls back to the existing (insecure) wNAF code in all other cases (e.g. $[a]G + [b]P$ in various digital signature scheme verifications). For cryptosystem use cases internal to the OpenSSL library, this provides secure-by-default scalar multiplication code path traversal.

For this new functionality, we chose the traditional powering ladder due to Montgomery [57], heralded for its favorable SCA properties. In modern implementations, straightlining the key-dependent ladder branches happens in one of two ways [60, Sec. 2]: “either by loading from (or storing to) addresses that depend on the secret scalar, or by using arithmetic operations to perform a conditional register-to-register move. The latter approach is very common on large processors with cache, where the former approach leaks through cache-timing information.”

We see both in practice: For example, `TomsFastMath`¹³ does not branch but reads and stores using (secret) pointer offsets, while `Mbed TLS`¹⁴ parses all the data and performs a manual conditional swap with arithmetic, even documenting their function `mbedtls1s_mpi_safe_cond_swap` with the comment: “Here it is not OK to simply swap the pointers, which would lead to different memory access patterns when X and Y are used afterwards.” This is in contrast to e.g. [44, Sec. 8.5]: “we implement the conditional swap operation after each ladder step by swapping pointer variables instead of data. We expect slightly better performance and also a reduced side-channel leakage.” While that is perhaps a valid strategy on

architectures lacking cache memory, we feel it is generally dubious advice since typical engineers are usually unaware of SCA subtleties.

Regardless, the “standard way” according to Bernstein [14, Sec. 3] uses arithmetic to implement conditional swaps on the data, not the pointers; the work also reviews a slight optimization, which we also implement. The two contiguous swaps conditional on bits k_i and k_{i-1} reduce to a single swap by XOR-merging the condition bits, i.e. only swap if the bit values differ. This optimization halves the number of conditional swaps.

Scalar padding. The above conditional swaps ensure favorable SCA behavior for ladder iterations. But [21] exploits the number of said iterations, fixed in an ECDSA-only fashion in 2011 by padding nonces. We remove this padding, and instead push it to the underlying EC module to ensure a constant number of ladder iterations. To accomplish this in an SCA-friendly way, we construct two values $k' = k+n$ and $k'' = k'+n$, subsequently using the above conditional swap to set k to either k' or k'' , whichever has bit-length precisely one more than n . We apply this padding directly preceding ladder execution.

Coordinate blinding. Originally proposed by Coron [31, Sec. 5.3] for standard projective coordinates as a DPA countermeasure, coordinate blinding transforms the input point to a random representative of the equivalence class. For generic curves over $GF(p)$, OpenSSL’s formulae are a fairly verbatim implementation of Jacobian projective coordinates [1, A.9.6] where the relation

$$(X, Y, Z) \equiv (\lambda^2 X, \lambda^3 Y, \lambda Z)$$

holds for all $\lambda \neq 0$ in $GF(p)$. Our implemented mitigation generates λ randomly, applying the map a single time directly preceding the ladder execution. This is, for example, the approach taken by `Mbed TLS` (function `ecp_randomize_jac`).

7.2 Modular Inversion: SCA Mitigations

Directly due to the work by Gueron and Krasnov [43, Sec. 6], OpenSSL integrated a contribution from Intel that included (1) high-speed, constant-time P-256 ECC on AVX2 architectures; (2) constant-time modular inversion modulo ECDSA group orders. It did the latter by internally exposing a function pointer within the `EC_METHOD` structure. If set, ECDSA signing code path calls said pointer (for which the custom P-256 method has a dedicated function), otherwise a series of default fallbacks including (1) FLT inversion with Montgomery modular exponentiation; (2) normal EEA-based inversion. We refactored the structure to expose this default behavior within the wrapper that checks the function pointer, the end goal being to expose it to the EC module as a whole and not limit to ECDSA, in turn allowing SM2DSA access to a strictly secure-by-default functionality. We explored two different options for inversion default behavior that resist SCA, summarized below.

Blinding. The classical way to compute modular inversions is through the EEA utilizing divisions, or binary variants utilizing shifts and subtracts. However, as previously described their control flow can leak critical algorithm state. Nevertheless, to prevent direct input deduction from this state one option is to choose blinding value b uniformly at random from $[1..n)$ then compute $k^{-1} = b(bk)^{-1}$ at

¹³<https://github.com/libtom/tomsfastmath/>

¹⁴<https://github.com/ARMmbed/mbedtls/>

the additional cost of two multiplications. This is, for example, the approach taken by Mbed TLS for ECDSA nonces.

Exponentiation. Although initially motivated by binary fields with normal basis representation where squaring is a simple bit rotation, the algorithm by Itoh and Tsujii [46] is one of the earliest examples of favorable implementation aspects of using FLT for finite field inversion. SCA benefits followed thereafter, e.g. Curve25519 where Bernstein [13, Sec. 5] weighs blinded EEA methods versus FLT: “An extended-Euclid inversion, randomized to protect against timing attacks, might be faster, but the maximum potential speedup is very small, while the cost in code complexity is large.”

Performance and security. Regarding security, it is clear that either method is a leap forward for OpenSSL with respect to secure-by-default. We feel that blinding has an intrinsic advantage over FLT-based methods, since the former resists bug attacks [15, 19] that exploit predictable execution flows. Regarding performance, we benchmarked both approaches to measure the potential differences alluded to by Bernstein, and found the results consistent. On an Intel Core i5-6500 CPU (Skylake) running at 3.2GHz, after all of our described and implemented countermeasures, one SM2DSA execution takes on average 1760913 cycles with FLT, and 1750984 cycles with blinded BEEA—a difference of a fraction of a percent.

In the end, the OpenSSL team declined our blinding contribution. They plan to increase the usage of the Montgomery arithmetic context within the EC module, so in that sense their decision is rational from a software architecture perspective. The team instead integrated our FLT refactoring, sufficient to thwart the attack in Section 5, and furthermore provide secure-by-default behavior to future callers conforming to the convention set by this API.

7.3 SCA Mitigations: Evaluation

Remote timings: evaluation. Using the same approach adopted in Section 4, Figure 7 shows the cumulative effect of three countermeasures: adopting the Montgomery ladder instead of the wNAF algorithm for regular scalar multiplication, scalar padding, and computation of modular inversion via exponentiation through FLT.

Both plots clearly show that the latencies measured for signature generation using nonces of different bit-lengths are indistinguishable, effectively preventing the attack, and a comparison with Figure 1 immediately shows the extent of the leakage reduction.

Cache-timings: evaluation. After introducing the mitigations, when SM2DSA performs a scalar multiplication it first calls the `EC_POINT_mul` function, a wrapper to `ec_wNAF_mul`. There the code takes an early exit, jumping to the powering ladder regular algorithm to perform a fixed point scalar multiplication $[k]G$. From the cache perspective, the ladder implementation consists of an always-double-and-add algorithm, largely unrelated to the wNAF representation of the nonce k . To support our claim, we follow the same approach as in Section 5, placing probes in the same underlying functions `BN_rshift1` and `BN_lshift`—called by `EC_POINT_add` and `EC_POINTdbl`—to trace the sequence of operations during scalar multiplication. Top trace in Figure 8 shows an example trace, which indeed tracks the sequence of double and add operations successfully, but due to the regular nature of the powering ladder

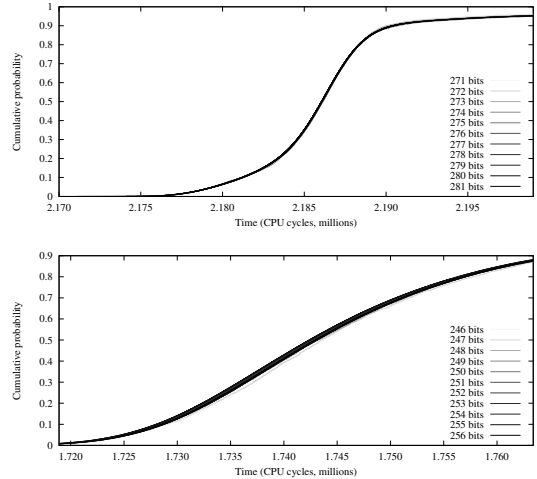


Figure 7: SM2DSA latency dependency on the nonce length on amd64 architecture, using (1) a Montgomery ladder algorithm for scalar multiplication instead of wNAF; (2) scalar padding; (3) modular inversion through exponentiation (FLT). Top: K-283 binary curve. Bottom: Recommended SM2 curve.

algorithm, no meaningful information can be retrieved from this sequence.

During modular inversion, the high level function `SM2_sig_gen` in SM2DSA no longer calls `BN_mod_inverse` but instead it calls directly `EC_GROUP_do_inverse_ord` on the private key $d_A + 1$. This function computes modular inversion by performing an exponentiation using FLT, therefore the underlying algorithm and its implementation are completely different compared to the Euclidean algorithm (and variants) used previously. Recall that during modular inversion using FLT, the exponent value is public; said value does not require SCA protection. Bottom trace in Figure 8 shows an example trace during modular inversion, probing the square and multiply operations based on the public exponent.

EM leakage: evaluation. To validate the efficacy of the applied mitigations, we repeated the T-test experiments (Section 6). Figure 10 shows the results of the new T-test for both fixed vs random key (S_1, S_2) and fixed vs random point (S_1, S_3). Figure 9 shows the EM traces, reflecting a regular sequence of ECC double and add operations due to ladder point multiplication. One interesting observation is the increase in the number of peaks for the add operation compared to Figure 4. This is due to the fact that `ec_wNAF_mul` uses mixed coordinates (projective and affine), a code path with less field operations compared to the fully projective coordinate path taken by the ladder.

It is clear from Figure 10 that the T-test shows a significant improvement due to the combined ladder application and projective coordinate randomization. The T-test easily passed for fixed vs random point (S_1, S_3) with $|Cr| < 4$. In case of fixed vs random key

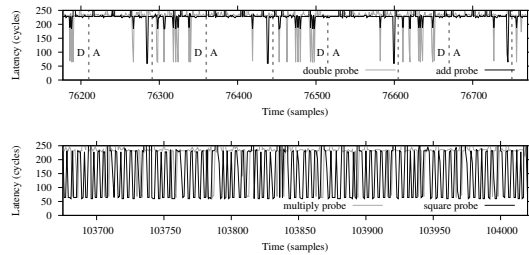


Figure 8: Partial raw cache-timing traces during SM2DSA. Top: Ladder scalar multiplication composed of regular double and add operations. Bottom: FLT modular inversion via exponentiation composed of regular squaring windows followed by a single multiply.

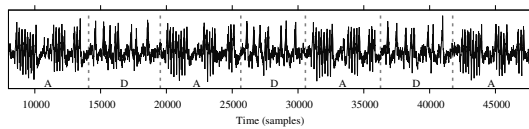


Figure 9: The filtered EM trace after applying the ladder countermeasure. As expected, it clearly reveals the sequence of ECC double and add operations during SM2PKE decryption, yet this sequence is regular and not useful for SCA-enabled attackers.

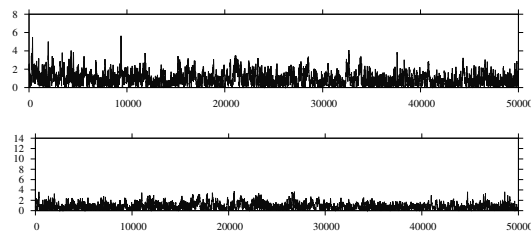


Figure 10: Top: T-test results between sets S_1 and S_2 versus sample index; for fixed vs random k the test marginally fails with leaks at the few points where the threshold is around 6. Bottom: T-test results between sets S_1 and S_3 versus sample index; for fixed vs random C the test passes since no peaks exceed the 4.5 threshold.

(S_1, S_2) we still observe a marginal number of peaks with magnitude roughly 6. In theory, it is still possible to exploit this; e.g. a key value that leading to special intermediate points on the curve such as zero-value [6] or same value points [59]. However, the leakage is so minimal, our analysis suggests mounting such attacks would be extremely difficult and feature significant data complexity. Moreover, the scalar randomization countermeasure [31] to thwart this leak introduces performance overhead, in this case unacceptable to OpenSSL when weighed vs risk.

8 CONCLUSION

Subsequent to an accelerated OpenSSL milestone to support SM2 cryptosystems, our work began with a security review of SM2DSA and SM2PKE implementations within OpenSSL pre-releases. Part of our review uncovered several side-channel deficiencies in the merged code, which we then verified with empirical remote timing, cache-timing, and EM traces. To mitigate these discovered vulnerabilities, we proposed and implemented several mitigations, now mainlined into the OpenSSL codebase. These mitigations target the underlying EC module, providing secure-by-default behavior not only for SM2 but future cryptosystems in the ECC family. Notably, the mitigations also bring security to the generic curve scalar multiplication code path in OpenSSL, a longstanding vulnerability since 2009. Finally, we performed an empirical SCA evaluation of these mitigations to demonstrate their efficacy.

We met our goal to intersect the recent OpenSSL 1.1.1 release and ensure these vulnerabilities do not affect release versions. However, given a more relaxed schedule, we outline future work to improve this secure-by-default approach: (1) the antiquated ECC point addition and doubling formulae should be renovated to more recent exception and/or branch-free versions; (2) support for ladder step function pointers, for more efficient ladder operations w.r.t. finite field operations; (3) at the standardization level, SM2DSA private key formats that, similar to RSA private keys with CRT parameters, store the value $(d_A + 1)^{-1}$ alongside the private key d_A for accelerated performance and a reduced SCA attack surface.

From the software engineering perspective, lessons learned from our work are twofold: (1) software projects, OpenSSL included, should maintain a stronger separation between release, beta, and development branches to inhibit “feeping creaturism” [69, Ch. 6] that can adversely shift milestones; (2) milestones for security-critical features should be set consistent with the complexity of the review process, to prevent premature merging. Luckily, in this case our responsible disclosure with the OpenSSL security team coupled with our mitigation development efforts yielded a favorable outcome. We strongly encourage adhering to the above two points to assist averting future security vulnerabilities.

ACKNOWLEDGMENTS

Supported in part by Academy of Finland grant 303814.

The third author was supported in part by a Nokia Foundation Scholarship and by the Pekka Ahonen Fund through the Industrial Research Fund of Tampere University of Technology.

This article is based in part upon work from COST Action IC1403 CRYPTACUS, supported by COST (European Cooperation in Science and Technology).

REFERENCES

- [1] 1999. *Standard Specifications for Public Key Cryptography*. IEEE P1363/D13. Institute of Electrical and Electronics Engineers.
- [2] 2009. *Elliptic Curve Cryptography*. SEC 1. Standards for Efficient Cryptography Group. <http://www.secg.org/sec1-v2.pdf>
- [3] 2013. *Digital Signature Standard (DSS)*. FIPS PUB 186-4. National Institute of Standards and Technology. <https://doi.org/10.6028/NIST.FIPS.186-4>
- [4] Onur Acicmez, Shay Gueron, and Jean-Pierre Seifert. 2007. New Branch Prediction Vulnerabilities in OpenSSL and Necessary Software Countermeasures. In *Cryptography and Coding, 11th IMA International Conference, Cirencester, UK, December 18-20, 2007, Proceedings (Lecture Notes in Computer Science)*.

- Steven D. Galbraith (Ed.), Vol. 4887. Springer, 185–203. https://doi.org/10.1007/978-3-540-77272-9_12
- [5] Onur Acıçimez, Werner Schindler, and Çetin Kaya Koç. 2005. Improving Brumley and Boneh timing attack on unprotected SSL implementations. In *Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS 2005, Alexandria, VA, USA, November 7–11, 2005*, Vijay Atluri, Catherine A. Meadows, and Ari Juels (Eds.). ACM, 139–146. <https://doi.org/10.1145/1102120.1102140>
- [6] Toru Akishita and Tsuyoshi Takagi. 2005. Zero-Value Register Attack on Elliptic Curve Cryptosystem. *IEICE Transactions* 88-A, 1 (2005), 132–139. <https://doi.org/10.1093/ietfiec/88-a-1.132>
- [7] Alejandro Cabrera Aldaya, Alejandro J. Cabrera Sarmiento, and Santiago Sánchez-Solano. 2017. SPA vulnerabilities of the binary extended Euclidean algorithm. *J. Cryptographic Engineering* 7, 4 (2017), 273–285. <https://doi.org/10.1007/s13389-016-0135-4>
- [8] Alejandro Cabrera Aldaya, Cesar Pereida Garcia, Luis Manuel Alvarez Tapia, and Billy Bob Brumley. 2018. Cache-Timing Attacks on RSA Key Generation. *IACR Cryptology ePrint Archive* 2018, 367 (2018). <https://eprint.iacr.org/2018/367>
- [9] Thomas Allan, Billy Bob Brumley, Katrina E. Falkner, Joop van de Pol, and Yuval Yarom. 2016. Amplifying side channels through performance degradation. In *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5–9, 2016*, Stephen Schwab, William K. Robertson, and Davide Balzarotti (Eds.). ACM, 422–435. <https://doi.org/10.1145/2991079.2991084>
- [10] Aurélie Bauer, Éliane Jaumes, Emmanuel Prouff, Jean-René Reinhard, and Justine Wild. 2015. Horizontal collision correlation attack on elliptic curves – Extended Version. *Cryptography and Communications* 7, 1 (2015), 91–119. <https://doi.org/10.1007/s12095-014-0111-8>
- [11] Naomi Bengier, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. 2014. “Ooh Aah... Just a Little Bit”: A Small Amount of Side Channel Can Go a Long Way. In *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23–26, 2014. Proceedings (Lecture Notes in Computer Science)*, Lejla Batina and Matthew Robshaw (Eds.), Vol. 8731. Springer, 75–92. https://doi.org/10.1007/978-3-662-44709-3_5
- [12] Daniel J. Bernstein. 2005. Cache-timing attacks on AES. <http://cr.ypt/papers.html#cachetiming>
- [13] Daniel J. Bernstein. 2006. Curve25519: New Diffie-Hellman Speed Records. In *Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, April 24–26, 2006, Proceedings (Lecture Notes in Computer Science)*, Moti Yung, Yevgeniy Dodis, Angelos Kiayias, and Tal Malkin (Eds.), Vol. 3958. Springer, 207–228. https://doi.org/10.1007/11745853_14
- [14] Daniel J. Bernstein. 2009. Batch Binary Edwards. In *Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16–20, 2009. Proceedings (Lecture Notes in Computer Science)*, Shai Halevi (Ed.), Vol. 5677. Springer, 317–336. https://doi.org/10.1007/978-3-642-03356-8_19
- [15] Eli Biham, Yaniv Carmeli, and Adi Shamir. 2016. Bug Attacks. *J. Cryptology* 29, 4 (2016), 775–805. <https://doi.org/10.1007/s00145-015-9209-1>
- [16] Daniel Bleichenbacher. 1998. Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1. In *Advances in Cryptology - CRYPTO '98, 18th Annual International Cryptology Conference, Santa Barbara, California, USA, August 23–27, 1998. Proceedings (Lecture Notes in Computer Science)*, Hugo Krawczyk (Ed.), Vol. 1462. Springer, 1–12. <https://doi.org/10.1007/BFb0055716>
- [17] Eric Brier, Christophe Clavier, and Francis Olivier. 2004. Correlation Power Analysis with a Leakage Model. In *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11–13, 2004. Proceedings (Lecture Notes in Computer Science)*, Marc Joye and Jean-Jacques Quisquater (Eds.), Vol. 3156. Springer, 16–29. https://doi.org/10.1007/978-3-540-28632-5_2
- [18] Billy Bob Brumley. 2015. Faster Software for Fast Endomorphisms. In *Constructive Side-Channel Analysis and Secure Design - 6th International Workshop, COSADE 2015, Berlin, Germany, April 13–14, 2015. Revised Selected Papers (Lecture Notes in Computer Science)*, Stefan Mangard and Axel Y. Poschmann (Eds.), Vol. 9064. Springer, 127–140. https://doi.org/10.1007/978-3-319-21476-4_9
- [19] Billy Bob Brumley, Manuel Barbosa, Dan Page, and Frederik Vercauteren. 2012. Practical Realisation and Elimination of an ECC-Related Software Bug Attack. In *Topics in Cryptology - CT-RSA 2012 - The Cryptographers' Track at the RSA Conference 2012, San Francisco, CA, USA, February 27 - March 2, 2012. Proceedings (Lecture Notes in Computer Science)*, Orr Dunkelman (Ed.), Vol. 7178. Springer, 171–186. https://doi.org/10.1007/978-3-642-27954-6_11
- [20] Billy Bob Brumley and Risto M. Hakala. 2009. Cache-Timing Template Attacks. In *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6–10, 2009. Proceedings (Lecture Notes in Computer Science)*, Mitsuru Matsui (Ed.), Vol. 5912. Springer, 667–684. https://doi.org/10.1007/978-3-642-10366-7_39
- [21] Billy Bob Brumley and Nicola Tuveri. 2011. Remote Timing Attacks Are Still Practical. In *Computer Security - ESORICS 2011 - 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12–14, 2011. Proceedings (Lecture Notes in Computer Science)*, Vijay Atluri and Claudia Diaz (Eds.), Vol. 6879. Springer, 355–371. https://doi.org/10.1007/978-3-642-23822-2_20
- [22] David Brumley and Dan Boneh. 2003. Remote Timing Attacks Are Practical. In *Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA, August 4–8, 2003*. USENIX Association. <https://www.usenix.org/conference/12th-usenix-security-symposium/remote-timing-attacks-are-practical>
- [23] David Brumley and Dan Boneh. 2005. Remote timing attacks are practical. *Computer Networks* 48, 5 (2005), 701–716. <https://doi.org/10.1016/j.comnet.2005.01.010>
- [24] Certicom Research. 2010. *Standards for Efficient Cryptography 2 (SEC 2): Recommended Elliptic Curve Domain Parameters (Version 2.0)*. Technical Report. Certicom Corp. <http://www.secg.org/sec2-v2.pdf>
- [25] Suresh Chari, Jossyula R. Rao, and Pankaj Rohatgi. 2002. Template Attacks. In *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13–15, 2002. Revised Papers (Lecture Notes in Computer Science)*, Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar (Eds.), Vol. 2523. Springer, 13–28. https://doi.org/10.1007/3-540-36400-5_3
- [26] Cai-Sen Chen, Tao Wang, and Jun-Jian Tian. 2013. Improving timing attack on RSA-CRT via error detection and correction strategy. *Information Sciences* 232 (2013), 464–474. <https://doi.org/10.1016/j.ins.2012.01.027>
- [27] Jiazhe Chen, Mingjie Liu, Hexin Li, and Hongsong Shi. 2015. Mind Your Nonces Moving: Template-Based Partially-Sharing Nonces Attack on SM2 Digital Signature Algorithm. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '15, Singapore, April 14–17, 2015*, Feng Bao, Steven Miller, Jianying Zhou, and Gail-Joon Ahn (Eds.), ACM, 609–614. <https://doi.org/10.1145/2714576.2714587>
- [28] Łukasz Chmielewski, Pedro Massolino, Jo Vliegen, Lejla Batina, and Nele Mentens. 2017. Completing the Complete ECC Formulae with Countermeasures. *Journal of Low Power Electronics and Applications* 7, 1 (2017), 3. <https://doi.org/10.3390/jlpea7010003>
- [29] Tom Chothia and Apratim Guha. 2011. A Statistical Test for Information Leaks Using Continuous Mutual Information. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011, Cernay-la-Ville, France, 27–29 June, 2011*. IEEE Computer Society, 177–190. <https://doi.org/10.1109/CSF.2011.19>
- [30] Christophe Clavier, Benoit Felix, Georges Gagnerot, Mylène Roussellet, and Vincent Verneuil. 2010. Horizontal Correlation Analysis on Exponentiation. In *Information and Communications Security - 12th International Conference, ICICS 2010, Barcelona, Spain, December 15–17, 2010. Proceedings (Lecture Notes in Computer Science)*, Miguel Soriano, Sihang Qing, and Javier López (Eds.), Vol. 6476. Springer, 46–61. https://doi.org/10.1007/978-3-642-17650-0_5
- [31] Jean-Sébastien Coron. 1999. Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems. In *Cryptographic Hardware and Embedded Systems, First International Workshop, CHES '99, Worcester, MA, USA, August 12–13, 1999. Proceedings (Lecture Notes in Computer Science)*, Çetin Kaya Koç and Christof Paar (Eds.), Vol. 1717. Springer, 292–302. https://doi.org/10.1007/3-540-48059-5_25
- [32] Scott A. Crosby, Dan S. Wallach, and Rudolf H. Riedi. 2009. Opportunities and Limits of Remote Timing Attacks. *ACM Transactions on Information and System Security (TISSEC)* 12, 3 (2009), 17:1–17:29. <https://doi.org/10.1145/1455526.1455530>
- [33] T. Dierks and C. Allen. 1999. *The TLS Protocol Version 1.0*. RFC 2246. RFC Editor. <https://doi.org/10.17487/RFC2246>
- [34] Margaux Dugardin, Louiza Papachristodoulou, Zakaria Najm, Lejla Batina, Jean-Luc Danger, and Sylvain Guilley. 2016. Dismantling Real-World ECC with Horizontal and Vertical Template Attacks. In *Constructive Side-Channel Analysis and Secure Design - 7th International Workshop, COSADE 2016, Graz, Austria, April 14–15, 2016. Revised Selected Papers (Lecture Notes in Computer Science)*, François-Xavier Standaert and Elisabeth Oswald (Eds.), Vol. 9689. Springer, 88–108. https://doi.org/10.1007/978-3-319-43283-0_6
- [35] Pierre-Alain Fouque and Frédéric Valette. 2003. The Doubling Attack - Why Upwards Is Better than Downwards. In *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8–10, 2003. Proceedings (Lecture Notes in Computer Science)*, Colin D. Walter, Çetin Kaya Koç, and Christof Paar (Eds.), Vol. 2779. Springer, 269–280. https://doi.org/10.1007/978-3-540-45238-6_22
- [36] Daniel Genkin, Lev Pachmanov, Itamar Pipman, and Eran Tromer. 2016. ECDH Key-Extraction via Low-Bandwidth Electromagnetic Attacks on PCs. In *Topics in Cryptology - CT-RSA 2016 - The Cryptographers' Track at the RSA Conference 2016, San Francisco, CA, USA, February 29 - March 4, 2016. Proceedings (Lecture Notes in Computer Science)*, Kazuo Sako (Ed.), Vol. 9610. Springer, 219–235. https://doi.org/10.1007/978-3-319-29485-8_13
- [37] Daniel Genkin, Lev Pachmanov, Itamar Pipman, Eran Tromer, and Yuval Yarom. 2016. ECDSA Key Extraction from Mobile Devices via Nonintrusive Physical Side Channels. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24–28, 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM, 1626–1638. <https://doi.org/10.1145/2976749.2978353>

- [38] Daniel Genkin, Adi Shamir, and Eran Tromer. 2014. RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis. In *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference*, Santa Barbara, CA, USA, August 17–21, 2014, *Proceedings, Part I (Lecture Notes in Computer Science)*, Juan A. Garay and Rosario Gennaro (Eds.), Vol. 8616. Springer, 444–461. https://doi.org/10.1007/978-3-662-44371-2_25
- [39] Daniel Genkin, Luke Valenta, and Yuval Yarom. 2017. May the Fourth Be With You: A Microarchitectural Side Channel Attack on Several Real-World Applications of Curve25519. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.), ACM, 845–858. <https://doi.org/10.1145/3133956.3134029>
- [40] Gabriel Goller and Georg Sigl. 2015. Side Channel Attacks on Smartphones and Embedded Devices Using Standard Radio Equipment. In *Constructive Side-Channel Analysis and Secure Design - 6th International Workshop, COSADE 2015, Berlin, Germany, April 13-14, 2015. Revised Selected Papers (Lecture Notes in Computer Science)*, Stefan Mangard and Axel Y. Poschmann (Eds.), Vol. 9064. Springer, 255–270. https://doi.org/10.1007/978-3-319-21476-4_17
- [41] Gilbert Goodwill, Benjamin Jun, Josh Jaffe, and Pankaj Rohatgi. 2011. A testing methodology for side-channel resistance validation. In *Non-Invasive Attack Testing Workshop, NIAT 2011, Nara, Japan, September 26-27, 2011. Proceedings*. NIST. https://csrc.nist.gov/csrc/media/events/non-invasive-attack-testing-workshop/documents/08_goodwill.pdf
- [42] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, Jaeyeon Jung and Thorsten Holz (Eds.), USENIX Association, 897–912. <https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/gruss>
- [43] Shay Gueron and Vlad Krasnov. 2015. Fast prime field elliptic-curve cryptography with 256-bit primes. *J. Cryptographic Engineering* 5, 2 (2015), 141–151. <https://doi.org/10.1007/s13389-014-0090-x>
- [44] Björn Haase and Benoît Labrique. 2017. Making Password Authenticated Key Exchange Suitable for Resource-Constrained Industrial Control Devices. In *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings (Lecture Notes in Computer Science)*, Wieland Fischer and Naofumi Homma (Eds.), Vol. 10529. Springer, 346–364. https://doi.org/10.1007/978-3-319-66787-4_17
- [45] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical Timing Side Channel Attacks Against Kernel Space ASLR. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*. The Internet Society. <https://www.ndss-symposium.org/ndss2013/practical-timing-side-channel-attacks-against-kernel-space-aslr>
- [46] Toshiya Itoh and Shigeo Tsujii. 1988. A fast algorithm for computing multiplicative inverses in $GF(2^m)$ using normal bases. *Inform. and Comput.* 78, 3 (1988), 171–177. [https://doi.org/10.1016/0890-5401\(88\)90024-7](https://doi.org/10.1016/0890-5401(88)90024-7)
- [47] Josh Jaffe, Pankaj Rohatgi, and Marc Wittenman. 2011. Efficient side-channel testing for public key algorithms: RSA case study. In *Non-Invasive Attack Testing Workshop, NIAT 2011, Nara, Japan, September 26-27, 2011. Proceedings*. NIST. https://csrc.nist.gov/CSRC/media/Events/Non-Invasive-Attack-Testing-Workshop/documents/09_jaffe.pdf
- [48] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy, SP 2019, Proceedings, 20-29 May 2019, San Francisco, California, USA*. IEEE, 19–37. <https://doi.org/10.1109/SP.2019.00002>
- [49] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings (Lecture Notes in Computer Science)*, Neal Koblitz (Ed.), Vol. 1109. Springer, 104–113. https://doi.org/10.1007/3-540-68697-5_9
- [50] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential Power Analysis. In *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings (Lecture Notes in Computer Science)*, Michael J. Wiener (Ed.), Vol. 1666. Springer, 388–397. https://doi.org/10.1007/3-540-48405-1_25
- [51] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, Thorsten Holz and Stefan Savage (Eds.), USENIX Association, 549–564. <https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/lipp>
- [52] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Melttdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018, William Enck and Adrienne Porter Felt (Eds.)*, USENIX Association, 973–990. <https://www.usenix.org/conference/usenixsecurity18/presentation/lipp>
- [53] Mingjie Liu, Jiazhe Chen, and Hexin Li. 2013. Partially Known Nonces and Fault Injection Attacks on SM2 Signature Algorithm. In *Information Security and Cryptology - 9th International Conference, Inscrypt 2013, Guangzhou, China, November 27-30, 2013, Revised Selected Papers (Lecture Notes in Computer Science)*, Dongdai Lin, Shouhuai Xu, and Moti Yung (Eds.), Vol. 8567. Springer, 343–358. https://doi.org/10.1007/978-3-319-12087-4_22
- [54] Jake Longo, Elke De Mulder, Dan Page, and Michael Tunstall. 2015. SoC It to EM: ElectroMagnetic Side-Channel Attacks on a Complex System-on-Chip. In *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings (Lecture Notes in Computer Science)*, Tim Güneysu and Helena Handschuh (Eds.), Vol. 9293. Springer, 620–640. https://doi.org/10.1007/978-3-662-48324-4_31
- [55] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. 2015. C5: Cross-Cores Cache Covert Channel. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 12th International Conference, DIMVA 2015, Milan, Italy, July 9-10, 2015, Proceedings (Lecture Notes in Computer Science)*, Magnus Almgren, Vincenzo Gulisano, and Federico Maggi (Eds.), Vol. 9148. Springer, 46–64. https://doi.org/10.1007/978-3-319-20550-2_3
- [56] Christopher Meyer, Juraj Somorovsky, Eugen Weiss, Jörg Schwenk, Sebastian Schinzel, and Erik Tews. 2014. Revisiting SSL/TLS Implementations: New Bleichenbacher Side Channels and Attacks. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, Kevin Fu and Jaeyeon Jung (Eds.), USENIX Association, 735–748. <https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/meyer>
- [57] Peter L. Montgomery. 1987. Speeding the Pollard and elliptic curve methods of factorization. *Math. Comp.* 48, 177 (1987), 243–264. <https://doi.org/10.2307/2007888>
- [58] Amir Moradi, Bastian Richter, Tobias Schneider, and François-Xavier Standaert. 2018. Leakage Detection with the χ^2 -Test. *LACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018, 1 (2018), 209–237. <https://doi.org/10.13154/tches.v2018i1.209-237>
- [59] Cédric Murdica, Sylvain Guilley, Jean-Luc Danger, Philippe Hoogvorst, and David Naccache. 2012. Same Values Power Analysis Using Special Points on Elliptic Curves. In *Constructive Side-Channel Analysis and Secure Design - Third International Workshop, COSADE 2012, Darmstadt, Germany, May 3-4, 2012. Proceedings (Lecture Notes in Computer Science)*, Werner Schindler and Sorin A. Huss (Eds.), Vol. 7275. Springer, 183–198. https://doi.org/10.1007/978-3-642-29912-4_14
- [60] Erick Nascimento, Lukasz Chmielewski, David Oswald, and Peter Schwabe. 2016. Attacking Embedded ECC Implementations Through cmov Side Channels. In *Selected Areas in Cryptography - SAC 2016 - 23rd International Conference, St. John's, NL, Canada, August 10-12, 2016, Revised Selected Papers (Lecture Notes in Computer Science)*, Roberto Avanzi and Howard M. Heys (Eds.), Vol. 10532. Springer, 99–119. https://doi.org/10.1007/978-3-319-69453-5_6
- [61] Erick Nascimento, Julio López, and Ricardo Dahab. 2015. Efficient and Secure Elliptic Curve Cryptography for 8-bit AVR Microcontrollers. In *Security, Privacy, and Applied Cryptography Engineering - 5th International Conference, SPACE 2015, Jaipur, India, October 3-7, 2015, Proceedings (Lecture Notes in Computer Science)*, Rajat Subhra Chakraborty, Peter Schwabe, and Jon A. Solworth (Eds.), Vol. 9354. Springer, 289–309. https://doi.org/10.1007/978-3-319-21426-5_17
- [62] Katsuyuki Okeya and Kouichi Sakurai. 2002. A Second-Order DPA Attack Breaks a Window-Method Based Countermeasure against Side Channel Attacks. In *Information Security, 5th International Conference, ISC 2002 Sao Paulo, Brazil, September 30 - October 2, 2002, Proceedings (Lecture Notes in Computer Science)*, Agnes Hui Chan and Virgil D. Gligor (Eds.), Vol. 2433. Springer, 389–401. https://doi.org/10.1007/3-540-45811-5_30
- [63] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings (Lecture Notes in Computer Science)*, David Pointcheval (Ed.), Vol. 3860. Springer, 1–20. https://doi.org/10.1007/11605805_1
- [64] Colin Percival. 2005. Cache Missing for Fun and Profit. In *BSDCan 2005, Ottawa, Canada, May 13-14, 2005, Proceedings*. <http://www.daemonology.net/papers/cachemissing.pdf>
- [65] Cesar Pereda Garcia and Billy Bob Brumley. 2017. Constant-Time Calleees with Variable-Time Callers. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, Engin Kirda and Thomas Ristenpart (Eds.), USENIX Association, 83–98. <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/garcia>
- [66] Jean-Jacques Quisquater and David Samyde. 2001. ElectroMagnetic Analysis (EMA): Measures and Counter-Measures for Smart Cards. In *Smart Card Programming and Security, International Conference on Research in Smart Cards, E-smart 2001, Cannes, France, September 19-21, 2001, Proceedings (Lecture Notes in Computer Science)*, Isabelle Attali and Thomas P. Jensen (Eds.), Vol. 2140. Springer, 200–210. https://doi.org/10.1007/3-540-45418-7_17
- [67] Tobias Schneider and Amir Moradi. 2016. Leakage assessment methodology – Extended version. *Journal of Cryptographic Engineering* 6, 2 (2016), 85–99. <https://doi.org/10.1007/s13389-016-0120-y>

- [68] Ru-Hui Shi, Zeng-Ju Li, Lei Du, Qian Peng, and Jiu-Ba Xu. 2015. Side Channel Analysis on SM2 Decryption Algorithm. *Journal of Cryptologic Research* 2, 5 (2015), 467–476. <https://doi.org/10.13868/j.cnki.jcr.000093>
- [69] Sam Tregar. 2002. *Writing Perl Modules for CPAN*. Apress. <https://doi.org/10.1007/978-1-4302-1152-5>
- [70] Joop van de Pol, Nigel P. Smart, and Yuval Yarom. 2015. Just a Little Bit More. In *Topics in Cryptology - CT-RSA 2015, The Cryptographer's Track at the RSA Conference 2015, San Francisco, CA, USA, April 20-24, 2015. Proceedings (Lecture Notes in Computer Science)*, Kaisa Nyberg (Ed.), Vol. 9048. Springer, 3–21. https://doi.org/10.1007/978-3-319-16715-2_1
- [71] Tom van Goethem, Wouter Joosen, and Nick Nikiforakis. 2015. The Clock is Still Ticking: Timing Attacks in the Modern Web. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, Indrajit Ray, Ninghui Li, and Christopher Kruegel (Eds.). ACM, 1382–1393. <https://doi.org/10.1145/2810103.2813632>
- [72] Pepe Vila and Boris Köpf. 2017. Loophole: Timing Attacks on Shared Event Loops in Chrome. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, Engin Kirda and Thomas Ristenpart (Eds.). USENIX Association, 849–864. <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/vila>
- [73] Samuel Weiser, Raphael Spreitzer, and Lukas Bodner. 2018. Single Trace Attack Against RSA Key Generation in Intel SGX SSL. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security, AsiaCCS 2018, Incheon, Republic of Korea, June 04-08, 2018*, Jong Kim, Gail-Joon Ahn, Seungjoo Kim, Yongdae Kim, Javier López, and Taesoo Kim (Eds.). ACM, 575–586. <https://doi.org/10.1145/3196494.3196524>
- [74] Bernard L. Welch. 1947. The generalization of 'Student's' problem when several different population variances are involved. *Biometrika* 34 (1947), 28–35. <http://www.jstor.org/stable/2332510>
- [75] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, USENIX Association, 719–732. <https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom>
- [76] Kaiyu Zhang, Sen Xu, Dawu Gu, Haihua Gu, Junrong Liu, Zheng Guo, Ruitong Liu, Liang Liu, and Xiaobo Hu. 2017. Practical Partial-Nonce-Exposure Attack on ECC Algorithm. In *13th International Conference on Computational Intelligence and Security, CIS 2017, Hong Kong, China, December 15-18, 2017*, IEEE, 248–252. <https://doi.org/10.1109/CIS.2017.00061>
- [77] Yingqian Zhang, Ari Juels, Michael K. Reifer, and Thomas Ristenpart. 2014. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, Gail-Joon Ahn, Moti Yung, and Ninghui Li (Eds.). ACM, 990–1003. <https://doi.org/10.1145/2660267.2660356>

A REMOTE TIMINGS SCA EVALUATION: ECDSA

As stated in the Introduction, as a secondary goal, we aimed at reviewing the abstraction level at which SCA countermeasures are implemented. Specifically, pushing for a *secure-by-default* approach, we proposed to move each one of the SCA countermeasures discussed in this work at the lowest possible abstraction level.

As a result, the changes we proposed affected also other existing cryptosystems, increasing their resistance to SCA. In particular, in this section we evaluate the impact of our patchset on the ECDSA cryptosystem, specifically when using *generic* prime curves (as opposed to curves for which an alternative optimized implementation is specifically provided).

Figure 11 shows an empirical evaluation—similar to the one presented in Section 4 and Section 7—on the impact of the proposed mitigations on ECDSA over the `secp256k1` [24] GLV prime curve used in the Bitcoin protocol.

Both plots show the latency dependency on the nonce length: the top plot related to the OpenSSL implementation as of version 1.1.1-pre3, while the bottom plot shows the results after applying the proposed patchset. Specifically, the original implementation already applied the *nonce padding* and the *FLT modular inversion*

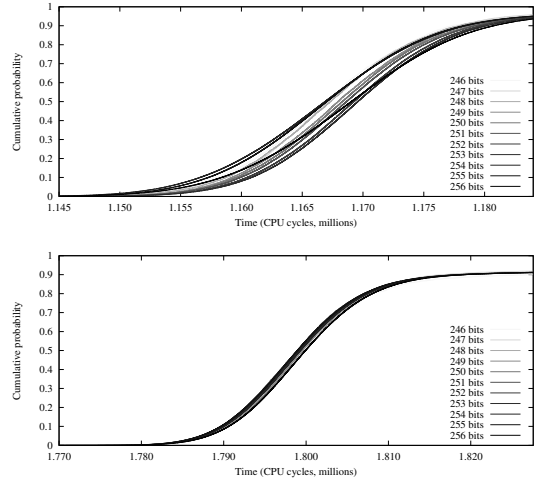


Figure 11: Latency dependency on the nonce length for ECDSA signature generation over the `secp256k1` prime curve on amd64 architecture. Top: OpenSSL 1.1.1-pre3, in which a wNAF algorithm is used to implement EC scalar multiplication (see Section 5.1). Bottom: After applying our patchset (see Section 7), most notably switching to a Montgomery ladder algorithm for scalar multiplication instead of wNAF.

countermeasures, so the main change between the two implementations is due to adopting a Montgomery ladder algorithm for EC scalar multiplication instead of the wNAF algorithm adopted in the original implementation (see Section 7).

Comparing the two plots, our mitigations introduce an improvement centered around the median values. This is due in part to the fact that timings in the top plot depend on the weight of scalars, while the timings in the bottom plot are independent of the weight. This leads to lower deviations for the majority of data points centered around the median.

PUBLICATION

VI

Port Contention for Fun and Profit

A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. Pereida García and N. Tuveri

2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019. 2019, 870–887

DOI: 10.1109/SP.2019.00066

Publication reprinted with the permission of the copyright holders

Port Contention for Fun and Profit

Alejandro Cabrera Aldaya*, Billy Bob Brumley†, Sohaib ul Hassan†, Cesar Pereida García†, Nicola Taveri†

*Universidad Tecnológica de la Habana (CUJAE), Habana, Cuba

†Tampere University, Tampere, Finland

Abstract—Simultaneous Multithreading (SMT) architectures are attractive targets for side-channel enabled attackers, with their inherently broader attack surface that exposes more per physical core microarchitecture components than cross-core attacks. In this work, we explore SMT execution engine sharing as a side-channel leakage source. We target ports to stacks of execution units to create a high-resolution timing side-channel due to port contention, inherently stealthy since it does not depend on the memory subsystem like other cache or TLB based attacks. Implementing our channel on Intel Skylake and Kaby Lake architectures featuring Hyper-Threading, we mount an end-to-end attack that recovers a P-384 private key from an OpenSSL-powered TLS server using a small number of repeated TLS handshake attempts. Furthermore, we show that traces targeting shared libraries, static builds, and SGX enclaves are essentially identical, hence our channel has wide target application.

I. INTRODUCTION

Microarchitecture side-channel attacks increasingly gain traction due to the real threat they pose to general-purpose computer infrastructure. New techniques emerge every year [1, 2], and they tend to involve lower level hardware, they get more complex but simpler to implement, and more difficult to mitigate, thus making microarchitecture attacks a more viable attack option. Many of the current microarchitecture side-channel techniques rely on the persistent state property of shared hardware resources, e.g., caches, TLBs, and BTBs, but non-persistent shared resources can also lead to side-channels [3], allowing leakage of confidential information from a trusted to a malicious process.

The microprocessor architecture is complex and the effect of a component in the rest of the system can be difficult (if not impossible) to track accurately: especially when components are shared by multiple processes during execution. Previous research [4, 5] confirms that as long as (persistent and non-persistent) shared hardware resources exist, attackers will be able to leak confidential information from a system.

In this work, we present a side-channel attack vector exploiting an inherent component of modern processors using Intel Hyper-Threading technology. Our new side-channel technique PORTSMASH is capable of exploiting timing information derived from port contention to the execution units, thus targeting a non-persistent shared hardware resource. Our technique can choose among several configurations to target different ports in order to adapt to different scenarios, thus offering a very fine spatial granularity. Additionally, PORTSMASH is highly portable and its prerequisites for execution are minimal, i.e., does not require knowledge of memory cache-lines, eviction

sets, machine learning techniques, nor reverse engineering techniques.

To demonstrate PORTSMASH in action, we present a complete end-to-end attack in a real-world setting attacking the NIST P-384 curve during signature generation in a TLS server compiled against OpenSSL 1.1.0h for crypto functionality. Our *Spy* program measures the port contention delay while executing in parallel to ECDSA P-384 signature generation, creating a timing signal trace containing a noisy sequence of *add* and *double* operations during scalar multiplication. We then process the signal using various techniques to clean the signal and reduce errors in the information extracted from each trace. We then pass this partial key information to a recovery phase, creating lattice problem instances which ultimately yield the TLS server’s ECDSA private key.

We extend our analysis to SGX, showing it is possible to retrieve secret keys from SGX enclaves by an unprivileged attacker. We compare our PORTSMASH technique to other side-channel techniques in terms of spatial resolution and detectability. Finally, we comment on the impact of current mitigations proposed for other side-channels on PORTSMASH, and our recommendations to protect against it.

In summary, we offer a full treatment of our new technique: from microarchitecture and side-channel background (Section II); to the nature of port contention leakage when placed in an existing covert channel framework (Section III); to its construction as a versatile timing side-channel (Section IV); to its application in real-world settings, recovering a private key (Section V); to discussing (lack of) detectability and mitigations (Section VI). We conclude in Section VII.

II. BACKGROUND

A. Microarchitecture

This section describes some of Intel’s microarchitectural components and how they behave with Intel SMT implementation (i.e., Hyper-Threading technology). Intel launched its SMT implementation with the Pentium 4 MMX processor [6]. Hyper-Threading technology (HT) aims at providing parallelism without duplicating all microarchitectural components in a physical processor. Instead, a processor supporting Hyper-Threading has at least two logical cores per physical core where some components are shared between the logical ones.

Figure 1 shows a high-level description of the layout of an Intel i7 processor [7]. This figure shows four physical cores, each with two logical cores. In this setting, the OS sees a processor with eight cores.

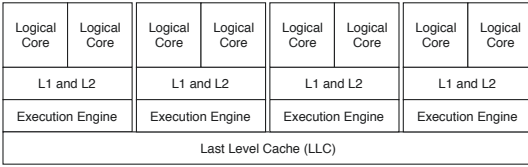


Fig. 1. Intel i7 Core processor.

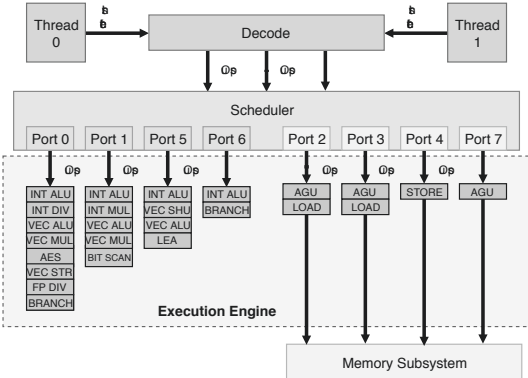


Fig. 2. Skylake/Kaby Lake microarchitecture.

Figure 1 sketches some microarchitectural components with a sharing perspective. L1 and L2 caches are shared between a pair of logical cores in the same physical core. The next level depicts how an Execution Engine (EE) is also shared between two logical cores. This component is very important for this paper as the presented microarchitectural side-channel relies on this logical-core-shared feature. On the other hand, the last level cache (LLC) is shared between all cores.

Generally speaking, the EE is responsible for executing instructions therefore it is closely related to the pipeline concept [7, 8]. A simplified pipeline model consists of three phases: (1) fetch, (2) decode, and (3) execute. While these phases have complex internal working details, Figure 2 provides a high-level abstraction focusing mainly on the EE part, and its description below also follows the same approach. For more information about its inner working details we defer to [6–8].

Each logical core has its own registers file, and the pipeline fetches instructions from memory according to the program counter on each of them. For the sake of processing performance fairness, this fetching is interleaved between the logical cores. After the fetch stage, a decoding phase decomposes each instruction into simpler micro-operations (*uops*). Each micro-operation does a single task, therefore this splitting helps out-of-order execution by interleaving their executions for the sake of performance. After this point, all processing is done on *uops* instead of instructions. The decoding phase then issues these *uops* to the execution scheduler.

At the scheduler there is a queue of *uops* that belongs to both logical cores. One task of the scheduler is issuing these *uops* to the Execution Ports while maximizing performance. An Execution Port is a *channel* to the execution units, the latter being where *uops* are actually executed. Figure 2 shows execution units as gray-colored boxes with labels indicating their functionality. For example, ports 0, 1, 5, and 6 can be used to execute simple arithmetic instructions, because each of them is a channel to an ALU execution unit. While ports 2, 3, 4, and 7 are dedicated to memory-based *uops* (e.g., loads and stores).

As an illustrative example of how the whole process happens in this simplified model, let us consider the `adc mem, reg` instruction (AT&T syntax), which adds (with carry) the value at memory location `mem` into the content in register `reg`. According to Fog’s instruction table for Skylake microarchitecture [9], this instruction splits into two *uops*: one arithmetic *uop* (that actually performs the addition) and another for loading a value from memory. The former can be issued to ports 0 or 6, while the latter to port 2 and 3 [9]. However, if we change the operand order in the original instruction (i.e., now the addition result is stored back in the memory location `mem`), the new instruction splits into three *uops*: two are essentially the same as before and another is issued for storing the result back to memory (i.e., an operation handled by port 4).

This execution sequence behaves exactly the same in the presence of Hyper-Threading. At the scheduler, there are *uops* waiting for dispatch to some port for execution. These *uops* could actually belong to instructions fetched from any logical core, therefore, these cores share the EE in a very granular approach (at *uops* level).

B. SMT: Timing Attacks

Timing attacks on microprocessors featuring SMT technology have a long and storied history with respect to side-channel analysis. Since the revival of SMT in 1995 [10], it was noted that contention was imminent, particularly in the memory subsystem. Arguably, timing attacks became a more serious security threat once Intel introduced its Hyper-Threading technology on the Pentium 4 microarchitecture. Researchers knew that resource sharing leads to resource contention, and it took a remarkably short time to notice that contention introduces timing variations during execution, which can be used as a covert channel, and as a side-channel.

In his pioneering work, Percival [11] described a novel cache-timing attack against RSA’s Sliding Window Exponentiation (SWE) implemented in OpenSSL 0.9.7c. The attack exploits the microprocessor’s Hyper-Threading feature and after observing that threads “share more than merely the execution units”, the author creates a spy process that exfiltrates information from the L1 data cache. The L1 data cache attack correctly identifies accesses to the precomputed multipliers used during the SWE algorithm, leading to RSA private key recovery. As a countermeasure, to ensure uniform access to the cache lines, irrespective of the multiplier used,

the OpenSSL team included a “constant-time” Fixed Window Exponentiation (FWE) algorithm paired with a scatter-gather method to mask table access [12].

Cache-based channels are not the only shared resource to receive security attention. Wang and Lee [3] and Aciğmez and Seifert [13] analyzed integer multiplication unit contention in old Intel Pentium 4 processors with SMT support [6]. In said microarchitecture, the integer multiplication unit is shared between the two logical cores. Therefore contention could exist between two concurrent processes running in the same physical core if they issue integer multiplication instructions. Wang and Lee [3] explore its application as a covert channel, while Aciğmez and Seifert [13] expand the side-channel attack approach.

Aciğmez and Seifert [13] stated this side-channel attack is very specific to the targeted Intel Pentium 4 architecture due to the fact that said architecture only has one integer multiplier unit. They illustrated an attack against the SWE algorithm in OpenSSL 0.9.8e. For this purpose they developed a proof-of-concept, modifying OpenSSL source code to enhance the distinguishability between square and multiplication operations in the captured trace. In addition to integer multiplication unit sharing, their attack relies on the fact that square and multiplication operations have different latencies, an unnecessary assumption in our work.

In a 2016 blog post¹, Anders Fogh introduced Covert Shotgun, an automated framework to find SMT covert channels. The strategy is to enumerate all possible pairs of instructions in an ISA. For each pair, duplicate each instruction a small number of times, then run each block in parallel on the same physical core but separate logical cores, measuring the clock-cycle performance. Any pairwise timing discrepancies in the resulting table indicate the potential for a covert channel, where the source of the leakage originates from any number of shared SMT microarchitecture components. Fogh explicitly mentions caching of decoded *uops*, the reorder buffer, port congestion, and execution unit congestion as potential sources, even reproducing the `rdseed` covert channel [14] that remarkably works across physical cores.

Covert channels from Covert Shotgun can be viewed as a higher abstraction of the integer multiplication unit contention covert channel by Wang and Lee [3], and our side-channel a higher abstraction of the corresponding side-channel by Aciğmez and Seifert [13]. Now limiting the discussion to port contention, our attack focuses on the port sharing feature. This allows a *darker-box* analysis of the targeted binary because there is no need to know the exact instructions executed by the victim process, only that the attacker must determine the distinguishable port through trial and error. This feature is very helpful, for example, in a scenario where the targeted code is encrypted and only decrypted/executed inside an SGX enclave [15].

Analogous to [11], Aciğmez et al. [16] performed a cache-timing attack against OpenSSL DSA, but this time targeting

the L1 instruction cache. The authors demonstrate an L1 instruction cache attack in a real-world setting and using analysis techniques such as vector quantization and hidden Markov models, combined with a lattice attack, they achieve DSA full key recovery on OpenSSL version 0.9.8l. They perform their attack on an Intel Atom processor featuring Hyper-Threading. Moreover, due to the relevance and threat of cache-timing attacks, the authors list and evaluate several possible countermeasures to close the cache side-channels.

More recently, Yarom et al. [5] presented CacheBleed, a new cache-timing attack affecting some older processors featuring Hyper-Threading such as Sandy Bridge. The authors exploit the fact that cache banks can only serve one request at a time, thus issuing several requests to the same cache bank, i.e., accessing the same offset within a cache line, results in bank contention, leading to timing variations and leaking information about low address bits. To demonstrate the attack, the authors target the RSA exponentiation in OpenSSL 1.0.2f. During exponentiation, RSA uses the scatter-gather method adopted due to Percival’s work [11]. More precisely, to compute the exponentiation, the scatter-gather method accesses the cache bank or offset within a cache line according to the multiplier used, which depends on a digit of the private key. Thus, by detecting the used bank through cache bank contention timings, an attacker can determine the multiplier used and consequently digits of the private key. The attack requires very fine granularity, thus the victim and the spy execute in different threads in the same core, and after observing 16,000 decryptions, the authors fully recover 4096-bit RSA private keys.

In 2018, Gras et al. [4] presented TLBleed, a new class of side-channel attacks relying on the Translation Lookaside Buffers (TLB) and requiring Hyper-Threading to leak information. In their work, the authors reverse engineer the TLB architecture and demonstrate the TLB is a (partially) shared resource in SMT Intel architectures. More specifically, the L1 data TLB and L2 mixed TLB are shared between multiple logical cores and a malicious process can exploit this to leak information from another process running in the same physical core. As a proof-of-concept, the authors attack a non constant-time version of 256-bit EdDSA [17] and a 1024-bit RSA hardened against FLUSH+RELOAD as implemented in libgcrypt. The EdDSA attack combined with a machine-learning technique achieves a full key recovery success rate of 97%, while the RSA attack recovers 92% of the private key but the authors do not perform full key recovery. Both attacks are possible after capturing a single trace.

III. INSTANTIATING COVERT SHOTGUN

Being an automated framework, Covert Shotgun is a powerful tool to detect potential leakage in SMT architectures. But due to its black-box, brute-force approach, it leaves identifying the root cause of leakage as an open problem: “*Another interesting project would be identifying [subsystems] which are being congested by specific instructions*”. In this section, we fill this research gap with respect to port contention.

¹<https://cyber.wtf/2016/09/27/covert-shotgun/>

Our intention is not to utilize this particular covert channel in isolation, but rather understand how the channel can be better optimized for its later conversion to a side-channel in Section IV.

A. Concept

Assume cores C_0 and C_1 are two logical cores of the same physical core. To make efficient and fair use of the shared EE, a simple strategy for port allocation is as follows. Denote i the clock cycle number, $j = i \bmod 2$, and \mathcal{P} the set of ports.

- 1) C_j is allotted $\mathcal{P}_j \subseteq \mathcal{P}$ such that $|\mathcal{P} \setminus \mathcal{P}_j|$ is minimal.
- 2) C_{1-j} is allotted $\mathcal{P}_{1-j} = \mathcal{P} \setminus \mathcal{P}_j$.

There are two extremes in this strategy. For instance, if C_0 and C_1 are executing fully pipelined code with no hazards, yet make use of disjoint ports, then both C_0 and C_1 can issue in every clock cycle since there is no port contention. On the other hand, if C_0 and C_1 are utilizing the same ports, then C_0 and C_1 alternate, issuing every other clock cycle, realizing only half the throughput performance-wise.

Consider Alice and Bob, two user space programs, executing concurrently on C_0 and C_1 , respectively. The above strategy implies the performance of Alice depends on port contention with Bob, and vice versa. This leads to a covert timing channel as follows. Take two latency-1 instructions: `NOP0` that can only execute on port 0, and `NOP1` similarly on port 1. Alice sends a single bit of information to Bob as follows.

- 1) If Alice wishes to send a zero, she starts executing `NOP0` continuously; otherwise, a one and `NOP1` instead.
- 2) Concurrently, Bob executes a fixed number of `NOP0` instructions, and measures the execution time t_0 .
- 3) Bob then executes the same fixed number of `NOP1` instructions, and measures the execution time t_1 .
- 4) If $t_1 > t_0$, Bob receives a one bit; otherwise, $t_0 > t_1$ and a zero bit.

The covert channel works because if both Alice and Bob are issuing `NOP0` instructions, they are competing for port 0 and the throughput will be cut in half (similarly for `NOP1` and port 1). On the other hand, with no port contention both `NOP0` and `NOP1` execute in the same clock cycle, achieving full throughput and lower latency.

B. Implementation

In this section, we give empirical evidence that Intel Hyper-Threading uses the previous hypothetical port allocation strategy for SMT architectures (or one indistinguishable from it for our purposes). Along the way, we optimize the channel with respect to pipeline usage, taking into account instruction latencies and duplicated execution units.

In these experiments, we used an Intel Core i7-7700HQ Kaby Lake featuring Hyper-Threading with four cores and eight threads. Using the `perf` tool to monitor `uops` dispatched to each of the seven ports and the clock cycle count for a fixed number of instructions, we determined the port footprint and performance characteristics of several instructions, listed in Table I. We chose this mix of instructions to demonstrate

TABLE I
SELECTIVE INSTRUCTIONS. ALL OPERANDS ARE REGISTERS, WITH NO MEMORY OPS. THROUGHPUT IS RECIPROCAL.

Instruction	Ports	Latency	Throughput
<code>add</code>	0 1 5 6	1	0.25
<code>crc32</code>	1	3	1
<code>popcnt</code>	1	3	1
<code>vpermd</code>	5	3	1
<code>vpbroadcastd</code>	5	3	1

TABLE II
RESULTS OVER A THOUSAND TRIALS. AVERAGE CYCLES ARE IN THOUSANDS, RELATIVE STANDARD DEVIATION IN PERCENTAGE.

		Diff. Phys. Core		Same Phys. Core	
Alice	Bob	Cycles	Rel. SD	Cycles	Rel. SD
Port 1	Port 1	203331	0.32%	408322	0.05%
Port 1	Port 5	203322	0.25%	203820	0.07%
Port 5	Port 1	203334	0.31%	203487	0.07%
Port 5	Port 5	203328	0.26%	404941	0.05%

the extremes: from `add` that can be issued to any of the four integer ALUs behind ports 0, 1, 5, or 6, to `crc32` and `vpermd` that restrict to only ports 1 and 5, respectively. Furthermore, to minimize the effect of the memory subsystem on timings (e.g., cache hits and misses), in this work we do not consider any explicit store or load instructions, or any memory operands to instructions (i.e., all operands are registers).

Given the results in Table I, we construct the covert channel as follows: `crc32` (port 1) will serve as the `NOP0` instruction, and `vpermd` (port 5) as `NOP1`. Note that this configuration is one of the n^2 brute-force pairs of Covert Shotgun. However, as we are targeting port contention we take into account instruction latency, throughput, and port usage to maximize its impact. Being `crc32` and `vpermd` latency-3 instructions, we construct a block of three such instructions with disjoint operands to fill the pipeline, avoid hazards, and realize a throughput of one instruction per clock cycle. We repeated each block 64 times to obtain a low ratio of control flow logic to overall instructions retired. The Alice program sends a zero bit by executing the repeated `crc32` blocks in an infinite loop. Concurrently on the receiver side, using `perf`, we measured the number of clock cycles required for the Bob program to execute 2^{20} of the repeated `crc32` blocks, then again measured with the same number of repeated `vpermd` blocks. We then repeated the experiment with Alice sending a one bit analogously with the `vpermd` instruction. We carried out the experiments with both Alice and Bob pinned to separate logical cores of the same physical core, then also different physical cores. As a rough estimate, for full throughput we expect $3 \cdot 64 \cdot 2^{20} \approx 201$ million cycles (three instructions, with 64 repetitions, looping 2^{20} times); even with a latency of three, our construction ensures a throughput of one. Of course there is some overhead for control flow logic.

Table II contains the results, averaged over a thousand trials. First on separate physical cores, we see that the cycle count is

essentially the same and approaches our full throughput estimate, regardless of which port Alice and/or Bob are targeting. This confirms the channel does not exist across physical cores. In contrast, the results on the same physical core validates the channel. When Alice and Bob target separate ports, i.e., the port 1/5 and 5/1 cases, the throughput is maximum and matches the results on different physical cores. However, when targeting the same port, i.e., the port 1/1 and 5/5 cases, the throughput halves and the cycle count doubles due to the port contention. This behavior precisely matches the hypothesis in Section III-A.

IV. FROM COVERT TO SIDE-CHANNEL

One takeaway from the previous section is that, given two user space programs running on two separate logical cores of the same physical core, the clock cycle performance of each program depends on each other’s port utilization. Covert Shotgun leaves extension to side-channels as an open problem: “it would be interesting to investigate to what extent these covert channels extend to spying”. In this section, we fill this research gap by developing PORTSMASH, a new timing side-channel vector via port contention.

At a high level, in PORTSMASH the goal of the Spy is to saturate one or more ports with a combination of full instruction pipelining and/or generous instruction level parallelism. By measuring the time required to execute a series of said instructions, the Spy learns about the Victim’s rate and utilization of these targeted ports. A higher latency observed by the Spy implies port contention with the Victim, i.e., the Victim issued instructions executed through said ports. A lower latency implies the Victim did not issue such instructions, and/or stalled due to a hazard or waiting due to, e.g., a cache miss. If the Victim’s ability to keep the pipeline full and utilize instruction level parallelism depends on a secret, the Spy’s timing information potentially leaks that secret.

As a simple example conceptually related to our later application in Section V, consider binary scalar multiplication for elliptic curves. Each projective elliptic curve point double and conditional add is made up of a number of finite field additions, subtractions, shifts, multiplications, and squarings. These finite field operations utilize the pipeline and ports in very different ways and have asymptotically different running times. For example, shifts are extremely parallelizable, while additions via add-with-carry are strictly serial. Furthermore, the number and order of these finite field operations is not the same for point double and add. The Spy can potentially learn this secret sequence of doubles and conditional adds by measuring its own performance through selective ports, leading to (secret) scalar disclosure.

Figure 3 lists our proposed PORTSMASH Spy process. The first `rdtsc` wrapped by `lfence` establishes the start time. Then, depending on the architecture and target port(s), the Spy executes one of several strategies to saturate the port(s). Once those complete, the second `rdtsc` establishes the end time. These two counters are concatenated and stored out to a buffer at `rdi`. The Spy then repeats this entire process. Here we

```

mov $COUNT, %rcx                                     #elif defined(P0156)
l:                                                     .rept 64
lfence                                                add %r8, %r8
rdtsc                                                add %r9, %r9
lfence                                                add %r10, %r10
mov %rax, %rsi                                       add %r11, %r11
                                                       .endr
                                                       #else
                                                       #error No ports defined
                                                       #endif
                                                       lfence
                                                       rdtsc
                                                       shl $32, %rax
                                                       or %rsi, %rax
                                                       mov %rax, (%rdi)
                                                       add $8, %rdi
                                                       dec %rcx
                                                       jnz lb
                                                       .endr
#endif P1
.rept 48
crc32 %r8, %r8
crc32 %r9, %r9
crc32 %r10, %r10
.endr
#elif defined(P5)
.rept 48
vpermd %ymm0, %ymm1, %ymm0
vpermd %ymm2, %ymm3, %ymm2
vpermd %ymm4, %ymm5, %ymm4
.endr

```

Fig. 3. The PORTSMASH technique with multiple build-time port configurations P1, P5, and P0156.

choose to store the counter values and not only the latency, as the former helps identify interrupts (e.g., context switches) and the latter can always be derived offline from the former, but the converse is not true. It is also worth mentioning the Spy must ensure some reasonable number of instructions retired between successive `rdtsc` calls to be able to reliably detect port contention; we expand later.

In general, strategies are architecture dependent and on each architecture there are several strategies, depending on what port(s) the Spy wishes to measure. We now provide and describe three such example strategies (among several others that naturally follow) for Intel Skylake and Kaby Lake: one that leverages instruction level parallelism and targets multiple ports with a latency-1 instruction, and two that leverage pipelining and target a single port with higher latency instructions.

Multiple ports: In Figure 3, the `P0156` block targets ports 0, 1, 5, and 6. These four `add` instructions do not create hazards, hence all four can execute in parallel to the four integer ALUs behind these ports, and as a latency-1 instruction in total they should consume a single clock cycle. To provide a window to detect port contention, the Spy replicates these instructions 64 times. With no port contention, this should execute in 64 clock cycles, and 128 clock cycles with full port contention.

Single port: In Figure 3, the `P1` and `P5` blocks target port 1 and 5, respectively, in a similar fashion. Since these are latency-3 instructions, we pipeline three sequential instructions with distinct arguments to avoid hazards and fill the pipeline, achieving full throughput of one instruction per cycle. Here the window size is 48, so the block executes with a minimum $3 \cdot 48 + 2 = 146$ clock cycles with no port contention, and with full port contention the maximum is roughly twice that.

A. Comparison

Our PORTSMASH technique relies on secret-dependent execution port footprint, a closely related concept to secret-

dependent *instruction execution cache footprint*. Although similar in spirit to L1 icache attacks or LLC cache attacks, since both rely on a secret-dependent footprint in a microarchitecture component, we demonstrate that PORTSMASH offers finer granularity and is stealthier compared to other techniques. To differentiate PORTSMASH from previous techniques, we compare them with respect to spatial resolution, detectability, cross-core, and cross-VM applicability. We admit that detectability is extremely subjective, especially across different microarchitecture components; our rating is with respect to a malicious program while the target victim is idle, i.e., waiting to capture.

Initially, Osvik et al. [18] proposed the PRIME+PROBE technique against the L1 dcache, relying on SMT technology to provide asynchronous execution. Newer enhancements to this technique allowed cross-core (and cross-VM) successful attacks [22–24]. The spatial resolution of this attack is limited to cache-set granularity, that is usually a minimum of 512 bytes. Typically, the PRIME+PROBE technique occupies all cache sets, moderately detectable if cache activity monitoring takes place.

Later, Yarom and Falkner [19] proposed the FLUSH+RELOAD technique, a high resolution side-channel providing cache-line granularity with an improved eviction strategy. Closely related, Gruss et al. [20] proposed FLUSH+FLUSH, a stealthier version of FLUSH+RELOAD. Both techniques rely on shared memory between Victim and Spy processes, in addition to the `clflush` instruction to evict cache lines from the LLC. While this is a typical setting in cross-core scenarios due to the use of shared libraries, the impact in cross-VM environments is limited due to the common practice of disabling page de-duplication [25, Sec. 3.2].

More recently, Gras et al. [4] proposed TLBLEED as another microarchitecture attack technique. Even if this is not a “pure” cache technique, it exploits TLBs, a form of cache for memory address translations [7]. Interestingly, this subtle distinction is sufficient for making it stealthier to cache countermeasures [4]. On the downside, the spatial resolution of this attack is limited to a memory page (4 KB). Since no cross-core improvements have been proposed for either TLBLEED or PORTSMASH, it could be seen as a drawback of these attacks. However, attackers can launch multiple Spy processes to occupy all cores and ensure co-location on the same physical core; see [26, Sec. 3.1] for a related discussion.

Recent microarchitecture attacks have been proposed achieving intra cache-line granularity. Yarom et al. [5] demonstrated that intra-cache granularity is possible—at least in older Intel microprocessors—with their CacheBleed attack. This attack proposes two techniques to achieve this granularity: cache bank conflicts and *write-after-read* false dependencies. Cache bank conflicts have a limited impact, considering the authors discovered that current Intel microprocessors no longer have cache banks; thus this technique does not apply to newer

30f0	<x64_foo>	4150	<x64_bar>
30f0	test %rdi,%rdi	4150	test %rdi,%rdi
30f3	je 4100 <x64_foo+0x1010>	4153	je 5100 <x64_bar+0xffb0>
30f9	jmpq 4120 <x64_foo+0x1030>	4159	jmpq 5140 <x64_bar+0xffb0>
....		
4100	popcnt %r8,%r8	5100	popcnt %r8,%r8
4105	popcnt %r9,%r9	5105	popcnt %r9,%r9
410a	popcnt %r10,%r10	510a	popcnt %r10,%r10
410f	popcnt %r8,%r8	510f	popcnt %r8,%r8
4114	popcnt %r9,%r9	5114	popcnt %r9,%r9
4119	popcnt %r10,%r10	5119	popcnt %r10,%r10
411e	jmp 4100 <x64_foo+0x1010>	511e	popcnt %r8,%r8
4120	vpbroadcastd %xmm0,%ymm0	5123	popcnt %r9,%r9
4125	vpbroadcastd %xmm1,%ymm1	5128	popcnt %r10,%r10
412a	vpbroadcastd %xmm2,%ymm2	512d	popcnt %r8,%r8
412f	vpbroadcastd %xmm0,%ymm0	5132	popcnt %r9,%r9
4134	vpbroadcastd %xmm1,%ymm1	5137	popcnt %r10,%r10
4139	vpbroadcastd %xmm2,%ymm2	513c	jmp 5100 <x64_bar+0xffb0>
413e	jmp 4120 <x64_foo+0x1030>	513e	xchg %ax,%ax
4140	retq	5140	vpbroadcastd %xmm0,%ymm0
		5145	vpbroadcastd %xmm1,%ymm1
		514a	vpbroadcastd %xmm2,%ymm2
		514f	vpbroadcastd %xmm0,%ymm0
		5154	vpbroadcastd %xmm1,%ymm1
		5159	vpbroadcastd %xmm2,%ymm2
		515e	vpbroadcastd %xmm0,%ymm0
		5163	vpbroadcastd %xmm1,%ymm1
		5168	vpbroadcastd %xmm2,%ymm2
		516d	vpbroadcastd %xmm0,%ymm0
		5172	vpbroadcastd %xmm1,%ymm1
		5177	vpbroadcastd %xmm2,%ymm2
		517c	jmp 5140 <x64_bar+0xffb0>
		517e	retq

Fig. 4. Two Victims with similar port footprint, i.e., port 1 and port 5, but different cache footprint. Left: Instructions span a single cache-line. Right: Instructions span multiple cache-lines.

microprocessors. To that end, Moghimi et al. [21] improved the previous work and proposed a *read-after-write* false dependency side-channel. The authors highlight the potential 5 cycle penalty introduced when a memory write is closely followed by a read, a more critical condition compared to a read closely followed by a memory write. This technique gives a 4-byte granularity on the cache-lines, thus allowing them to exploit the 5 cycle delay to perform a key recovery attack against a constant-time AES implementation on Intel IPP library.

To understand our detectability criteria in Table III, consider the following example. During a typical round of attack, a FLUSH+RELOAD process constantly reloads a previously flushed memory address, observing a large number of cache-misses, thus highly detectable. In contrast, a FLUSH+FLUSH process does not perform explicit loads, instead it relies on the time of `clflush` execution to determine the existence of data in the cache, thus lowly detectable. Sitting in the middle, a PRIME+PROBE process reloads data from cache at a slower rate compared to FLUSH+RELOAD, but still observing a significant amount of cache-misses, hence fairly detectable. On the other hand, TLBLEED, MemJam and CacheBleed attacks do not follow the same combination of cache eviction and memory load operations, instead they rely on timing variations observed when executing typical instructions during a computation, i.e., no `clflush`, thus their detectability is low.

Table III compares the previously mentioned techniques in their original version. As can be appreciated, our PORTSMASH technique enjoys the highest spatial resolution among them, since it goes beyond the cache-line and instead, it considers individual *uops* dispatched to the execution units. As an example, consider the two functions `x64_foo` and `x64_bar` in Figure 4. These two functions get passed an argument of either zero or one (e.g., a secret bit): in the former

²Cache-set size depends on the microprocessor specifications and can be calculated as (*cache line size* × *cache associativity*).

TABLE III
COMPARISON OF MICROARCHITECTURE ATTACK TECHNIQUES (ORIGINAL VERSIONS)

Attack	Spatial Resolution	Size	Detectability	Cross-Core	Cross-VM
TLBLEED [4]	Memory Page (Very low)	4 KB	Low	No	Yes/SMT
PRIME+PROBE [18]	Cache-set (Low)	512 bytes ²	Medium	Yes	Yes/SharedMem
FLUSH+RELOAD [19]	Cache-line (Med)	64 bytes	High	Yes	Yes/SharedMem
FLUSH+FLUSH [20]	Cache-line (Med)	64 bytes	Low	Yes	Yes/SharedMem
CacheBleed [5]	Intra cache-line (High)	8 bytes	Medium	No	Yes/SMT
MemJam [21]	Intra cache-line (High)	4 bytes	Medium	No	Yes/SMT
PORTSMASH	Execution port (Very High)	<i>uops</i>	Low	No	Yes/SMT

case, they start executing pipelined `popcnt` instructions in a loop, and `vpbroadcastd` instructions in the latter. The `x64_foo` function has all its functionality for both branches within a single cache line (64B), starting at address `0x4100`. In contrast, the `x64_bar` function has distinct cache lines for each branch: the zero case starts at address `0x5100` and the one case at `0x5140`, and the control flow for each corresponding loop restricts to its single cache line.

The `x64_bar` function is a potential target for L1 icache attacks, FLUSH+RELOAD attacks, FLUSH+FLUSH attacks, etc. since there are two different targets that span two different cache lines. In contrast, the `x64_foo` control flow resides in a single cache line: L1 icache attacks, FLUSH+RELOAD attacks, FLUSH+FLUSH attacks, etc. only have cache line granularity, and are not designed to distinguish varying code traversal within a single line. Remarkably, both `x64_foo` and `x64_bar` are potential targets for our new method. In this light, at a very high level what CacheBleed accomplished for dcache attacks—the ability to target at less than data cache line granularity—our method accomplishes for the code side, and furthermore potentially with a single trace instead of averaging traces.

To validate our findings, we ran the following set of PORT SMASH experiments. First, we configured the Victim process to execute the `x64_foo` function passing 0 as an argument, causing the Victim to issue `popcnt` commands, using port 1. In parallel, we configured the Spy process with the P1 strategy in the sibling logical core to issue and time `crc32` commands, thus creating contention and the Spy successfully tracks the Victim state by observing high latency. Then, we repeated the experiment but this time we passed 1 as an argument to the Victim process, executing `vpbroadcastd` instructions, using port 5. Since the Spy process is still using the P1 strategy, i.e., timing `crc32` instructions, port contention does not occur, hence the Spy successfully tracks the Victim state by observing low latency. Figure 5 (Top) shows the resulting trace for both cases, i.e., contention vs no-contention from a Spy process perspective configured with the P1 strategy. We then reconfigured the Spy to use the P5 strategy, and repeated the experiments, shown in Figure 5 (Bottom). This raw empirical data—that is clearly linearly separable—confirms not only the validity of our new side-channel in general, but furthermore the symmetry in the plots confirms that our technique even allows to leak code traversal information with granularity finer than

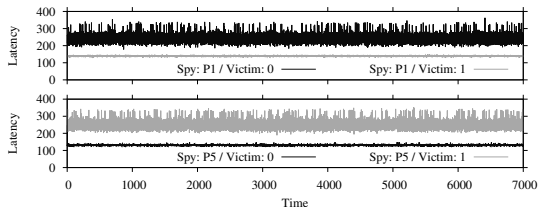


Fig. 5. Top: Timings for the PORTSMASH Spy when configured with P1, in parallel to the Victim executing `x64_foo` with `rdi` as both zero and one in two consecutive runs. Bottom: Analogous but with the Spy configured with P5.

cache-line, since in this case it is dependent on port utilization by the executed instructions *within* the cache-line.

V. APPLICATIONS

In the previous section, we developed a generic PORT SMASH Spy process to procure timing signals that detect port contention. In this section, we present the first attack using our technique in a real-world setting. We start with some background on ECC, and explain why P-384 is a highly relevant standardized elliptic curve, and examine its scalar multiplication code path within OpenSSL 1.1.0h and earlier, based on an implementation featuring secret-dependent execution flow, thus satisfying the PORTSMASH technique requirement. We then design and implement an end-to-end P-384 private key recovery attack that consists of three phases:

- 1) In the *procurement phase*, we target an stunnel TLS server authenticating with a P-384 certificate, using our tooling that queries the TLS server over multiple handshakes with the Spy measuring port contention in parallel as the server produces ECDSA signatures.
- 2) In the *signal processing phase*, we filter these traces and output partial ECDSA nonce information for each digital signature.
- 3) In the *key recovery phase*, we utilize this partial nonce information in a lattice attack to fully recover the server's P-384 private key.

We close this section with a discussion on applications to statically linked binaries and SGX enclaves. The rationale behind our choice to demonstrate an end-to-end attack for the non-SGX case is based on our perceived real-world implications. The number of web-servers powered by OpenSSL

outside SGX enclaves largely outweighs the number within SGX enclaves, by at least several orders of magnitude.

A. ECC and P-384

Koblitz [27] and Miller [28] introduced elliptic curves to cryptography during the mid 1980’s. By 1995, the National Security Agency (NSA) became a strong supporter of Elliptic Curve Cryptography (ECC) [29] and pushed for the adoption of ECDSA, the ECC variant of the (then) recently approved Digital Signature Algorithm (DSA) [30].

In 2005, NSA’s support of ECC was clear, mandating its use “for protecting both classified and unclassified National Security information[...], the NSA plans to use the elliptic curves over finite fields with large prime moduli (256, 384, and 521 bits) published by NIST” [31]. Shortly after, the NSA announced Suite B, a document recommending cryptography algorithms approved for protecting classified information up to Secret and Top Secret level, including P-256 at 128 bits of security, and P-384 at 192 bits.

During 2012, the Committee for National Security Systems (CNSS) issued CNSSP-15 [32], a document defining the set of public key cryptographic standards recommended to protect classified information until public standards for post-quantum cryptography (PQC) materialize, further pushing the support for both curves, P-256 and P-384. Suddenly in August 2015, and after a long history of ECC support, the NSA released a statement [33] urging the development of PQC and discouraging the late adoption of ECC, and instead focusing on the upcoming upgrade to quantum-resistant algorithms. Parallel to this statement, the Suite B recommendation was updated, mysteriously removing P-256 from the list of approved curves without giving any reason, and leaving P-384 as the only ECC option to protect information up to Top Secret level. In January 2016, the NSA issued a FAQ [34] derived from the statement released five months prior. They informed about the replacement of Suite B with an updated version of CNSS-15, and also finally commented on the removal of P-256 from the previous Suite B. We cherry-pick three statements from the document: (1) “equipment for NSS that is being built and deployed now using ECC should be held to a higher standard than is offered by P-256”; (2) “Elimination of the lower level of Suite B also resolves an interoperability problem raised by having two levels”; and (3) “CNSSP-15 does not permit use of P-521”.

To summarize, P-384 is the only compliant ECC option for Secret and Top Secret levels. Unfortunately, its implementations have not received the same scrutiny as P-256 and P-521; we expand later in this section.

ECDSA: For the purpose of this paper, we restrict to short Weierstrass curves over prime fields. With prime $p > 3$, all of the $x, y \in GF(p)$ solutions to the equation

$$E : y^2 = x^3 + ax + b$$

along with the point-at-infinity (identity) form a group. The domain parameters of interest are the NIST standard curves that set p a Mersenne-like prime and $a = -3 \in GF(p)$.

The user’s private-public keypair is (d_A, Q_A) where d_A is chosen uniformly from $[1..n)$ and $Q_A = [d_A]G$ holds. Generator $G \in E$ is of prime order n . A digital signature on message m compute as follows.

- 1) Select a secret nonce k uniformly from $[1..n)$.
- 2) Compute $r = (k[G])_x \bmod n$.
- 3) Compute $s = k^{-1}(h(m) + d_A r) \bmod n$.
- 4) Return the digital signature tuple (m, r, s) .

The hash function h can be any “approved” function, e.g., SHA-1, SHA-256, and SHA-512. Verification is not relevant to this work, hence we omit the description.

ECDSA and P-384 in OpenSSL: In OpenSSL, each elliptic curve has an associated method structure containing function pointers to common ECC operations. For ECDSA, scalar multiplication is the most performance and security-critical ECC operation defined in this method structure, and the actual algorithm to perform scalar multiplication depends on several factors, e.g., curve instantiated, scalar representation, OpenSSL version, and both library and application build-time options. We further elaborate on how these factors influence the final scalar multiplication execution path in Appendix B, while for the rest of this work we will focus on the code paths executed in OpenSSL 1.1.0h and below and specifically, as described in this paragraph, on the default implementation for elliptic curves over prime fields. Due to the long history of timing attacks against ECDSA and the possibility of improving the performance of some curves, over the years OpenSSL mainlined several implementations for scalar multiplication, especially for popular NIST curves over prime fields.

Based on work by Kasper [35]—and as a response to the data cache-timing attack by Brumley and Hakala [36]—OpenSSL introduced `EC_GFp_nistp256_method`, a constant-time scalar multiplication method for the NIST P-256 curve (and analogous methods for P-224 and P-521). This method uses secure table lookups (through masking) and fixed-window combing during scalar multiplication. This is a portable C implementation, but requires support for 128-bit integer types. Later, Gueron and Krasnov [37] introduced a faster constant-time method with their `EC_GFp_nistz256_method`. This method uses Intel AVX2 SIMD assembly to increase the performance of finite field operations, thus providing a considerable speedup when compared to `EC_GFp_nistp256_method` that is portable C. The NIST curve P-256 quickly became (arguably) the most widely used, fast, and timing-attack resistant of all NIST curves in OpenSSL.

Unfortunately, P-384 was neglected, and it missed all of the previous curve-specific improvements that provided timing attack security for P-224, P-256, and P-521. Instead, P-384—like any other short Weierstrass curve over a prime field, including e.g. `secp256k1` (adopted for Bitcoin operations) and Brainpool curves (RFC 5639[38])—follows the default OpenSSL implementation for scalar multiplication on prime curves. It is a non constant-time interleaving algorithm that uses Non-Adjacent Form (*wNAF*) for scalar representation

[39, Sec. 3.2]. Although this implementation has been repeatedly targeted for side-channel vulnerabilities [36, 40–42], it has never been exploited in the context of P-384 in OpenSSL.

During ECDSA signature generation, OpenSSL calls `ecdsa_sign_setup@crypto/ec/ecdsa_ossl.c` to perform steps 1 and 2 of the ECDSA algorithm described above. For the latter, the underlying `ec_wNAF_mul` function gets called to perform the scalar multiplication, where $r = [k]G$ is the relevant computation for this work. That function first transforms the scalar k to its w NAF representation and then, based on this representation, the actual scalar multiplication algorithm executes a series of *double* and *add* operations. To perform double and add operations, OpenSSL calls `ec_GFp_simple_dbl` and `ec_GFp_simple_add` respectively. There, these methods have several function calls to simpler and lower level Montgomery arithmetic, e.g., shift, add, subtract, multiply, and square operations. A single ECC double (or add) operation performs several calls to these arithmetic functions. Among the strategies mentioned in Section IV, we found that for our target the P5 strategy results in the cleanest trace overall.

In summary, by using the PORTSMASH technique during OpenSSL P-384 ECDSA signature generation, we can measure the timing variations due to port contention. More specifically, we capture the port contention delay during double and add operations, resulting in an accurate raw signal trace containing the sequence of operations during scalar multiplication, and leaking enough LSDs of multiple nonces k to later succeed in our *key recovery phase*.

B. Procurement Phase: TLS

Stunnel³ provides TLS/SSL tunneling services to servers (and clients) that do not speak TLS natively; during the *procurement phase* we used stunnel 5.49 as the TLS server. We compiled it from source and linked it against OpenSSL 1.1.0h for crypto functionality. Our setup consists of an Intel Core i7-6700 Skylake 3.40GHz featuring Hyper-Threading, with four cores and eight threads, running Ubuntu 18.04 LTS “Bionic Beaver”. In addition, we disabled TurboBoost to minimize any interference due to CPU frequency scaling. Nonetheless, we hypothesize enabling it would merely introduce some clock skew without substantially affecting the side-channel leakage itself. Scalar multiplication is a demanding task, so TurboBoost should already activate during execution and quickly reach the maximum stable frequency. This would have little impact on our results since we are more interested in the trailing portion of the trace. This decision is consistent with existing results in the literature, e.g. [16, 36, 41, 43].

We configured the stunnel server with a P-384 ECDSA certificate and `ECDHE-ECDSA-AES128-SHA256` as the TLS 1.2 cipher suite. We wrote a custom TLS client to connect to our TLS server. Typically, during a TLS handshake, the client and the server exchange several protocol messages, including `ClientHello`, `ServerHello`, and

³<https://www.stunnel.org>

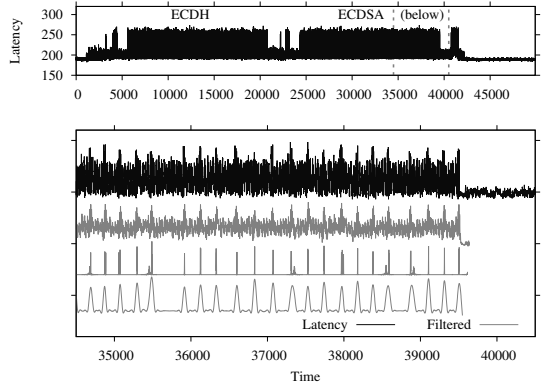


Fig. 6. Multiple TLS trace stages. Top: Raw TLS handshake trace showing scalar multiplications during ECDH and ECDSA. Bottom: Zoom at the end of the previous ECDSA trace, peaks (filtered) represent add operations. For example, this trace ends with an add operation, indicating the nonce is odd.

`ServerKeyExchange` parameters. These messages are concatenated, hashed, and digitally signed by the server. Then, the client verifies the signature before finally establishing a session with the server.

Our custom TLS client, acting as an attacker, serves two purposes: (1) it controls the start of the attack by initiating a TLS handshake with the stunnel service, alerting the Spy process to start capturing OpenSSL scalar multiplication operations performed by the server during the handshake; and (2) it collects protocol messages and digital signatures during the TLS handshake. Figure 6 (Top) shows a trace captured by the Spy process, containing the two scalar multiplication operations during TLS handshake, i.e. ECDH and ECDSA respectively.

The client drops the handshake as soon as the server presents the digital signature; since we are only interested in capturing up to the digital signature generation, this allows us to capture a trace in roughly 4 ms (~ 12.5 million clock cycles). Additionally, our client concatenates the protocol messages, hashes the resulting concatenation, and stores the message digest. Similarly, it stores the respective DER-encoded P-384 ECDSA signatures for each TLS handshake. This process is repeated as needed to build a set of traces, digest messages, and digital signatures that our lattice attack uses later in the *key recovery phase*.

Once the data tuples are captured, we proceed to the *signal processing phase*, where the traces are trimmed and filtered to reduce the noise and output useful information. Figure 6 (Bottom) shows a zoom at the end of the (Top) trace, where the filters reveal peaks representing add operations, separated by several double operations.

At a high level—returning to the discussion in Section IV—the reason our signal modulates is as follows. The w NAF algorithm executes a (secret) sequence of double and add operations. In turn, these operations are sequences of finite

field additions, subtractions, multiplications, and squarings. Yet the number and order of these finite field operations are not identical for double and add. This is eventually reflected in their transient port utilization footprint.

C. Signal Processing Phase

After verifying the existence of SCA leakage in the captured TLS traces, we aim to extract the last double and add sequence to provide partial nonce information to the *key recovery phase*. Although visual inspection of the raw trace reveals the position of double and add operations, this is not enough to automatically and reliably extract the sequence due to noise and other signal artifacts.

Since our target is ECDSA point multiplication, we cropped it from the rest of the TLS handshake by applying a root-mean-square envelope over the entire trace. This resulted in a template used to extract the second point multiplication corresponding to the ECDSA signature generation. To further improve our results, we correlated the traces to the patterns found at the beginning and end of the point multiplication. This was possible as the beginning shows a clear pattern (trigger) due to OpenSSL precomputation, and the end of the trace has a sudden decrease in amplitude.

We then used a low pass filter on the raw point multiplication trace to remove any high frequency noise. Having previously located the end of point multiplication, we focused on isolating the add operations to get the last add peak, while estimating the doubles using their length. To accomplish this, we applied a source separation filtering method known as Singular Spectrum Analysis (SSA) [44]. SSA was first suggested in SCA literature for power analysis to increase signal to noise ratio in DPA attacks [45], and later used as a source separation tool for extracting add operations in an EM SCA attack on ECDSA [46]. We discuss the theoretical aspects of SSA in Appendix A.

For our purpose, we decided to threshold the SSA window size as suggested in [45]. Since the total length of the trace was around 15000 samples, this gave us a window size of 30. However, based on experimentation, a window of size 20 yielded optimal results using the second and the third component.

The traces occasionally encountered OS preemptions, corrupting them due to the Spy or Victim being interrupted. We detect Spy interrupts as high amplitude peaks or low amplitude gaps, depending on whether they happened while during or between latency measurement windows. Similarly, the Victim interrupts exhibit a low amplitude gap in our traces, since there was no Victim activity in parallel. In any case, we discarded all such traces (around 2.5%) when detecting any interrupt during the last double and add sequence.

Finally, by applying continuous wavelet transform [47] in the time-frequency domain we were able to detect the high energy add peaks, therefore isolating them. Moreover, a root-mean-square of the resulting peaks smoothed out any irregularities. Figure 6 illustrates the results of signal processing steps on a TLS trace from top to bottom. Even after applying

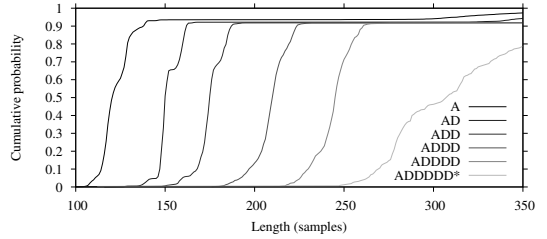


Fig. 7. Length distributions for various patterns at the end of scalar multiplication.

these steps, some traces where the adds were indistinguishable due to noise still occur, decreasing the accuracy of our results by about 2%.

The output of this phase, for each trace, is the distance from the last add operation to the end of the point multiplication: estimating the number of trailing doubles by counting the number of samples. Figure 7 depicts the CDF of the resulting sequences using our distance metric, having clear separation for each trailing double and add sequence.

D. Key Recovery Phase: Lattices

The output of the *signal processing phase* eventually provides us with partial nonce information, as the trailing sequence tells us the bit position of the lowest set bit. We then use this information to build a lattice attack to solve a *Hidden Number Problem*, retrieving the long-term private key used to sign the TLS handshakes. We build on previous work for our lattice attack, deferring to [48] for a more detailed mathematical description of the methodology. We use the BKZ reduction algorithm ($\beta = 20$) to efficiently look for solutions to the Shortest Vector Problem (SVP), closely following the construction by Benger et al. [41], yet with different parameters, and also a brute-force heuristic. In what follows, we: (1) describe exploring the lattice parameter space using traces modeled without errors; then (2) combine this study with profiling of the experimental data and the constraints of the computational resources at our disposal to launch a real-world end-to-end attack.

Exploration of the lattice parameter space: The main parameter to tune in implementing the lattice attack is the size (d) of the set of signatures used to build the lattice basis. Theoretically, given an infinite amount of time, if the selected subset of signatures does not contain any error and if the lattice embeds more bits of knowledge than the bit-length of the secret key, it should eventually succeed. In this scenario, optimizing for the smallest d that delivers enough bits of knowledge to recover the private key would be the preferred metric, as it requires less overall data from the *procurement phase* (lowering the risk of detection) and also improves success chances of the heuristic process (dealing with potential errors in the *signal processing phase*).

In a more realistic scenario we want to model the lattice parameters to succeed in a “reasonable” amount of time.

This definition is not rigorous and largely depends on the capabilities of a dedicated attacker: in this academic contest, constrained by the grid computing resources available to us, we define a lattice instance as *successful* if it correctly retrieves the secret key in under 4 hours when running on a single 2.50 GHz Xeon E5-2680 core (as done in Table 3 of [48]). We believe this definition is very conservative with respect to the potential computational resources of a nation-state level adversary or the availability and costs of dynamically scalable computing cloud services for individuals and organizations.

We modeled our preliminary experiments using random nonces, biased to have a trailing sequence of zero bits: this is equivalent to assuming error-free traces from the *signal processing phase*. We ran two sets of experiments, one with a bias mask of 0×3 , i.e., with at least two trailing zero bits (using the notation from [41], $z \geq 2$ and $l \geq 3$), and the other with a bias mask of 0×1 , i.e., with at least one trailing zero bit ($z \geq 1$ and $l \geq 2$).

To determine the optimal d for each bias case, we ran 10000 instances of the lattice algorithm against the two sets of modeled perfect traces and measured the corresponding amount of known nonce bits (Figure 10), the number of iterations for successful instances (Figure 11), the overall execution time for successful instances (Figure 12), and the *success* probability (Figure 9). The results indicate $d = 450$ is optimal for the 0×1 biased ideal traces, with success probability exceeding 90% coupled with a small number of iterations as well as overall execution time. Analogously, $d = 170$ is optimal for the 0×3 bias case.

Experimental parameters with real traces: Real traces come with errors, which lattices have no recourse to compensate for. The traditional solution is oversampling, using a larger set of t traces (with some amount e of traces with errors), and running in parallel a number (i) of lattice instances, each picking a different subset of size d from the larger set. Picking the subsets uniformly random, the probability for any subset to be error-free is:

$$\Pr(\text{No error in a random subset of size } d) = \frac{\binom{t-e}{d}}{\binom{t}{d}}$$

For typical values of $\{t, e, d\}$, the above probability is small and not immediately practical. But given the current capabilities for parallelizing workloads on computing clusters, repeatedly picking different subsets compensates:

$$\Pr(\geq 1 \text{ error-free subset over } i \text{ inst.}) = 1 - \left(1 - \frac{\binom{t-e}{d}}{\binom{t}{d}}\right)^i \quad (1)$$

Profiling the *signal processing phase* results, we determined to utilize thresholding to select traces belonging to the “AD”, “ADD”, “ADDD” and “ADDDD” distributions of Figure 7. In our setup, other traces are either useless for our lattice model or have too low accuracy. To ensure accuracy, we determined very narrow boundaries around the distributions to limit overlaps at the cost of very strict filtering. Out of the original 10000 captures, the filtering process selects a set

of 1959 traces with a 0×1 bias mask (i.e. nonces are even) including $e = 34$ (1.74%) errors. Combining this with $d = 450$ from our empirical lattice data, (1) leads us to $i \geq 36000$ instances required to achieve a probability $\geq 99\%$ of picking at least one subset without errors. This number of instances is beyond the parallel computational resources available to us, hence we move to the remaining case.

Filtering out also the 1060 traces categorized as “AD” delivers a total of 899 traces with a 0×3 bias mask (i.e. $k = 0 \pmod{4}$), including $e = 14$ (1.56%) errors. Combining this with $d = 170$ for the higher nonce bias and substituting in (1) leads us to $i \geq 200$ instances to achieve a probability $\geq 99.99\%$ of picking at least one subset without errors.

When using the actual attack data we noticed that while our filtering process increases accuracy, it has the side-effect of straying from the statistics determined in ideal conditions. We speculate this is due to filtering out longer trailing zero bits (low accuracy) decreasing the average amount of known nonce bits per signature, resulting in wider lattice dimensions with lower than expected useful information. This negatively affects the overall success rate and the amount of required iterations for a successful instance. We experimentally determined that when selecting only nonces with a bias mask between 0×3 and $0 \times F$, $d = 290$ compensates with a success rate (for an error-free subset) of 90.72%. Using these values in (1) leads us to $i = 2000$ instances to achieve a 99.97% probability of picking at least one subset without errors—well within the computing resources available to us.

Finally, running the entire process on the real data obtained from the *signal processing phase* on the original 10000 captures, using parameters $t = 899$, $e = 14$, and $d = 290$ over $i = 2000$ instances running in parallel on the described cluster resulted in 11 instances that successfully retrieved the secret key, the fastest of which terminated in 259 seconds after only two BKZ reduction iterations.

E. SGX

Intel Software Guard Extensions (SGX)⁴ is a microarchitecture extension present in modern Intel processors. SGX aims at protecting software modules by providing integrity and confidentiality to their code and memory contents. In SGX terminology, an SGX *enclave* is a software module that enjoys said protections. SGX was designed to defend processes against tampering and inspection from OS-privileged adversaries, providing strong isolation between enclave memory regions and the outer world. Despite these strong protections, side-channel attacks are still considered a major threat for enclaves, as SGX by itself does not protect against them [49]. In this regard, as the SGX threat model considers an OS-level adversary, it is even possible to mount more powerful side-channel attacks against enclaves where the measurement noise can be reduced considerably [50–52].

From a practical perspective, it is interesting to know which unprivileged side-channel techniques are a threat to

⁴<https://software.intel.com/en-us/sgx>

SGX enclaves. Regarding cache attacks, FLUSH+RELOAD and FLUSH+FLUSH do not apply in the unprivileged scenario since they require shared memory with the SGX enclave, which does not share its memory [15, 49]. However, researchers use other attack techniques against SGX, such as L1-based PRIME+PROBE attacks [52], and false dependency attacks [21]. It is worth mentioning that these methods assume an attacker with privileged access. However, we strongly believe these attacks would succeed without this assumption at the cost of capturing traces with a higher signal-to-noise ratio. Finally, TLBLEED [4] could be a potential successful attack technique against SGX, yet the authors leave it for future work.

The rest of this section analyzes PORTSMASH impact on SGX enclave security. Our first (strong) hypothesis is a PORTSMASH attack can target SGX enclaves transparently. The rationale relies on the difference between PORTSMASH root cause and the computing layer SGX protects. PORTSMASH aims at identifying the sequence of execution ports employed by a Victim process, while SGX provides protection at the memory subsystem level. This means that they operate at different computing layers—instruction execution and memory subsystem, respectively—providing soundness to our initial hypothesis. Nevertheless, for the sake of completeness we empirically evaluate our hypothesis, filling a research gap left by Covert Shotgun as an open problem: “*That would be especially interesting say in SGX scenarios*”. While developing an end-to-end attack like in previous sections on SGX enclaves might appear interesting, we instead focus on collecting sufficient experimental evidence to demonstrate that SGX enclaves do leak through the port contention side-channel—the most important contribution of this section.

For our experiments, we developed two Victim processes. One Victim is a standard process statically linked against OpenSSL 1.1.0h, and the other is an Intel SGX SSL enclave. Both Victim processes follow the same scalar multiplication code path analyzed in Section V, therefore we have two processes executing exactly the same code path with and without SGX protections.

Following the rationale that a PORTSMASH attack is oblivious to SGX enclaves, we applied the P5 strategy employed in Section V. We captured two traces on an Intel Core i7-7700HQ Kaby Lake, one for each setting: SGX and non-SGX. Figure 8 shows both the raw and filtered traces for each of them. Note the similarities between both raw traces, and after applying a noise reduction filter, the similarities become more evident since the position of adds are clearly revealed in both traces as amplitude peaks.

This demonstrates the leakage from SGX is essentially identical to the leakage outside SGX, validating our hypothesis that a PORTSMASH attack can be applied to SGX enclaves as well as to non-SGX processes. Therefore SGX enclaves do leak through port contention. The similarities in Figure 8 support the claim that developing an end-to-end attack against Intel SGX SSL should be straightforward, employing the tools explained in Section V. Moreover, it also shows two important characteristics: (1) the amount of noise does not significantly

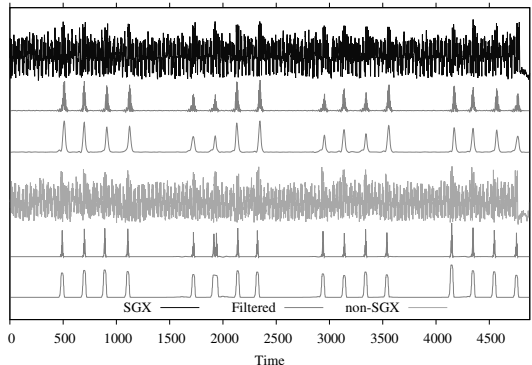


Fig. 8. From top to bottom: raw trace of our SGX Victim; said trace after filtering; raw trace of our user space Victim; said trace after filtering. Both victims received the same input, i.e., a scalar that induces 16 point adds at the end of the trace, clearly identifiable by the peaks in the filtered traces.

vary between both scenarios; and (2) PORTSMASH obliviousness regarding SGX as both traces were captured employing the same port contention strategy.

Furthermore, regardless of SGX the observant reader can also appreciate the similarities between traces in Figure 8 and Figure 6, demonstrating a PORTSMASH attack is also independent of the binary linking strategy (static vs dynamic).

VI. MITIGATIONS

A. Existing Work

Due to the copious amount of microarchitecture side-channel attacks in recent years, several countermeasures and mitigations appear in the literature; see [53] for a complete survey on countermeasures. From all the microarchitecture side-channel attacks proposed, cache-timing attacks and their respective techniques have arguably the most impact of all. This translates to the development of specific memory-based mitigations such as cache partitioning [11, 54], cache flushing [55, 56], and (partially) disabling caching [16]. Nevertheless, generally these solutions do not provide protections against non memory-based side-channels. To that end, another mitigation technique angle follows malware analysis methods. One way to categorize these countermeasures is by *binary* and *runtime* analysis.

Binary analysis looks for *code signatures* that allows classifying a binary as malicious or not. Irazoqui et al. [57] proposed MASCAT, a binary analysis framework for detecting microarchitecture malware. This framework analyzes a target binary by searching for a set of signature instructions often used during microarchitecture attacks, e.g., high-resolution timers, fence instructions, and cache-flushing instructions. Nevertheless, [15] showed that is possible to hide malicious code from static analysis of binaries.

Runtime analysis inspects potentially malicious processes while they execute, looking for dubious activities. Several

approaches propose microarchitecture attack mitigations [58–60]. Most of them focus mainly on monitoring hardware performance counters (HPC) to detect irregular execution patterns that may suggest an ongoing side-channel attack. Kulah et al. [58] and Zhang et al. [59] focus on unusual cache-activity rates, while Raj and Dharanipragada [60] aim at detecting an attack by measuring memory bandwidth differences.

Wichelmann et al. [61] recently proposed a combination of these categories. Their framework MicroWalk applies Dynamic Binary Instrumentation and Mutual Information Analysis to not only detect leakage in binaries, but also to locate the source of the leakage in the binary. The framework combines the memory footprint and the program control-flow to determine the side-channel leakage. They apply their technique successfully to closed source cryptographic libraries such as Intel IPP and Microsoft CNG.

From this brief survey, most of the work to mitigate microarchitecture side-channels is in the area of cache-based channels. Hence, many of these frameworks and techniques are not directly applicable to detect and mitigate our PORTSMASH technique. Since our technique does not target the cache, but instead focuses on the execution units, we argue it is extremely challenging to detect it. For example, when using an HPC-based countermeasure, it must distinguish normal port utilization between highly optimized code and PORTSMASH. At the end of the day, microprocessor manufacturers and code developers expect full core resource utilization. We agree that it is conceptually possible to adapt some of the previous countermeasures to detect our technique, but it is an open question how difficult, effective, and practical these countermeasures would be.

B. Recommendations

Our PORTSMASH technique relies on SMT and exploits transient microarchitecture execution port usage differences, therefore two immediate countermeasures arise: (1) remove SMT from the attack surface; and (2) promote execution port-independent code.

So far, the best and most recommended strategy against attacks relying on SMT—e.g., CacheBleed, MemJam, and TLBleed—is to simply disable this feature. Even OpenBSD developers⁵ recently followed this approach, since it is the simplest solution that exists but it comes at the cost of performance loss on thread-intensive applications. In order to minimize this loss, Wang and Lee [3] proposed a selective approach by modifying the OS to support logical core isolation requests from user space, such that security-critical code can trigger it on demand. This selective SMT-disabling reduces performance loss but is costly to implement since it requires changes in the OS and the underlying libraries, hindering portability and large-scale adoption.

The second option, port-independent code, can be achieved through secret-independent execution flow secure coding practices, similar to constant-time execution. Constant-time imple-

mentations that execute the same set of instructions independently from the secret—i.e., all code and data addresses are assumed public—fulfill the port-independent code requirement we propose to mitigate this technique. See Appendix B for a discussion on experimentally validating the effectiveness of this recommendation with respect to OpenSSL.

VII. CONCLUSION

We presented a new SCA technique exploiting timing information against a non-persistent shared HW resource, derived from port contention in shared CPU execution units on SMT architectures. Our PORTSMASH technique features interesting properties including high adaptability through various configurations, very fine spatial granularity, high portability, and minimal prerequisites. We demonstrated it is a practical attack vector with a real-world end-to-end attack against a TLS server, successfully recovering an ECDSA P-384 secret key; we further demonstrated it is a viable side-channel to endanger the security of SGX enclaves and discussed potential mitigations.

Following responsible disclosure procedures, we reported our findings to the manufacturer and OS vendors, which resulted in the assignment of CVE-2018-5407 to track the vulnerability. Subsequent to public disclosure, we released our proof-of-concept software to the open source community [62] in support of open science.

We leave as future work exploring the impact of memory ports for a PORTSMASH-like attack, answering the question: *are they more of a leakage or noise source?* It is also interesting to evaluate the capabilities of PORTSMASH on other architectures featuring SMT, especially on AMD Ryzen systems: our initial experiments suggest it is a viable security threat.

Finally, we conclude with a remark on how this work, together with the increasingly fast-paced publications of scientific results in the same field, confirms once again SCA as a practical and powerful tool to find, exploit—and eventually mitigate—significant and often underestimated threats to the security of our data and communications.

Acknowledgments

We thank Tampere Center for Scientific Computing (TCSC) for generously granting us access to computing cluster resources.

This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No 804476).

This article is based in part upon work from COST Action IC1403 CRYPTACUS, supported by COST (European Cooperation in Science and Technology).

We thank Nokia Foundation for funding a research visit of Alejandro Cabrera Aldaya to Tampere University during the development of this work.

⁵<https://marc.info/?l=openbsd-cvs&m=152943660103446>

REFERENCES

- [1] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, W. Enck and A. P. Felt, Eds. USENIX Association, 2018, pp. 973–990. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity18/presentation/lipp>
- [2] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy, SP 2019, Proceedings, 20-22 May 2019, San Francisco, California, USA*. IEEE, 2019, pp. 19–37. [Online]. Available: <https://doi.org/10.1109/SP.2019.00002>
- [3] Z. Wang and R. B. Lee, "Covert and side channels due to processor architecture," in *Proceedings of the 22nd Annual Conference on Computer Security Applications, ACSAC 2006, Miami Beach, FL, USA, December 11-15, 2006*. IEEE Computer Society, 2006, pp. 473–482. [Online]. Available: <https://doi.org/10.1109/ACSAC.2006.20>
- [4] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks," in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, W. Enck and A. P. Felt, Eds. USENIX Association, 2018, pp. 955–972. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity18/presentation/gras>
- [5] Y. Yarom, D. Genkin, and N. Heninger, "CacheBleed: A timing attack on OpenSSL constant time RSA," in *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, ser. Lecture Notes in Computer Science, B. Gierlichs and A. Y. Poschmann, Eds., vol. 9813. Springer, 2016, pp. 346–367. [Online]. Available: https://doi.org/10.1007/978-3-662-53140-2_17
- [6] D. Boggs, A. Baktha, J. Hawkins, D. T. Marr, J. A. Miller, P. Roussel, R. Singhal, B. Toll, and K. Venkatraman, "The microarchitecture of the Intel Pentium 4 processor on 90nm technology," *Intel Technology Journal*, vol. 8, no. 1, pp. 7–23, 2004.
- [7] "Intel 64 and IA-32 architectures software developers manual," Intel, Volume 1 253665-067US, May 2018. [Online]. Available: <https://software.intel.com/sites/default/files/managed/a4/60/253665-sdm-vol-1.pdf>
- [8] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton, "Hyper-Threading Technology architecture and microarchitecture," *Intel Technology Journal*, vol. 6, no. 1, 2002.
- [9] A. Fog, *Instruction tables (2018)*, Sep. 2018. [Online]. Available: http://www.agner.org/optimize/instruction_tables.pdf
- [10] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Retrospective: Simultaneous multithreading: Maximizing on-chip parallelism," in *25 Years of the International Symposia on Computer Architecture (Selected Papers)*, G. S. Sohi, Ed. ACM, 1998, pp. 115–116. [Online]. Available: <http://doi.acm.org/10.1145/285930.285971>
- [11] C. Percival, "Cache missing for fun and profit," in *BSDCan 2005, Ottawa, Canada, May 13-14, 2005, Proceedings*, 2005. [Online]. Available: <http://www.demonology.net/papers/cachemissing.pdf>
- [12] E. Brickell, G. Graunke, M. Neve, and J. Seifert, "Software mitigations to hedge AES against cache-based software side channel vulnerabilities," *IACR Cryptology ePrint Archive*, vol. 2006, no. 52, 2006. [Online]. Available: <http://eprint.iacr.org/2006/052>
- [13] O. Acicmez and J. Seifert, "Cheap hardware parallelism implies cheap security," in *Fourth International Workshop on Fault Diagnosis and Tolerance in Cryptography, 2007, FDTC 2007: Vienna, Austria, 10 September 2007*, L. Breveglieri, S. Gueron, I. Koren, D. Naccache, and J. Seifert, Eds. IEEE Computer Society, 2007, pp. 80–91. [Online]. Available: <https://doi.org/10.1109/FDTC.2007.4318988>
- [14] D. Evtushkin and D. V. Ponomarev, "Covert channels through random number generator: Mechanisms, capacity estimation and mitigations," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds. ACM, 2016, pp. 843–857. [Online]. Available: <http://doi.acm.org/10.1145/2976749.2978374>
- [15] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware guard extension: Using SGX to conceal cache attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment - 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings*, ser. Lecture Notes in Computer Science, M. Polychronakis and M. Meier, Eds., vol. 10327. Springer, 2017, pp. 3–24. [Online]. Available: https://doi.org/10.1007/978-3-319-60876-1_1
- [16] O. Acicmez, B. B. Brumley, and P. Grabher, "New results on instruction cache attacks," in *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010, Proceedings*, ser. Lecture Notes in Computer Science, S. Mangard and F. Standaert, Eds., vol. 6225. Springer, 2010, pp. 110–124. [Online]. Available: https://doi.org/10.1007/978-3-642-15031-9_8
- [17] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B. Yang, "High-speed high-security signatures," *J. Cryptographic Engineering*, vol. 2, no. 2, pp. 77–89, 2012. [Online]. Available: <https://doi.org/10.1007/s13389-012-0027-1>
- [18] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," in *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings*, ser. Lecture Notes in Computer Science, D. Pointcheval, Ed., vol. 3860. Springer, 2006, pp. 1–20. [Online]. Available: https://doi.org/10.1007/11605805_1
- [19] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. USENIX Association, 2014, pp. 719–732. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom>
- [20] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+flush: A fast and stealthy cache attack," in *Detection of Intrusions and Malware, and Vulnerability Assessment - 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings*, ser. Lecture Notes in Computer Science, J. Caballero, U. Zurutuza, and R. J. Rodríguez, Eds., vol. 9721. Springer, 2016, pp. 279–299. [Online]. Available: https://doi.org/10.1007/978-3-319-40667-1_14
- [21] A. Moghimi, T. Eisenbarth, and B. Sunar, "MemJam: A false dependency attack against constant-time crypto implementations in SGX," in *Topics in Cryptology - CT-RSA 2018 - The Cryptographers' Track at the RSA Conference 2018, San Francisco, CA, USA, April 16-20, 2018, Proceedings*, ser. Lecture Notes in Computer Science, N. P. Smart, Ed., vol. 10808. Springer, 2018, pp. 21–44. [Online]. Available: https://doi.org/10.1007/978-3-319-76953-0_2
- [22] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 2015, pp. 605–622. [Online]. Available: <https://doi.org/10.1109/SP.2015.43>
- [23] G. I. Apecechea, T. Eisenbarth, and B. Sunar, "SSA: A shared cache attack that works across cores and defies VM sandboxing - and its application to AES," in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 2015, pp. 591–604. [Online]. Available: <https://doi.org/10.1109/SP.2015.42>
- [24] M. Kayaalp, N. B. Abu-Ghazaleh, D. V. Ponomarev, and A. Jaleel, "A high-resolution side-channel attack on last-level cache," in *Proceedings of the 53rd Annual Design Automation Conference, DAC 2016, Austin, TX, USA, June 5-9, 2016*. ACM, 2016, pp. 72:1–72:6. [Online]. Available: <http://doi.acm.org/10.1145/2897937.2897962>
- [25] Q. Ge, Y. Yarom, T. Chothia, and G. Heiser, "Time protection: the missing OS abstraction," *CoRR*, vol. abs/1810.05345, 2018. [Online]. Available: <http://arxiv.org/abs/1810.05345>
- [26] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "ARMageddon: Cache attacks on mobile devices," in *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, T. Holz and S. Savage, Eds. USENIX Association, 2016, pp. 549–564. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/lipp>
- [27] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of Computation*, vol. 48, no. 177, pp. 203–209, 1987.
- [28] V. S. Miller, "Use of elliptic curves in cryptography," in *Advances in Cryptology - CRYPTO '85, Santa Barbara, California, USA, August 18-22, 1985, Proceedings*, ser. Lecture Notes in Computer Science, H. C. Williams, Ed., vol. 218. Springer, 1985, pp. 417–426. [Online]. Available: https://doi.org/10.1007/3-540-39799-X_31
- [29] A. H. Koblitz, N. Koblitz, and A. Menezes, "Elliptic curve

- cryptography: The serpentine course of a paradigm shift,” *Journal of Number Theory*, vol. 131, no. 5, pp. 781–814, 2011. [Online]. Available: <https://doi.org/10.1016/j.jnt.2009.01.006>
- [30] “Digital signature standard (DSS),” National Institute of Standards and Technology, FIPS PUB 186-2, Jan. 2000.
- [31] *The Case for Elliptic Curve Cryptography*, National Security Agency, Oct. 2005. [Online]. Available: tinyurl.com/NSAandECC
- [32] “National information assurance policy on the use of public standards for the secure sharing of information among national security systems,” Committee on National Security Systems, CNSSP 15, Oct. 2012. [Online]. Available: <https://www.cnss.gov/CNSS/issuances/Policies.cfm>
- [33] *Commercial National Security Algorithm Suite*, National Security Agency, Aug. 2015. [Online]. Available: <https://apps.nsa.gov/iaarchive/programs/iaid-initiatives/cnsa-suite.cfm>
- [34] “Commercial national security algorithm suite and quantum computing FAQ,” National Security Agency, MFQ-UOO 815099-15, Jan. 2016. [Online]. Available: <https://apps.nsa.gov/iaarchive/library/ia-guidance/ia-solutions-for-classified/algorithm-guidance/cnsa-suite-and-quantum-computing-faq.cfm>
- [35] E. Käsper, “Fast elliptic curve cryptography in OpenSSL,” in *Financial Cryptography and Data Security - FC 2011 Workshops, RLCPS and WECSR 2011, Rodney Bay, St. Lucia, February 28 - March 4, 2011, Revised Selected Papers*, ser. Lecture Notes in Computer Science, G. Danezis, S. Dietrich, and K. Sako, Eds., vol. 7126. Springer, 2011, pp. 27–39. [Online]. Available: https://doi.org/10.1007/978-3-642-29889-9_4
- [36] B. B. Brumley and R. M. Hakala, “Cache-timing template attacks,” in *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009, Proceedings*, ser. Lecture Notes in Computer Science, M. Matsui, Ed., vol. 5912. Springer, 2009, pp. 667–684. [Online]. Available: https://doi.org/10.1007/978-3-642-10366-7_39
- [37] S. Gueron and V. Krasnov, “Fast prime field elliptic-curve cryptography with 256-bit primes,” *J. Cryptographic Engineering*, vol. 5, no. 2, pp. 141–151, 2015. [Online]. Available: <https://doi.org/10.1007/s13389-014-0090-x>
- [38] J. Merkle and M. Lochter, “Elliptic Curve Cryptography (ECC) Brainpool Standard Curves and Curve Generation,” Internet Requests for Comments, RFC Editor, RFC 5639, Mar. 2010. [Online]. Available: <https://datatracker.ietf.org/doc/rfc5639/>
- [39] B. Möller, “Algorithms for multi-exponentiation,” in *Selected Areas in Cryptography, 8th Annual International Workshop, SAC 2001 Toronto, Ontario, Canada, August 16-17, 2001, Revised Papers*, ser. Lecture Notes in Computer Science, S. Vaudenay and A. M. Youssef, Eds., vol. 2259. Springer, 2001, pp. 165–180. [Online]. Available: https://doi.org/10.1007/3-540-45537-X_13
- [40] T. Allan, B. B. Brumley, K. E. Falkner, J. van de Pol, and Y. Yarom, “Amplifying side channels through performance degradation,” in *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016*, S. Schwab, W. K. Robertson, and D. Balzarotti, Eds. ACM, 2016, pp. 422–435. [Online]. Available: <http://doi.acm.org/10.1145/2991079.2991084>
- [41] N. Benger, J. van de Pol, N. P. Smart, and Y. Yarom, ““Ooh Aah... Just a Little Bit”: A small amount of side channel can go a long way,” in *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014, Proceedings*, ser. Lecture Notes in Computer Science, L. Batina and M. Robshaw, Eds., vol. 8731. Springer, 2014, pp. 75–92. [Online]. Available: https://doi.org/10.1007/978-3-662-44709-3_5
- [42] N. Tuveri, S. ul Hassan, C. Pereida García, and B. B. Brumley, “Side-channel analysis of SM2: A late-stage featurization case study,” in *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. ACM, 2018, pp. 147–160. [Online]. Available: <https://doi.org/10.1145/3274694.3274725>
- [43] J. van de Pol, N. P. Smart, and Y. Yarom, “Just a little bit more,” in *Topics in Cryptology - CT-RSA 2015, The Cryptographer’s Track at the RSA Conference 2015, San Francisco, CA, USA, April 20-24, 2015, Proceedings*, ser. Lecture Notes in Computer Science, K. Nyberg, Ed., vol. 9048. Springer, 2015, pp. 3–21. [Online]. Available: https://doi.org/10.1007/978-3-319-16715-2_1
- [44] R. Vautard, P. Yiou, and M. Ghil, “Singular-spectrum analysis: A toolkit for short, noisy chaotic signals,” *Physica D: Nonlinear Phenomena*, vol. 58, no. 1, pp. 95 – 126, 1992. [Online]. Available: [https://doi.org/10.1016/0167-2789\(92\)90103-T](https://doi.org/10.1016/0167-2789(92)90103-T)
- [45] S. M. D. Pozo and F. Standaert, “Blind source separation from single measurements using singular spectrum analysis,” in *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, ser. Lecture Notes in Computer Science, T. Güneşu and H. Handschuh, Eds., vol. 9293. Springer, 2015, pp. 42–59. [Online]. Available: https://doi.org/10.1007/978-3-662-48324-4_3
- [46] D. Genkin, L. Pachmanov, I. Pipman, E. Tromer, and Y. Yarom, “ECDSA key extraction from mobile devices via nonintrusive physical side channels,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds. ACM, 2016, pp. 1626–1638. [Online]. Available: <http://doi.acm.org/10.1145/2976749.2978353>
- [47] I. Daubechies, “The wavelet transform, time-frequency localization and signal analysis,” *IEEE Trans. Information Theory*, vol. 36, no. 5, pp. 961–1005, 1990. [Online]. Available: <https://doi.org/10.1109/18.57199>
- [48] C. Pereida García and B. B. Brumley, “Constant-time callees with variable-time callers,” in *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, E. Kirda and T. Ristenpart, Eds. USENIX Association, 2017, pp. 83–98. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/garcia>
- [49] V. Costan and S. Devadas, “Intel SGX explained,” *IACR Cryptology ePrint Archive*, vol. 2016, no. 86, 2016. [Online]. Available: <http://eprint.iacr.org/2016/086>
- [50] Y. Xu, W. Cui, and M. Peinado, “Controlled-channel attacks: Deterministic side channels for untrusted operating systems,” in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 2015, pp. 640–656. [Online]. Available: <https://doi.org/10.1109/SP.2015.45>
- [51] A. Moghimi, G. Irazoqui, and T. Eisenbarth, “CacheZoom: How SGX amplifies the power of cache attacks,” in *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, ser. Lecture Notes in Computer Science, W. Fischer and N. Homma, Eds., vol. 10529. Springer, 2017, pp. 69–90. [Online]. Available: https://doi.org/10.1007/978-3-319-66787-4_4
- [52] F. Dall, G. D. Micheli, T. Eisenbarth, D. Genkin, N. Heninger, A. Moghimi, and Y. Yarom, “CacheQuote: Efficiently recovering long-term secrets of SGX EPID via cache attacks,” *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2018, no. 2, pp. 171–191, 2018. [Online]. Available: <https://doi.org/10.13154/tches.v2018.i2.171-191>
- [53] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, “A survey of microarchitectural timing attacks and countermeasures on contemporary hardware,” *J. Cryptographic Engineering*, vol. 8, no. 1, pp. 1–27, 2018. [Online]. Available: <https://doi.org/10.1007/s13389-016-0141-6>
- [54] H. Raj, R. Nathuji, A. Singh, and P. England, “Resource management for isolation enhanced cloud services,” in *Proceedings of the first ACM Cloud Computing Security Workshop, CCSW 2009, Chicago, IL, USA, November 13, 2009*, R. Sion and D. Song, Eds. ACM, 2009, pp. 77–84. [Online]. Available: <http://doi.acm.org/10.1145/1655008.1655019>
- [55] Y. Zhang and M. K. Reiter, “Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud,” in *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS 2013, Berlin, Germany, November 4-8, 2013*, A. Sadeghi, V. D. Gligor, and M. Yung, Eds. ACM, 2013, pp. 827–838. [Online]. Available: <http://doi.acm.org/10.1145/2508859.2516741>
- [56] M. M. Godfrey and M. Zulkernine, “Preventing cache-based side-channel attacks in a cloud environment,” *IEEE Trans. Cloud Computing*, vol. 2, no. 4, pp. 395–408, 2014. [Online]. Available: <https://doi.org/10.1109/TCC.2014.2358236>
- [57] G. Irazoqui, T. Eisenbarth, and B. Sunar, “MASCAT: Preventing microarchitectural attacks before distribution,” in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy, CODASPY 2018, Tempe, AZ, USA, March 19-21, 2018*, Z. Zhao, G. Ahn, R. Krishnan, and G. Ghinita, Eds. ACM, 2018, pp. 377–388. [Online]. Available: <http://doi.acm.org/10.1145/3176258.3176316>
- [58] Y. Kulah, B. Dincer, C. Yilmaz, and E. Savas, “SpyDetector: An approach for detecting side-channel attacks at runtime,” *International Journal of Information Security*, Jun. 2018. [Online]. Available:

<https://doi.org/10.1007/s10207-018-0411-7>

- [59] T. Zhang, Y. Zhang, and R. B. Lee, "CloudRadar: A real-time side-channel attack detection system in clouds," in *Research in Attacks, Intrusions, and Defenses - 19th International Symposium, RAID 2016, Paris, France, September 19-21, 2016, Proceedings*, ser. Lecture Notes in Computer Science, F. Monrose, M. Dacier, G. Blanc, and J. García-Alfaro, Eds., vol. 9854. Springer, 2016, pp. 118–140. [Online]. Available: https://doi.org/10.1007/978-3-319-45719-2_6
- [60] A. Raj and J. Dharanipragada, "Keep the PokerFace on! Thwarting cache side channel attacks by memory bus monitoring and cache obfuscation," *J. Cloud Computing*, vol. 6, p. 28, 2017. [Online]. Available: <https://doi.org/10.1186/s13677-017-0101-4>
- [61] J. Wichelmann, A. Moghimi, T. Eisenbarth, and B. Sunar, "MicroWalk: A framework for finding side channels in binaries," in *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. ACM, 2018, pp. 161–173. [Online]. Available: <https://doi.org/10.1145/3274694.3274741>
- [62] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. Pereira Garcia, and N. Tuveri, *PortSmash Proof-of-Concept exploit*. Zenodo, Jan. 2019. [Online]. Available: <https://doi.org/10.5281/zenodo.2552315>
- [63] I. Markovsky, "Structured low-rank approximation and its applications," *Automatica*, vol. 44, no. 4, pp. 891–909, 2008. [Online]. Available: <https://doi.org/10.1016/j.automatica.2007.09.011>
- [64] "SEC 2: Recommended Elliptic Curve Domain Parameters," Standards for Efficient Cryptography, Standards for Efficient Cryptography Group, SEC 2, Jan 2010. [Online]. Available: <http://www.secg.org/sec2-v2.pdf>
- [65] V. Dolmatov and A. Degtyarev, "GOST R 34.10-2012: Digital Signature Algorithm," Internet Requests for Comments, RFC Editor, RFC 7091, Dec. 2013. [Online]. Available: <https://datatracker.ietf.org/doc/rfc7091/>
- [66] Y. Nir, S. Josefsson, and M. Pégourié-Gonnard, "Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS) Versions 1.2 and Earlier," Internet Requests for Comments, RFC Editor, RFC 8422, Aug. 2018. [Online]. Available: <https://datatracker.ietf.org/doc/rfc8422/>

APPENDIX A SINGULAR SPECTRUM ANALYSIS

To improve the detection of add operations in the scalar multiplications, we applied the filtering technique Singular Spectrum Analysis (SSA) [44].

The SSA filter performs an eigen-spectra decomposition of the original signal using a trajectory matrix into different components which are then analyzed and selected accordingly for reconstructing a filtered signal. The first step *embedding* converts the single dimension signal $\{m_k\}_{k=1}^N$ of length N into a multidimensional trajectory matrix M which contains I column vectors each of size w where $I = N - w + 1$. The window size $1 < w \leq N/2$ dictates the quality and performance of the reconstruction phase. The second step *singular value decomposition* (SVD) decomposes the trajectory matrix M into non-zero eigenvalues λ_k of MM^T sorted in decreasing ranks of their magnitudes along with their corresponding eigenvectors u_k . With $v_k = M^T u_k \sqrt{\lambda_k}$ and $Y_k = u_k v_k$ the projection matrices, SVD can be shown as:

$$M = \sum_{k=1}^d \sqrt{\lambda_k} Y_k^T$$

To obtain the reconstructed components $\{y_i\}_{i=1}^N$, next perform a diagonal averaging also known as Hankelization by computing the average over the skewed diagonal of the projection

matrices Y_k [63]. The original signal can thus be reproduced by summing all the reconstructed components:

$$\{m_i\}_{i=1}^N = \sum_{k=1}^d \{y_i^k\}_{i=1}^N$$

For source separation, only the useful components can be chosen, leaving out the noisy ones from all the d possible choices.

APPENDIX B ECC IN OPENSLL

As stated in Section V-A, OpenSSL features several implementations for ECC operations: each elliptic curve has an associated method structure containing function pointers to common ECC operations, and for this work we specifically focus on the scalar multiplication operation.

The actual method structure associated with a particular ECC cryptosystem depends on a variety of factors, including the OpenSSL version, the particular curve instantiated, build-time options, and capabilities of the targeted architecture. The intent of this section is to discuss PORTSMASH applications to OpenSSL ECC outside of P-384 and across different OpenSSL versions and build-time options.

A. OpenSSL versions

The OpenSSL project currently actively supports three releases of the library:

- 1.0.2 is the old *long-term support* (LTS) release, supported until the end of 2019;
- 1.1.0 is the previous non-LTS release, currently in its final year of support, thus updated only with security fixes;
- 1.1.1 the latest LTS release.

Letter releases (e.g., 1.0.2a) are periodically issued and exclusively contain bug and security fixes and no new features; *minor releases* (i.e., where only the last number is changed) contain new features, but in a way that does not break binary compatibility, so that existing applications do not need to be recompiled; finally, *major releases* (e.g., from 1.0.2 to 1.1.0) can contain major changes precluding both binary and API compatibility, thus requiring applications to be recompiled against the new version of the library, and, potentially, significant changes in the application source code to adapt to API changes.

It should be noted that the OpenSSL library is often installed as an OS system library or bundled with binary application packages, and as a result in most cases users depend on third party vendors, such as OS distribution maintainers, for the version of the library used by their applications and for its build-time options.

This is particularly relevant in terms of support for bugs and security fixes, as often the release strategies of third party vendors are not compatible with that of the OpenSSL project (see [?] for a discussion), resulting in major delays between upstream releases and versions installed in the majority of systems. For example, currently the latest Ubuntu Linux

LTS release (18.04)—used in many online servers—features OpenSSL version 1.1.0g that is more than one year older than the latest upstream letter release (1.1.0j) for that release branch.

B. Scalar multiplication implementations

Excluding `curve25519` and `curve448`, which are defined separately, scalar multiplications for prime curves in the OpenSSL library are handled by one of the following implementations:

- `EC_GFp_nistp256_method`, based on [35], offering a timing-resistant portable C implementation for 64-bit architectures supporting 128-bit integer types, optimized for the NIST P-256 curve (with variants for NIST P-224/P-521). Support for these three methods is conditional to the aforementioned architecture and compiler support and must be explicitly enabled at compilation time;
- `EC_GFp_nistz256_method`, based on [37], offers a faster timing-resistant method for NIST P-256, using Intel AVX2 SIMD assembly instructions to increase the performance of finite field operations. This method is automatically enabled at compilation time if the target architecture supports the Intel AVX2 SIMD instructions, unless assembly optimized implementations are explicitly disabled at build time;
- non constant-time multiplication, based on [39, Sec. 3.2] using a modified windowed Non-Adjacent Form (*wNAF*) for scalar representation. It was the default generic implementation in OpenSSL 1.1.0h and earlier (and up to the later 1.0.2p version in the 1.0.2 branch). This is the code path used in the end-to-end attack presented in this work;
- (only in 1.1.1+, 1.1.0i+, 1.0.2q+) timing-resistant generic implementation based on a Montgomery ladder, featuring a fixed sequence of operations without scalar-dependent branches; it was introduced during the development of version 1.1.1 (and backported to the older release branches starting with releases version 1.1.1i and 1.0.2q) as a result of work by Tuveri et al. [42], further discussed in Appendix B-C.

Of the 39 standard defined prime curves supported by OpenSSL 1.0.2 and 1.1.0 releases, only the aforementioned NIST P-224, P-256 and P-521 have specialized timing-resistant implementations. Every other prime curve will use the generic default implementation, which will be one of the last two implementations in the above list, depending on the OpenSSL library version. Among these curves, it is worth mentioning:

- “Brainpool” (RFC 5639[38]) curves;
- most prime curves standardized by SECG [64], including the `secp256k1` curve (adopted for Bitcoin operations);
- any generic prime curve defined over custom parameters, e.g., when using the `gost`⁶ engine to implement

⁶<https://github.com/gost-engine/engine>

RFC 7091[65] or when using explicit arbitrary parameters in TLS versions 1.2 and earlier—a feature that has been recently deprecated (RFC 8422[66]) but is still supported for compatibility with legacy products.

Moreover, the specialized implementations for the three NIST curves are not enabled if any of the mentioned requirements is not met, so depending on architecture and compilation options, even these curves could fall back to the default generic implementation targeted in the end-to-end attack we demonstrated. This is particularly relevant, considering that often users rely on third party vendors for a binary distribution of the library, and said vendors could prioritize portability over performance and disable architecture-dependent features at build time.

C. Relevant mitigations in OpenSSL

The default *wNAF* scalar multiplication implementation has been the target of several side-channel attacks [36, 40–43]. Independently from this current work, this implementation was finally replaced during the development cycle of OpenSSL 1.1.1 with a timing-resistant one, as a consequence of [42]. The set of changes described there fulfills the port-independent code requirement we propose to mitigate the PORTSMASH technique.

This changeset was backported to the 1.1.0 branch and released starting with version 1.1.0i. But at the time, backporting to the LTS 1.0.2 branch was deemed unnecessary, due to additional complexity in the backport process (caused by major design differences), and a lower coverage for automated detection of regressions and new defects. It was only as a result of this work and the disclosure of CVE-2018-5407 that the additional effort and risks were considered acceptable⁷, thus we backported⁸ the changes to the 1.0.2 branch. For the old LTS branch, our mitigation has been released since version 1.0.2q.

⁷<https://www.openssl.org/news/secadv/20181112.txt>

⁸<https://github.com/openssl/openssl/pull/7593>

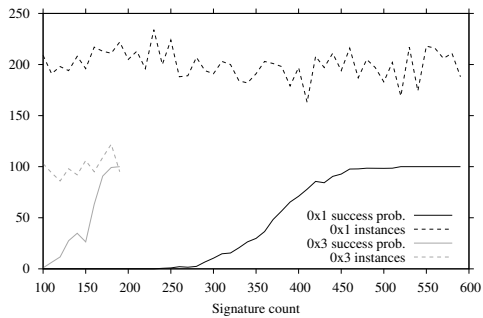


Fig. 9. Probability (percentage) of *success* of the lattice algorithm against the two sets of modeled perfect traces. The dashed lines track the number of instances running with different signature counts.

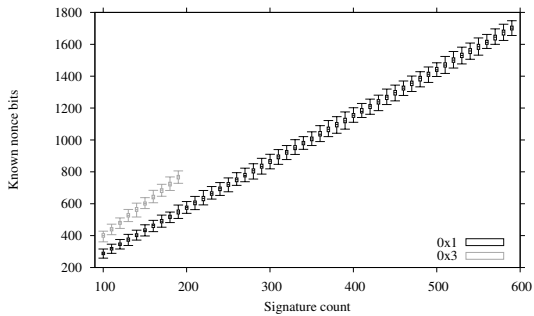


Fig. 10. Cumulative known nonce bits for the lattice algorithm against the two sets of modeled perfect traces, varying the count of signatures used to model the lattice.

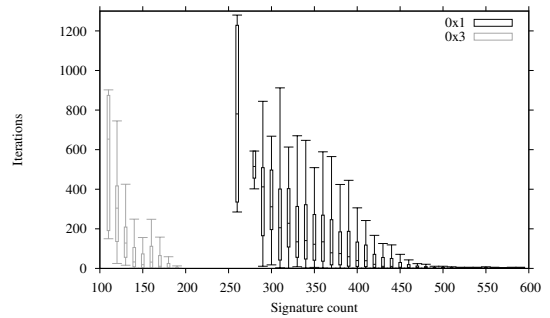


Fig. 11. Number of BKZ iterations for *successful* instances of the lattice algorithm against the two sets of modeled perfect traces, varying the count of signatures used to model the lattice.

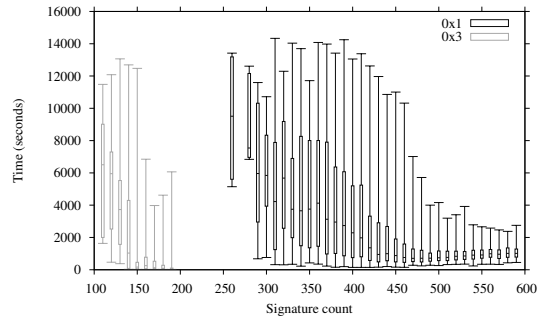


Fig. 12. Execution time of *successful* instances of the lattice algorithm against the two sets of modeled perfect traces, varying the count of signatures used to model the lattice.

PUBLICATION VII

Certified Side Channels

C. Pereida García, S. ul Hassan, N. Tuveri, I. Gridin, A. C. Aldaya and
B. B. Brumley

29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020.

Ed. by S. Capkun and F. Roesner. 2020, 2021–2038

Publication reprinted with the permission of the copyright holders



Certified Side Channels

Cesar Pereida García, Sohaib ul Hassan, Nicola Taveri, and Iaroslav Gridin, Tampere University; Alejandro Cabrera Aldaya, Tampere University and Universidad Tecnológica de la Habana; Billy Bob Brumley, Tampere University

<https://www.usenix.org/conference/usenixsecurity20/presentation/garcia>

**This paper is included in the Proceedings of the
29th USENIX Security Symposium.**

August 12–14, 2020

978-1-939133-17-5

**Open access to the Proceedings of the
29th USENIX Security Symposium
is sponsored by USENIX.**



Certified Side Channels

Cesar Pereida García¹, Sohaib ul Hassan¹, Nicola Tuveri¹,
Iaroslav Gridin¹, Alejandro Cabrera Aldaya^{1,2}, and Billy Bob Brumley¹

¹Tampere University, Tampere, Finland

{cesar.pereidagarcia,n.sohaibulhassan,nicola.tuveri,iaroslav.gridin,billy.brumley}@tuni.fi

²Universidad Tecnológica de la Habana (CUJAE), Habana, Cuba
aldaya@gmail.com

Abstract

We demonstrate that the format in which private keys are persisted impacts Side Channel Analysis (SCA) security. Surveying several widely deployed software libraries, we investigate the formats they support, how they parse these keys, and what runtime decisions they make. We uncover a combination of weaknesses and vulnerabilities, in extreme cases inducing completely disjoint multi-precision arithmetic stacks deep within the cryptosystem level for keys that otherwise seem logically equivalent. Exploiting these vulnerabilities, we design and implement key recovery attacks utilizing signals ranging from electromagnetic (EM) emanations, to granular microarchitecture cache timings, to coarse traditional wall clock timings.

1 Introduction

Academic SCA tends to focus on implementations of cryptographic primitives in isolation. With this view, the assumption is that any higher level protocol or system built upon implementations of these primitives will naturally benefit from SCA mitigations in place at lower levels.

Our work questions this assumption, and invalidates it with several concrete vulnerabilities and attacks against modern software libraries: we dub these *Certified Side Channels*, since the novel attack vector is deeply rooted in cryptography standards. For this vector, “certified” is in the certificate sense (e.g. X.509), not in the Common Criteria sense. Counter-intuitively, we demonstrate that the format in which keys are stored plays a significant role in real world SCA security. Detailed security recommendations for key persistence are scarce; e.g. FIPS 140-2 vaguely states “*Cryptographic keys stored within a cryptographic module shall be stored either in plaintext form or encrypted form [...] Documentation shall specify the key storage methods employed by a cryptographic module*” [1, 4.7.5].

There are (at least) two high level dimensions at play regarding key formats as an SCA attack vector: (i) Among

the multitude of standardized cryptographic key formats to choose from when persisting keys: *which one to choose, and does the choice matter?* Surprisingly, it does—we demonstrate different key formats trigger different behavior within software libraries, permeating all the way down to the low level arithmetic for the corresponding cryptographic primitive. (ii) At the specification level, alongside required parameters, standardized key formats often contain optional parameters: *does including or excluding optional parameters impact security?* Surprisingly, it does. We demonstrate that omitting optional parameters can cause extremely different execution flows deep within a software library, and also that two keys seemingly mathematically identical at the specification level can be treated by a software library as inequivalent, again reaching very different arithmetic code deep within the library.

Furthermore, we demonstrate that key parsing in general is a lucrative SCA attack vector. This is due mostly to software engineering constraints. Complex libraries inevitably stray to convoluted data structures containing generous nesting levels to meet the demands of broad standardized cryptography. This is exacerbated by the natural urge to handle keys generically when faced with extremely diverse cryptographic standards spanning RSA, DSA, ECDSA, Ed25519, Ed448, GOST, SM2, etc. primitives. The motivation behind this generalization is to abstract away underlying cryptographic details from application developers linking against a library—more often than not, these developers are not cryptography experts. Nevertheless, we observe that when loading keys modern security libraries make varying design choices that ultimately impact SCA security. From the functionality perspective, these design choices are sensible; from the security perspective, we demonstrate they are often questionable.

Outline. Section 2 gives an overview of the related background and previous work. Section 3 discusses the vulnerabilities discovered as a result of our analysis, with microarchitecture SCA evaluations on OpenSSL RSA, DSA, and mbedTLS RSA. We also demonstrate end-to-end attacks on OpenSSL ECDSA using timing and EM side channels in Section 4. We

conclude in Section 5.

2 Background

2.1 Public Key Cryptography

ECDSA. Denote an order- n generator $G \in E$ of an elliptic curve group E with cardinality fn and n a large prime and f the small cofactor. The user's private key α is an integer uniformly chosen from $\{1..n-1\}$ and the corresponding public key is $D = [\alpha]G$. With approved hash function $\text{Hash}()$, the ECDSA digital signature (r, s) on message m (denoting with $h < n$ the representation of $\text{Hash}(m)$ as an integer) is

$$r = ([k]G)_x \bmod n, \quad s = k^{-1}(h + \alpha r) \bmod n \quad (1)$$

where k is a nonce chosen uniformly from $\{1..n-1\}$.

RSA. According to the PKCS #1 v2.2 standard (RFC 8017 [55]), an RSA private key consists of the eight parameters $\{N, e, p, q, d, d_p, d_q, i_q\}$ where all but the first two are secret, and $N = pq$ for primes p, q . Public exponent e is usually small and the following holds:

$$d = e^{-1} \bmod \text{lcm}(p-1, q-1) \quad (2)$$

In addition, Chinese Remainder Theorem (CRT) parameters are stored for speeding up RSA computations:

$$d_p = d \bmod p, \quad d_q = d \bmod q, \quad i_q = q^{-1} \bmod p \quad (3)$$

2.2 Key Formats

Interoperability among different software and hardware platforms in handling keys and other cryptographic objects requires common standards to serialize and deserialize such objects. *ASN.1* or *Abstract Syntax Notation One* is an interface description language to define data structures and their (de)serialization, standardized [69] jointly by ITU-T and ISO/IEC since 1984 and widely adopted. It supports several encoding rules, among which the *Distinguished Encoding Rules* (DER), a binary format ensuring uniqueness and concision, has been preferred for the representation of cryptographic objects. PEM (RFC 7468 [45]) is a textual file format to store and transmit cryptographic objects, widespread despite being originally developed as part of the now obsolete IETF standards for *Privacy-Enhanced Mail* after which it is named. PEM uses *base64* to encode the binary DER serialization of an object, providing some degree of human readability and support for text-based protocols like e-mail and HTTP(S).

Object Identifiers. The *ASN.1* syntax also defines an `OBJECT IDENTIFIER` primitive type which represents a globally unique identifier for an object. ITU-T and ISO jointly manage a decentralized hierarchical registry of object identifiers or `OID`s. The registry is organized as a tree structure, where every node is authoritative for its descendants, and

decentralization is obtained delegating the authority on subtrees to entities such as countries and organizations. This mechanism solves the problem of assigning globally unique identifiers to entities to facilitate global communication.

RSA private keys. PKCS #1 (RFC 8017 [55]) also defines the *ASN.1* DER encoding for an RSA private key, defining an item for each of its eight parameters. As further discussed in Section 3.4, the standard does not strictly require implementations to include all the eight parameters during serialization, nor to invalidate the object during deserialization if one of the parameters is not included.

EC private keys. The ANSI X9.62 standard [51] is the normative reference for the definition of the ECDSA cryptosystem and the encoding of ECDSA public keys, but omits a serialization for private keys. The SEC1 standard [2] follows ANSI X9.62 for the public key *ASN.1* and provides a DER encoding also for EC private keys, but allows generous variation as it seems to assume different encapsulating options depending on different protocols in which the EC private key can be used. Flexibility in the format brings complexity in the deserializer implementation, that needs to be stateful w.r.t. parsing of the container of the private key encoding and flexible enough to interoperate with other implementations and interpretations of the standards: this already suggests that the parsing stage shows potential as a lucrative SCA attack vector. The SEC1 *ASN.1* notation for `ECPrivateKey` contains the private scalar as an octet string, an optional (depending on the container) `ECDomainParameters` field, and an optional bit string field to include the public part of the key pair. The `ECDomainParameters` can be null, if the curve parameters are specified in the container encapsulating the `ECPrivateKey`, or contain either an `OID` for a “named” curve, or a `SpecifiedECDomain` structure. The latter, simplifying, contains a description of the field over which the EC group is defined, the definition of the curve equation in terms of the coefficients of its Short Weierstrass form, an encoding of the EC base point, and its order n . Finally it can *optionally* contain a component to represent a small cofactor f as defined at the beginning of this section. In Section 3.1 we will further discuss about the security consequences caused in actual implementations by the logic required to support the cofactor as an optional field.

MSBLOB key format. MSBLOB is the OpenSSL implementation of Microsoft's private key BLOB format¹ supporting different cryptosystems, using custom defined structures and fields. DSS key BLOB uses an arbitrary structure, while RSA key BLOBs follows PKCS #1 with minor differences. To identify each cryptosystem, a “magic member” is used in the key BLOB structure—the member is the hexadecimal representation of the ASCII encoding of the cryptosystem name, e.g. “RSA1”, “RSA2”, “DSS1”, “DSS2”, etc., where the integer

¹<https://docs.microsoft.com/en-us/windows/win32/seccrypto/base-provider-key-blobs>

dictates if it is a public or a private key. Public and private key BLOBs are stored as binary files in little-endian order and by default the private key BLOBs are not encrypted—it is up to the developers to choose whether to encrypt the key. Microsoft created the public and private key BLOBs in order to support cryptographic service providers (CSP), i.e. third party cryptographic software modules. It is worth noting that both private and public BLOBs are independent from each other, thus allowing a CSP to only support and implement the desired format according to the cryptosystem in use, meaning that public keys can be computed on-demand using the private key BLOB information.

PVK key format. The PriVate Key (PVK) format is a Microsoft proprietary key format used in Windows supporting signature generation using both DSA and RSA private keys. Little information is available about this format but a key is typically composed of a header containing metadata, and a body containing a private key BLOB structure as per the previous description. Following the same idea as in the private key BLOB, the PVK header metadata contains the “magic” value `0xb0b5f11e`² to uniquely identify this key format. Additionally, PVK’s header contains metadata information for key password protection, preventing the storage of private key information in plain text. Unfortunately, PVK is an outdated format and it only supports RC4 encryption, moreover, in some cases PVK keys use a weakened encryption key to comply with the US export restrictions imposed during the 90’s³.

2.3 Side-Channel Analysis

SCA is a cryptanalysis technique used to target software and hardware implementations of cryptographic primitives. The main goal of SCA is to expose hidden algorithm state by measuring variations in time, power consumption, electromagnetic radiation, temperature, and sound. These variations might leak data or metadata that allows the retrieval of confidential information such as private keys and passwords. The history of SCA is long and rich—from the military program called TEMPEST [31] to current commodity PCs, SCA has deeply impacted security-critical systems and it has reached the most popular and widely used cryptosystems over the years such as AES, DSA, RSA, and ECC, implemented in the most widely used cryptographic libraries including OpenSSL, BoringSSL, LibreSSL, and mbedTLS.

SCA can be broadly categorized (w.r.t. signal procurement techniques) in two specific research fields: hardware and software. Both fields have evolved and developed their own techniques, and the line separating them has blurred as research improves, and attacks become more complex. Nevertheless, the ultimate goal is still the same: extract confidential informa-

tion from a device executing vulnerable cryptographic code. A brief overview follows.

Hardware. Ever since their inception, System-on-Chip (SoC) embedded devices have become passively ubiquitous in the form of mobile devices and IoT, performing security critical tasks over the Internet. Their basic building blocks—in terms of performing computations—are the CMOS transistors, drawing current during the switching activity to depict the behavior of logic gates. Power analysis attacks introduced by Kocher et al. [49] rely on the fact that accumulated switching activity of these transistors influence the overall power fluctuations while secret data dependent computations take place on the processor and memory subsystems.

While power analysis is one way to perform SCA, devices may also leak sensitive information through other means such as EM [5], acoustic [34], and electric potential [36]. In contrast to the power side channels which require physically tapping onto the power lines, EM and acoustic based SCA add a spatial dimension. There may be slight differences when it comes to acquiring and processing these signals, but in essence the concept is similar to traditional power analysis, hence the hardware based SCA techniques generally apply to all.

Over the years more powerful SCA techniques have emerged such as differential power analysis [49], correlation power analysis [19], template attacks [25], and horizontal attacks [12]. Most of these techniques rely on statistical methods to find small secret data dependent leakages.

Traditionally, hardware SCA research mainly focuses on architecturally simpler devices such as smart cards and microcontrollers [52, 65, 66]. Being simple here does not imply that developing and deploying such cryptosystems is simpler, rather in terms of their functionality and hardware architecture. Modern consumer electronics (e.g. smart phones) are more feature rich, containing SoC components, memory subsystems and multi-core processors with clock speeds in gigahertz. These devices are often running a full operating system (several in fact) making it possible to deploy software libraries such as OpenSSL. More recently, a new class of hardware side channel attacks on embedded, mobile devices and even PCs has emerged, targeting crypto software libraries such as OpenSSL [38, 50], GnuPG [34, 35, 36, 37], PolarSSL [29], Android’s Bouncy Castle [13], and WolfSSL [68]. They employ various signal processing tools to counter the noise induced by complex systems and microarchitectures. For further details, Tunstall [71] present an elaborate discussion on hardware based SCA techniques, while Danger et al. [27] and Abarzúa et al. [3] sum up various SCA attacks and their countermeasures.

Software. The widespread use of e-commerce and the need for security on the Internet sparked the development of cryptographic libraries such as OpenSSL. Researchers quickly began analyzing these libraries and it took a short time to find security flaws in these libraries. Impulsed by Kocher’s

²Leetspeak for “bobsfile”!

³<http://justsolve.archiveteam.org/wiki/PVK>

work [48], SCA timing attacks quickly gained traction. By measuring the amount of time required to perform private key operations, the author demonstrated that it was feasible to find Diffie-Hellman exponents, factor RSA keys, and recover DSA keys. Later Brumley and Boneh [23] demonstrated that it was possible to do the same but remotely, by measuring the response time from an OpenSSL-powered web server. Other TLS-level timing attacks include [47] with a software target and [53] with a hardware target.

As software SCA became more complex and sophisticated, a new subclass of attacks denominated “microarchitecture attacks” emerged. Typically, a modern CPU executes multiple programs either concurrently or via time-sharing, increasing the need to optimize resource utilization to obtain high performance. To achieve this goal, microarchitecture components try to predict future behavior and future resource usage based on past program states. Based on these observations, researchers [15, 60] discovered that some microarchitecture components—such as the memory subsystem—work wonderfully as communication channels. Due to their shared nature between programs, some of the microarchitecture components can be used to violate access control and achieve inter-process communication. Among these components, researchers noticed that the memory subsystem is arguably the easiest to exploit: by observing the memory footprint an attacker can leak algorithm state from an executing cryptographic library in order to obtain secret keys. Since the initial discovery, several SCA techniques have been developed to extract confidential data from different memory levels and under different threat models. Some of these techniques include FLUSH+RELOAD [78], PRIME+PROBE [59], EVICT+TIME [59], and FLUSH+FLUSH [42]. Moreover, recent research [9, 24, 74] shows that most (if not all) microarchitecture components shared among programs are a security hazard since they can potentially be used as side-channels. Ge et al. [33] provide a great overview on software SCA, including the types of channels, microarchitecture components, side-channel attacks, and mitigations.

2.4 Lattice Attacks

In Section 4 we present two attacks against ECDSA signing that differ in SCA technique, but share a common pattern: (i) gathering several (r, s, m) tuples in a collection phase, using SCA to infer partial knowledge about the nonce used during signature generation; (ii) a recovery phase combines the collected tuples and the associated partial knowledge to retrieve the long-term secret key.

To achieve the latter, we recur to the common strategy of constructing *hidden number problem* (HNP) [18] instances from the collected information, and then use lattice techniques to find the secret key. In this section we discuss the lattice technique used to recover the private keys.

We follow the formalization used in [61], which itself

builds on the work by Nguyen and Shparlinski [57, 58], that assumed a fixed amount of known bits (denoted ℓ) for each nonce used in the lattice, but also includes the improvements by Bengier et al. [14], using ℓ_i and a_i to represent, respectively, the amount of known bits and their value on a per-equation basis.

The collection phase of [61] as well as our Section 4.2 attack recovers information regarding the LSBs of each nonce, hence it annotates the nonce associated with i -th equation as $k_i = W_i b_i + a_i$, with $W_i = 2^{\ell_i}$, where ℓ_i and a_i are known, and since $0 < k_i < n$ it follows that $0 \leq b_i \leq n/W_i$. Denote $\lfloor x \rfloor_n$ modular reduction of x to the interval $\{0..n-1\}$ and $\lfloor x \rfloor_{(n-1)/2}$ to the interval $\{-(n-1)/2..(n-1)/2\}$. Combining (1), define (attacker-known) values $t_i = \lfloor r_i / (W_i s_i) \rfloor_n$ and $\hat{u}_i = \lfloor (a_i - h_i / s_i) / W_i \rfloor_n$, then $0 \leq \lfloor \alpha t_i - \hat{u}_i \rfloor_n < n/W_i$ holds. Setting $u_i = \hat{u}_i + n/2W_i$ we obtain $v_i = |\alpha t_i - u_i|_n \leq n/2W_i$, i.e. integers λ_i exist such that $\text{abs}(\alpha t_i - u_i - \lambda_i n) \leq n/2W_i$ holds. Thus u_i approximate αt_i since they are closer than a uniformly random value from $\{1..n-1\}$, leading to an instance of the HNP [18]: recover α given many (t_i, u_i) pairs.

Consider the rational $d + 1$ -dimension lattice generated by the rows of the following matrix.

$$B = \begin{bmatrix} 2W_1 n & 0 & \dots & \dots & 0 \\ 0 & 2W_2 n & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & 0 & \vdots \\ 0 & \dots & 0 & 2W_d n & 0 \\ 2W_1 t_1 & \dots & \dots & 2W_d t_d & 1 \end{bmatrix}$$

Denoting $\vec{x} = (\lambda_1, \dots, \lambda_d, \alpha)$, $\vec{y} = (2W_1 v_1, \dots, 2W_d v_d, \alpha)$, and $\vec{u} = (2W_1 u_1, \dots, 2W_d u_d, 0)$, then $\vec{x}B - \vec{u} = \vec{y}$ holds. Solving the Closest Vector Problem (CVP) with inputs B and \vec{u} yields \vec{x} , and hence the private key α . Finally, as in [61], we embed the CVP into a Shortest Vector Problem (SVP) using the classical strategy [39, Sec. 3.4], and employ an extended search space heuristic [32, Sec. 5].

The presence of outliers among the results of the collection phase usually has a detrimental effect on the chances of success of the lattice attack. The traditional solution is to oversample, filtering $t > d$ traces from the collection phase if d traces are required to embed enough leaked information in the lattice instance to solve the HNP. Indicating with e the amount of traces with errors in the filtered set of size t , picking a subset of size d uniformly at random, the probability for any such subset to be error-free is $\hat{p} = \binom{t-e}{d} / \binom{t}{d}$. For typical values of $\{t, e, d\}$, \hat{p} will be small. Viewing the process of randomly picking a subset and attempting to solve the resulting lattice instance as a Bernoulli trial, the number of expected trials before first success is $1/\hat{p}$. So an attacker can compensate for small \hat{p} by running $j = 1/\hat{p}$ jobs in parallel.

2.5 Triggerflow

Triggerflow [40] is a tool for tracking execution paths, previously used to facilitate SCA of OpenSSL. After users mark up source code with annotations of Points Of Interest (POI) and filtering rules for false positive considerations, Triggerflow runs the binary executable under a debugger and records the execution paths that led up to POIs. The user supplies binary invocation lines called “triggers”. These techniques are useful in SCA of software, where areas that do not execute in constant time are known and the user needs to find code that leads up to them. The authors designed Triggerflow with continuous integration (CI) in mind, and maintain an automatic testing setup which continuously monitors all non-EOL branches of OpenSSL for new vulnerabilities by watching execution flows that enter known problematic areas.

Triggerflow is intended for automated regression testing and has no support for automatic POI detection. Thus offensive leakage detection methodologies including (but certainly not limited to) CacheAudit [28], templating [21, 41], CacheD [75], and DATA [77] complement Triggerflow to establish POIs. One approach is to apply these leakage detection methodologies, filter out false positives, limit to functions deemed security-critical and worth tracking, then use the result to add Triggerflow source code annotations for CI. See [40, Sec. 7] for a more extensive discussion.

3 Vulnerabilities

We used Triggerflow to analyze several code paths on multiple cryptographic libraries, discovering SCA vulnerabilities across OpenSSL and mbedTLS. In this section, we discuss these vulnerabilities, including the unit tests we developed for Triggerflow that detected each of them, then identify the root cause in each case. Following Figure 1, Triggerflow executes each line of the unit tests given in a text file. Triggerflow will trace the execution of lines beginning with `debug` to detect break points getting hit at SCA-critical points in the code. Each such line is security critical—in these examples, generating a key pair or using the private key to e.g. digitally sign a message. Hence if Triggerflow encounters said break points during execution, it represents a potential SCA vulnerability. We compiled the target executables (and shared libraries) with debug symbols, and source code annotated using Triggerflow’s syntax to mark previously known SCA-vulnerable functions. Lines that do not begin with `debug` are not traced by Triggerflow, merely executed as preparation steps for subsequent triggers (e.g. setting up public fixed parameters).

Vulnerability-wise, the main results of this section are as follows: (i) bypassing SCA countermeasures using ECC explicit parameters (Section 3.1, OpenSSL); (ii) bypassing SCA countermeasures for DSA using PVK and MSBLOB key formats (Section 3.2, OpenSSL); (iii) bypassing SCA countermeasures for RSA by invoking key validation (Section 3.3,

```
1 # ECDSA with explicit curve parameters, zero cofactor
2 debug openssl genpkey -algorithm EC -pkeyopt ec_paramgen_curve:P-256
3   -pkeyopt ec_param_enc:explicit -outform DER -out p256.der
4 sed -i 's/\x25\x51\x02\x01\x01/\x25\x51\x02\x01\x00/' p256.der
5 debug openssl dgst -sha256 -sign p256.der -keyform DER -out /dev/null
6   - /etc/lsb-release
7
8 # DSA with PVK key format
9 openssl genpkey -genparam -algorithm DSA -out dsa.params -pkeyopt
10   dsa_paramgen_bits:1024 -pkeyopt dsa_paramgen_q_bits:160
11 debug openssl genpkey -paramfile dsa.params -out dsa.pkey
12 debug openssl dsa -in dsa.pkey -outform PVK -pvk-none -out dsa.pvk
13 debug openssl dgst -sha1 -sign dsa.pvk -keyform PVK -out /dev/null
14   - /etc/lsb-release
15
16 # DSA with MSBLOB key format
17 openssl genpkey -genparam -algorithm DSA -out dsa.params -pkeyopt
18   dsa_paramgen_bits:1024 -pkeyopt dsa_paramgen_q_bits:160
19 debug openssl genpkey -paramfile dsa.params -out dsa.pkey
20 debug openssl dsa -in dsa.pkey -outform MS\ PRIVATEKEYBLOB -out dsa.blob
21 debug openssl dgst -sha1 -sign dsa.blob -keyform MS\ PRIVATEKEYBLOB
22   -out /dev/null /etc/lsb-release
23
24 # RSA key validation in OpenSSL
25 openssl genrsa -out rsa.pem 2048
26 debug openssl rsa -in rsa.pem -check
27 debug openssl pkey -in rsa.pem -check
28
29 # RSA key loading in mbedTLS
30 create_rsa.pem.sh without_d > custom.pem
31 debug mbedtls_pk_sign custom.pem
```

Figure 1: New Triggerflow unit tests.

OpenSSL); (iv) bypassing SCA countermeasures for RSA through key loading (Section 3.4, mbedTLS).

3.1 ECC: Bypass via Explicit Parameters

From a standardization perspective, curve data for ECC key material gets persisted in one of two ways: either including the specific OID that points to a named curve with fixed parameters, or explicitly specifying the curve with ASN.1 syntax. Mathematically, they seem equivalent. To explore the potential difference in security implications between these options, we constructed three keys: (i) a NIST P-256 private key as a named curve, using the `ec_param_enc:named_curve` argument to the OpenSSL `genpkey` utility; (ii) a NIST P-256 private key with explicit curve parameters, using the `ec_param_enc:explicit` argument; (iii) a copy of the previous key, but post-modified with the OpenSSL `asn1parse` utility to remove the optional cofactor. The first two keys additionally used the `ec_paramgen_curve:P-256` argument to specify the target curve. We highlight that, from a standards perspective, all three of these keys are valid. We then integrated the commands to produce these keys into the Triggerflow framework as unit tests. Finally, we added an OpenSSL `dgst` utility unit test for each of these keys in Triggerflow, to induce ECDSA signing. What follows is a discussion on the three distinct control flow cases for each key, regarding the security-critical scalar multiplication operation.

Named curve. Triggerflow indicated `ecp_nistz256_points_mul` handled the operation. The reason for this is OpenSSL uses an `EC_METHOD` structure for legacy ECC; the assignment of structure instances to specific curves happens at library compile time, allowing different curves to have different (optimized) implementations depending on archi-

ture and compiler features. This particular function is part of the `EC_GFp_nistz256_method`, an `EC_METHOD` optimized for AVX2 architectures [43]. The implementation is constant time, hence this is the best case scenario.

Explicit parameters. Triggerflow indicated `ec_scalar_mul_ladder` handled the operation, through the default `EC_GFp_simple_method`, the generic implementation for curves over prime fields. In fact this is the oldest `EC_METHOD` in the codebase, present since ECC support appeared in 2001. The implementation of this particular function was mainlined in 2018 [72] as a result of CVE-2018-5407 [9], SCA-hardening generic curves with the standard Montgomery ladder. Interpreting this Triggerflow result, we conclude OpenSSL has no runtime mechanism to match explicit parameters to named curves present in the library. Ideally, it would match the explicit parameters to `EC_GFp_nistz256_method` for improved performance and SCA resistance. Failure to do so bypasses one layer of SCA mitigations, but in this particular case the default method still features sufficient SCA hardening.

Explicit parameters, no cofactor. Triggerflow indicated `ec_wNAF_mul` handled the operation through the same `EC_METHOD` as the previous case. This is a known SCA-vulnerable function since 2009 [21], and is a POI maintained in the Triggerflow patchset to annotate OpenSSL for automated CI. Root causing the failed Triggerflow unit test, the function only early exits to the SCA-hardened ladder if both the curve generator order and the curve cardinality cofactor are non-zero. Since the optional cofactor is not present in the key, the library assigns zero as the default, indicating either the provided cofactor was zero or not provided at all. The OpenSSL ladder implementation utilizes the cofactor as part of SCA hardening, hence the code unfortunately falls through to the SCA-insecure version in this case, bypassing the last layer of SCA defenses for scalar multiplication. This is the path we will exploit in Section 4.

Keys in the wild. While we reached a vulnerable code path through a standards-compliant, valid, non-malicious key, the fact is the OpenSSL CLI will not organically emit a key in this form. One can argue that OpenSSL is far from the only security tool that produces keys conforming to the specification, that it must subsequently parse since they are valid. Nevertheless, this leaves us with the question: *do keys like this exist—does this vulnerability matter?* Investigating, we at least found two deployment classes this vulnerability affects: (i) The GOST engine⁴ for OpenSSL, dynamically adding support for Russian cryptographic primitives in RFC 4357 [64]. Since the curves from the standard are not built-in to OpenSSL, the engine programatically constructs the curve based on fixed parameters inside the engine. However, since the cofactor parameter to the OpenSSL `EC_GROUP_set_generator` API is optional, the engine developers omit it in earlier versions,

passing `NULL`. When GOST keys are persisted, they have their own OID distinct from legacy ECC standards and only support named curves; however, the usage of these curves within the engine hits the same exact code path. (ii) GOSTCoin⁵ is the official software stack for a cryptocurrency. It links against OpenSSL for cryptographic functionality, but does not support the GOST engine. Examining the digital wallet, we manually extracted several DER-encoded legacy (OID-wise) ECC private keys from the binary. Parsing these keys revealed they are private keys with explicit parameters from RFC 4357 [64], “Parameter Set A”. Upon closer inspection, the cofactor is present in the `ASN.1` encoding, yet explicitly set to zero. Similar to the previous case, this is due to failure to supply the correct cofactor to the OpenSSL EC API when constructing the curve.

From this brief study, we can conclude that failure to provide the valid cofactor to the OpenSSL EC API when constructing curves programatically (the only choice for curves not built-in to the library), or importing a (persisted) ECC private key with explicit parameters containing a zero or omitted (spec-optional) cofactor are characteristics of applications affected by this vulnerability.

Related work. Concurrent to our work, Takahashi and Tibouchi [70] utilize explicit parameters in OpenSSL to mount a fault injection attack. They invasively induce a fault during key parsing to change OpenSSL’s representation of a curve coefficient. This causes decompression of the explicit generator point to emit a point on a weaker curve, subsequently mounting a degenerate curve attack [56]. At a high level, the biggest differences from our work are the invasive attack model and limited set of applicable curves.

Subsequent to our work, CVE-2020-0601 tracks the “Curveball” vulnerability. It affects the Windows CryptoAPI and uses ECC explicit parameters to match a named curve in all but the custom generator point, allowing to spoof code-signing certificates.

3.2 DSA: Bypass via Key Formatting

As the Swiss knife of cryptography, OpenSSL provides support for PVK and MSBLOB key formats to perform digital signatures using DSA. In fact, OpenSSL has supported these formats since version 1.0.0, hence the library has a dedicated file in `crypto/pem/pvkfmt.c` for parsing these keys. The file contains all the logic to parse Microsoft’s DSA and RSA private key BLOBs, common to both PVK and MSBLOB key formats. Unfortunately, the bulk of code for parsing the keys has seen few changes throughout the years, and more importantly it has missed important SCA countermeasures that other parts of the code base have received [62], allowing this vulnerability to go unnoticed in all OpenSSL branches until now.

⁴<https://github.com/gost-engine/engine>

⁵<https://github.com/GOSTSec/gostcoin>

As mentioned previously, PVK and MSBLOB key files contain only private key material but OpenSSL expects the public key to be readily available. Thus every time it loads any of these key formats, the library computes the corresponding public key. More specifically, the upper level function `b2i_dss` reads the private key material and subsequently calls the `BN_mod_exp` function to compute the public key using the default modular exponentiation function, without first setting the constant-time flag `BN_FLG_CONSTTIME`. Note that this vulnerability does not depend on whether the PVK key is encrypted or not, because when the code reaches the `b2i_dss` function, the key has been already decrypted, and the modular exponentiation function is already leaking private key material. This default SCA-vulnerable modular exponentiation algorithm follows a square-and-multiply approach—first pre-computes a table of multipliers, and then accesses the table during the square-and-multiply step. Already in 2005 Percival [60] demonstrated an L1 data cache-timing attack against this function during RSA decryption. We found that the original flaw is still present, but this time in the context of DSA.

Figure 2 demonstrates the side-channel leakage obtained by our L1 data-cache malicious spy process running in parallel with OpenSSL during a modular exponentiation operation while computing the DSA public key using PVK and MSBLOB key formats. Using the PRIME+PROBE technique, our spy process is able to measure the latency of accessing a specific cache set (y-axis) over time (x-axis) to obtain a sequence of pre-computed multipliers accessed during computation. In OpenSSL a multiplier is represented as a `BIGNUM` structure spanning approximately across three different cache sets. Reading from top-to-bottom and left-to-right, and after a brief period of noise, the figure shows that every block of approximately three continuous high latency cache sets corresponds to a multiplier access. An attacker can not only trace the multipliers accessed, but also the order in which they were accessed during the exponentiation, leaking more than half of the exponent bits. This information greatly reduces the effort to perform full key recovery. Moreover, the public key is computed every time the private key is loaded, thus an attacker has several attempts at tracing the sequence of operations performed during the exponentiation. Our experiments reveal that cache sets stay constant across multiple invocations of modular exponentiation, reducing the attacker’s effort and permitting the use of statistical techniques to improve the leakage quality.

Keys in the wild. PVK and MSBLOB are based on MS proprietary private key formats—nevertheless they are widely found in use in open source software. MSBLOB keys are supported by MS Smart Card CSP and OpenSC⁶, an open source software library for smart cards linking to OpenSSL. In fact, OpenSC has a function⁷ that creates a key container—by call-

⁶<https://github.com/OpenSC/OpenSC>

⁷<https://github.com/OpenSC/OpenSC/blob/master/src/minidriver/minidriver.c#L3308>

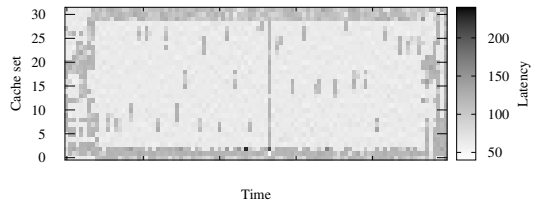


Figure 2: L1 dcache trace showing distinctive access patterns to pre-computed multipliers in cache sets 6-8, 9-11, 13-16, 15-17, 22-24, 25-27, 28-30 during DSA public key computation.

ing the OpenSSL vulnerable function—whenever “the card either does not support internal key generation or the caller requests that the key be archived in the card”, facilitating the attack in a smart card setting. On the other hand, MS Visual Studio 2019 provides tools⁸ to generate, convert, and sign Windows drivers, libraries, and catalog files using the PVK format. In a typical workflow, `MakeCert` generates certificates and the corresponding private key, then `Pvk2Pfx` encapsulates private keys and certificates in a PKCS #12 container, and finally `SignTool` signs the driver. Interestingly, `MakeCert` and `SignTool` successfully generate keys and signatures using RSA and DSA, but `Pvk2Pfx` fails to accept any key that is not RSA—a gap filled by the vulnerable OpenSSL, creating compliant PKCS #12 keys. Other libraries such as `jsign`⁹, `osslsigncode`¹⁰, and the Mono Project¹¹ exist to provide signing capabilities using MS proprietary private key formats outside of Windows. We can expect this vulnerability to be exploitable by an attacker targeting Windows developers.

3.3 RSA: Bypass via Key Validation

RSA key validation is a common operation required in a cryptography library supporting RSA to verify that an input key is indeed a valid RSA key. We found that OpenSSL function `RSA_check_key_ex` located at `crypto/rsa/rsa_chk.c` contains several SCA vulnerabilities. In fact, we found that the affected function `RSA_check_key_ex` can be accessed by two public entry points: a direct call to `RSA_check_key`, and through the public EVP interface calling `EVP_PKEY_check` on an RSA key. Figure 1 shows the commands in OpenSSL leading to the affected code path through the two different public functions. Note that any external, OpenSSL-linking application calling any of these two public functions is also affected.

The check function takes as input an RSA key, parses the

⁸<https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/tools-for-signing-drivers>

⁹<https://ebourg.github.io/jsign/>

¹⁰<https://sourceforge.net/projects/osslsigncode/files/osslsigncode/>

¹¹<https://www.mono-project.com/>

key, and reads all of the private and public components, checking the correctness of all the components. In general, the function validates the primality of p and q , then it recomputes the rest of the values $\{N, d, d_p, d_q, i_q\}$ to compare against the parsed values and check their validity. Unfortunately, we found that in several cases OpenSSL uses by default SCA-vulnerable functions to recompute these secret values.

Primality testing vulnerabilities. The prime values p and q are the first components verified during the process. The verification is done using the Miller-Rabin primality test [67] as implemented in the function `BN_is_prime_fasttest_ex`. This function calls a lower level *witness* function named `bn_miller_rabin_is_prime`¹² where a b base value is chosen randomly to compute $b^m \bmod p$, in which p is the candidate prime and the relation $2^m = p - 1$ holds. The *witness* exponentiation is performed using the `BN_mod_exp_mont` function, where unfortunately the `BN_FLG_CONSTTIME` is not set beforehand. Thus a variable-time sliding window exponentiation is used, allowing a malicious process to potentially perform a data cache-timing attack to recover half of the bits from the exponent [60]. This is enough information to recover both prime values p and q . Moreover, the exponentiation function gets called several times by the *witness* function with different b values in order to obtain confidence about the prime values, providing multiple attempts for an attacker to capture the leakage and perform error correction during its key recovery attack.

In addition to the previous vulnerability, as part of the *witness* function, a Montgomery setup phase occurs in `BN_MONT_CTX_set`, where the inverse of $2^w \bmod p$ for w -bit architectures is computed. The modular inverse function `BN_mod_inverse` is called without setting the constant-time flag. The inverse operation uses a variation of the greatest common divisor (GCD) algorithm, which is dependent on its inputs $\{2^w, p \bmod 2^w\}$, thus leaking algorithm state equivalent to the least significant word of both p and q .

Secret value vulnerabilities. Once the prime values p and q are deemed correct, the key validation continues by computing the rest of the secret components where more vulnerabilities are found. To compute the private exponent d during the verification code path, OpenSSL uses the least common multiple (LCM) of $p - 1$ and $q - 1$. Nevertheless, this operation is computed as

$$\text{lcm}(p-1, q-1) = \frac{(p-1) \cdot (q-1)}{\text{gcd}(p-1, q-1)} \quad (4)$$

performing the GCD computation using the `BN_gcd` function. This function does not have an early exit to a constant-time function, instead it completely ignores the flag existence, so even if it was set it would not have any effect on the code path taken. Finally, the last vulnerability is observed during CRT i_q computation. OpenSSL computes this parameter using the

¹²In OpenSSL 1.0.2 the function is called *witness*.

`BN_mod_inverse` function, which yet again fails to properly set the constant-time flag, leaving the computation $q^{-1} \bmod p$ unprotected.

It is worth noting that variable-time GCD functions, and variants, potentially leak all the algorithm state. Depending on the attacker capabilities [6], an attacker is fully capable of recovering the input values, i.e. p and q .

As can be observed, all of the vulnerabilities leak on p and q at different degrees, but by combining all the leaks, an attacker can use the redundancy and number-theoretic constraints to correct errors and obtain certainty on the bits leaked.

Keys in the wild. Surprisingly, the vulnerabilities presented in this section do not depend on a special key format. In fact, the vulnerabilities are triggered whenever an RSA key is checked for validity using the OpenSSL library, thus a potential attacker could simply wait for the right moment to exploit these vulnerabilities. The potential impact of these vulnerabilities is large, but it is minimized by two important factors: the user must trigger an RSA key validation; and the attacker must be collocated in the same CPU as the user. Nevertheless, this is not a rare scenario, and thus exploitation is very much possible.

3.4 RSA: Bypass via Missing Parameters

Recalling Section 2, an RSA private key is composed by some redundant parameters while at the same time not all of them are mandatory per RFC 8017 [55]: “An RSA private key should be represented”. This implies that cryptography implementations must deal with RSA private keys that do not contain all parameters, requiring potentially computing them on demand. Natural questions arise: (i) *How do software libraries handle this uncertainty?* (ii) *Does this uncertainty mask SCA threats?* Shifting focus from OpenSSL, the remainder of this section analyzes the open source mbedTLS library in this regard.

Fuzzing RSA private key loading. Following the Triggerflow methodology, we developed unit tests for the mbedTLS library, specifically for targeting RSA key loading code paths. To this end, we analyzed the mbedTLS v2.18.1 `bignum` implementation and set three POIs for Triggerflow: (i) GCD computation, `mbedtls_mpi_gcd`; (ii) Modular multiplicative inverse, `mbedtls_mpi_inv_mod`; (iii) Modular exponentiation, `mbedtls_mpi_exp_mod`. We arrived at these POIs from state-of-the-art SCA applied to cryptography libraries where these operations are commonly exploited. The first two functions are based on the binary GCD algorithm, previously shown weak to SCA [4, 7, 10, 61, 76], while exponentiation is a classical SCA target [17, 26, 35, 49, 62].

With these POIs, we fuzz the RSA mbedTLS private key loading code path to identify possible vulnerabilities. The fuzzing consists of testing the loading of an RSA private key when some parameters are equal to zero (i.e. empty PKCS #1 parameter).

After configuring the potential leaking functions as Triggerflow POIs, we created an RSA private key fuzzing utility that generates all possible combinations of PKCS #1-compliant private keys. This ranges from a private key that includes all PKCS #1 parameters to none. While the latter is clearly invalid as it carries no information, other missing combinations could be interesting regarding SCA. As PKCS #1 defines eight parameters, the number of private key combinations compliant with this standard is 256.

Triggerflow provides a powerful framework for testing all these combinations smoothly. Using Triggerflow for each of these private keys, we tested the generic function of mbedTLS for loading public keys: `mbedtls_pk_parse_keyfile`. The advantage of using Triggerflow for this task is that we can automate the whole process of testing each code corner of this execution path, searching for SCA threats. Figure 1 (bottom) shows a Triggerflow unit test of one of these parameter combinations, with a private key missing d . Unit tests for the other combinations are similar.

Results. For each combination, we obtained a report that indicates if and where POIs were hit or not, also recording the program return code. A quick analysis of the generated reports indicates the 256 combinations group in four classes (i.e. only four unique reports were generated for all 256 private key parameter combinations). Table 1 shows the number of keys for each group. The majority of private key combinations yield an “Invalid” return code without hitting a POI before returning.

The group “Public” contains those remaining *valid* private keys for which $\{d, p, q\}$ is not a subset of included parameters. In this case, mbedTLS recognized the key as a public key even if the CRT secret parameters are present. Nevertheless, identified as “Public” by mbedTLS, we ignore them, since no secret data processing takes place.

Table 1: Report groups for the 256 private keys.

Group	Number of keys
Invalid	216
Public	8
POI-hit (CRT)	16
POI-hit (CRT & d)	16

The last two groups in Table 1 contain those private keys (32 in total) that indeed hit at least one POI. Analyzing both reports on these groups, we identified two potential leakage points. One is related to processing of the CRT parameters, and the other to computation of the private exponent d . We now investigate if these hits represent an SCA threat. Appendix A details the complete list of parameter combinations that hit a POI.

Leakage analysis: CRT. The last two report groups have at least one hit at a Triggerflow POI in a CRT related computation. In both groups, the report regarding this code path is

identical, hence the following analysis applies to both.

The Triggerflow report reveals hitting the modular inverse POI; the parent function is `mbedtls_rsa_deduce_crt`, computing the CRT parameters in (3) as $i_q = q^{-1} \bmod p$ using `mbedtls_mpi_inv_mod`. It is a variant of the binary extended Euclidean algorithm (BEEA) with an execution flow highly dependent on its inputs, therefore an SCA vulnerability. This is similar to OpenSSL’s Section 3.3 vulnerability. Yet in contrast to OpenSSL, this code path in mbedTLS executes every time this library loads a private key: the vulnerability exists regardless of missing parameters in the private key.

Leakage analysis: private exponent. The last group in Table 1 contains the CRT leakage previously described in addition to one related to private exponent d processing. The targeted POIs hit by all private key parameter combinations in this group are `mbedtls_mpi_gcd` and `mbedtls_mpi_inv_mod`. Both are called by the parent function `mbedtls_rsa_deduce_private_exponent`, that aims at computing the private exponent if it is missing in the private key using (2), involving a modular inversion. However, for computing $\text{lcm}(p-1, q-1)$ using (4), the value $\text{gcd}(p-1, q-1)$ needs to be computed first. Therefore, the report indicates a call first to `mbedtls_mpi_gcd` with inputs $p-1$ and $q-1$. This call represents an SCA vulnerability as the binary GCD algorithm is vulnerable in these instances [4, 8, 10]. Note, this leakage is also present in OpenSSL (Section 3.3), however the contexts differ. We observed OpenSSL leakage when verifying d correctness, whereas mbedTLS computes d because it is missing. This difference is crucial regarding SCA, because OpenSSL verifies by checking if $de = 1 \bmod \text{lcm}(p-1, q-1)$ holds; yet mbedTLS indeed computes d , executing a modular inversion (2). Therefore this vulnerability is present in mbedTLS, and absent in OpenSSL.

After obtaining $\text{lcm}(p-1, q-1)$, it computes d using (2) through a call to `mbedtls_mpi_inv_mod`. [61, 76] exploit OpenSSL’s BEEA using microarchitecture attacks, so at a high level it represents a serious security threat. A deeper analysis follows for this mbedTLS case.

Summarizing, the private exponent computation in mbedTLS contains two vulnerable code paths: (i) GCD computation of $p-1$ and $q-1$; and (ii) modular inverse computation of e modulo $\text{lcm}(p-1, q-1)$. Next, we investigate which of these represents the most critical threat.

The inputs of the first code path (GCD computation) are roughly the same size. This characteristic implies that, for some SCA signals, the number of bits that can be recovered is small and not sufficient to break RSA. [7, 61] practically demonstrated this limitation using different SCA techniques: the former power consumption, the latter microarchitecture timings.

However, note the inputs of the second code path (modular inversion) differ considerably in size. The public exponent e is typically small, e.g. 65537. Following (4), $\text{lcm}(p-1, q-1)$ has roughly the same number of bits as $(p-1)(q-1)$;

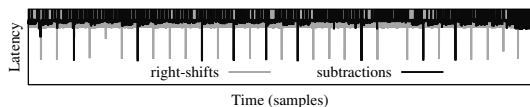


Figure 3: Sequence of right-shifts and subtractions from a FLUSH+RELOAD attack targeting mbedTLS modular inversion.

more than 1024 because $\gcd(p-1, q-1)$ is small with high probability [44]. This significant bit length difference between `mbedtls_mpi_inv_mod` inputs makes this algorithm extremely vulnerable to SCA [8]. This difference implies the attacker knows part of the algorithm execution flow beforehand, and it is exactly this part that is usually difficult to obtain and considerably limits the number of bits that can be recovered employing some SCA techniques as demonstrated in [7, 61]. This characteristic means the attacker only needs to distinguish the main two arithmetic operations present in this algorithm (i.e. right-shift and subtraction) to fully recover the input $\text{lcm}(p-1, q-1)$ that yields d .

Regarding microarchitecture attacks, this distinction lends itself to a FLUSH+RELOAD attack. As part of our validation, we attacked this implementation using a FLUSH+RELOAD attack paired with a performance degradation technique [11]. We probed two cache lines: one detecting right-shift executions, the other subtractions. Figure 3 shows the start of a trace, demonstrating the sequence extraction of right-shifts and subtraction is straightforward.

In addition, the key loading application threat model allows capturing several traces corresponding to the processing of the same secret data. Therefore, the attacker can correct errors that may appear in captured traces (e.g. fix errors produced by preemptions) by combining the information as they are redundant.

Recap. After the analysis of both leaking code paths we detected, we conclude the private exponent leakage is easier to exploit than that of CRT due to the large bitlength difference between the modular inversion algorithm inputs in the former [8, 10, 76]. On the other hand, the private exponent leakage is only present when the private key does not include d ; whereas the CRT-related leakage always represents a threat regardless of missing parameters [6]. The number of bits that can be recovered exploiting these leaking code paths depends on the side-channel signal employed. However, these code paths potentially leak all the bits of the processed secrets, as demonstrated in [6, 8, 10, 76].

Keys in the wild. As such, in the context of mbedTLS the simplest example of a vulnerable RSA key is the default key typically generated by libraries, including *all* parameters. We verified this default behavior on e.g. mbedTLS, OpenSSL, and BoringSSL. Hence such keys are ubiquitous in nature. For example, Let’s Encrypt’s `certbot` tool for automated

certificate renewal only supports RSA keys. We conclude that any application linking to mbedTLS for RSA functionality including key parsing is potentially vulnerable, including (but certainly not limited to) ACME-backed web servers relying on mbedTLS for TLS functionality.

4 Two End-to-End Attacks

As highlighted in Section 3, the format used to encode a private key can lead to the bypass of side-channel countermeasures in cryptographic libraries: these are *Certified Side Channels*. In this section we concretely instantiate the threat in Section 3.1 with two SCA attacks against ECDSA signature generation over the popular NIST P-256 curve against OpenSSL 1.1.1a: a remote timing attack and an EM attack.

Target application. For computing the ECDSA signatures from the protocol stack application layer we chose RFC 3161 [79] Time Stamp Protocol. The protocol ensures the means of establishing a time stamping service: a time stamp request message from a client and the corresponding time stamp response from the Trusted Timestamp Authority (TSA). In short, the TSA acts as a trusted third party that binds the Time Stamp Token (TST) to a valid client request message—one way hash of some information—and digitally signs it with the private key. Anyone with a valid TSA certificate can thus verify the existence of the information with the particular time stamp, ensuring timeliness and non-repudiation.

In principle, the client generates a time stamp request message containing the version information, OID of the one way hash algorithm, and a valid hash of the data. Optionally, the client may also send TSA policy OID to be used for creating the time stamp instead of TSA default policy, a random nonce for verifying the response time of the server, and additionally request the signing public key certificate in the TSA response message. The server timestamp response contains a status value and a TST with the OID for the content type and the content itself composed of DER-encoded TST information (TSTinfo). The TSTinfo field incorporates the version number info, the TSA policy used to generate the time stamp response, the message imprint (same as the hashed data in the client request), a unique serial number for the TST, and the UTC based TST generation time along with the accuracy in terms of the time granularity. Depending on the client request, the server response may additionally contain the signing certificate and the client provided nonce value. For further details on TSP, the reader may refer to RFC 3161 [79].

Our attack exploits point multiplication in the ECDSA signature generation during the TSA response phase to recover the long term private key of the server. As a protocol-level target, we compiled and deployed unmodified `uts-server`¹³ v0.2.0 without debug symbols, an open source TSA server linking against an unmodified debug build of OpenSSL 1.1.1a.

¹³<https://github.com/kakwa/uts-server>

We configured the server with a NIST P-256 X.509 digital certificate, using the private key containing explicit parameters with a zero cofactor, i.e. the preconditions for our Section 3.1 vulnerability. We used the OpenSSL time stamp utility `ts` to create time stamp requests with SHA256 as the hash function, along with a request for the server’s public key certificate for verification. We used the provided HTTP configuration for `uts-server`, hence the TSP messages between the (victim) server and our (attacker) client were transported via standard HTTP.

Target device. We selected a Linux-based PINE A64-LTS board with an Allwinner A64 Quad Core SoC based on Cortex-A53 which supports a 64-bit instruction set with a maximum clock frequency of 1.15 GHz. The board runs Ubuntu 16.04.1 LTS without any modifications to the stock image. We set the board’s frequency governor to “performance”.

Threat model. As discussed (Section 3.1), when handling such a key in OpenSSL 1.1.1a, the underlying implementation for the EC scalar multiplication is based on a `wNAF` algorithm, which has been repeatedly targeted in SCA works over the last decade, usually focusing on the recovery of the LSBs of the secret scalar. Contributions from Google [46] partially mitigated the attack vector for select named curves with new `EC_METHOD` implementations, then fully even for generic curves due to the results and contributions from [72]. With the attack vector now open again, this section presents two end-to-end attacks with different signal procurement methods: (i) a novel remote timing attack (Section 4.1), where it is assumed the attacker can measure the overall wall clock time it takes for the TSA server to respond to a request—note this attacker is indistinguishable from a legitimate user of the service; (ii) an EM attack (Section 4.2), similar in spirit to [38, 72], which has the same aforementioned threat model but additionally assumes physical proximity to non-invasively measure EM emanations. The motivation for the two different threat models is due to both practicality and the number of required samples, which will become evident by the end of this section.

4.1 ECDSA: Remote Timing Attack

In contrast to previous work on this code path and to widen potential real-world application, we performed a remote timing attack on the TSA server application via TCP. Instead of taking measurements on this code path server side like e.g. `PRIME+PROBE` [20, 21] and `FLUSH+RELOAD` [11, 14, 30, 73], we (as a non-privileged, normal user of the service), make network requests and measure the wall clock response time.

Experiment setup. We connected the PINE A64-LTS board directly by Ethernet cable to a workstation equipped with an Intel i5-4570 CPU and an onboard I217-LM (rev 04) Ethernet controller. To measure the remote wall clock latency

and reduce noise, we created a custom HTTP client for time stamp requests. Its algorithm is as follows: (i) establish a TCP connection to the server; (ii) write the HTTP request and the body, sans a single byte; (iii) start the timer; (iv) write the last body byte—now the server can begin computing the digital signature; (v) read the HTTP response headers—the server might write at least part of them before computing the digital signature; (vi) read one byte of HTTP response body—the digital signature is received by the server directly from linked OpenSSL in an octet string, so reading one byte guarantees it has been generated; (vii) stop the timer; (viii) finish reading the HTTP response; (ix) record the timing information and digital signature in a database; (x) close the TCP connection; (xi) repeat until the requested number of samples has been gathered. We implemented the measurement software in C to achieve maximum performance and control over operations. For the client timer, we used the `x86 rtdtsc` instruction that is freely accessible from user space. In recent Intel processors the `constant_tsc` feature is available—a frequency-independent and easily accessible precision timer.

Performing a traditional timing analysis under the above assumptions, we discovered a direct correlation between the wall clock execution time of ECDSA signature generation and the bitlength of the nonce used to compute the signature, as shown in Figure 4. This happens because given a scalar k and its recoded NAF representation \hat{k} , the algorithm execution time is a function of both the NAF length of \hat{k} and its Hamming weight. While the NAF length is a good approximation for the bitlength of k (in fact at most one digit longer), its Hamming weight masks the NAF length linearly so it is not obvious how to correlate these two factors with the precise bitlength of k . Nevertheless, the empirical results (by sampling) shown in Figure 4 clearly demonstrate the latter is directly proportional to the overall algorithm execution time.

This result shares similarity to the one exploited in CVE-2011-1945 [22] (that built the foundation for the recent Minerva¹⁴ and TPM-FAIL [54] attacks), and in fact suggests that CVE applied to not only binary curves using the Montgomery ladder, but prime curves as well. Following their attack methodology, we devise an attack in two phases: (i) The collection phase exploits the timing dependency between the execution time and the bitlength of the nonce used to generate a signature, thus selecting (r, s, m) tuples associated with shorter-than-average nonces; (ii) The recovery phase then combines the partial knowledge inferred from the collection phase to instantiate an HNP instance and solve it through a lattice technique (Section 2.4).

Collection phase. Using our custom TCP time stamp client, across Ethernet we collect 500K traces for a single attack, sorting by the measured latency, and filter the first $t = 128$ items: empirically this is closely related to the selection by a fixed threshold suggested by Figure 4. We prefer the for-

¹⁴<https://minerva.crocs.fi.muni.cz/>

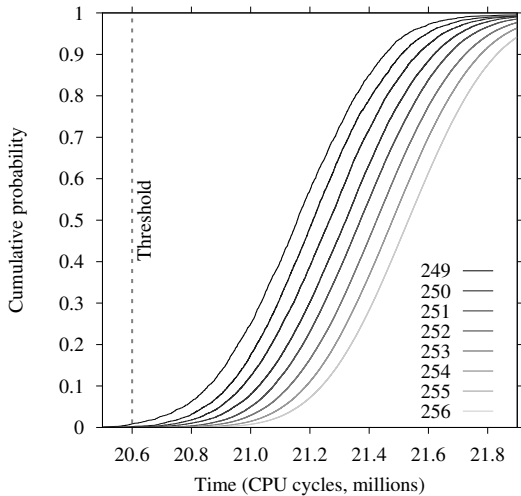


Figure 4: Direct correlation between wall-clock execution time of ECDSA signature generation and the bitlength of the nonce. Plots from left to right correspond to legend keys from top to bottom. Measured on NIST P-256 in OpenSSL on a Pine64-LTS, bypassing all SCA hardening countermeasures via a private key parsing trigger.

mulation where we set the dimension t of the filtered set and the total number of collected signatures, as these numbers are more significant for comparison with other works or directly used in the formalization of the subsequent lattice phase.

Lattice phase. As noted above, the collection phase in this attack selects shorter-than-average nonces, i.e. looking at the nonce k_i as a string of bits with the same bitlength of the generator order n ,

$$0 < k_i < 2^{(\lg(n)-\ell_i)} < 2^{(\lg(n)-\ell)} < n/2^\ell \equiv n/W < n$$

for some $W = 2^\ell$ bound, representing that at least ℓ consecutive MSBs are equal to 0. This is in contrast with the Section 2.4 formalization, which instead implies knowledge of nonce LSBs, so we need to slightly revise some definitions to frame the lattice problem using the same notation. Therefore, we can define $W_i = W = 2^\ell$ and, similarly to the formalization in Section 2.4, rearrange (1) as $k_i = \alpha(r_i/s_i) - (-h_i/s_i) \bmod n$ and then redefine $t_i = \lfloor r_i/s_i \rfloor_n$, $\hat{u}_i = \lfloor -h_i/s_i \rfloor_n$ which leads once again to $0 \leq \lfloor \alpha t_i - \hat{u}_i \rfloor_n < n/W_i$, from which the rest of the previous formalization follows unchanged.

Although it used a different lattice description, [22] also dealt with a leak based on nonce MSBs, which led to an interesting property that is valid also for the formalization used in this particular lattice attack. Comparing the definitions of t_i , \hat{u}_i , and u_i above with the ones from Section 2.4, we note

that in this particular attack no analogue of the a_i term features in the equations composing the lattice problem, from which follows that even if some k_i does not strictly satisfy the bound $k_i < n/W$ there is still a chance that the attack will succeed, leading to a better resilience to errors (i.e., entries in the lattice that do not strictly satisfy the bound above) in this lattice formulation. From the attacker perspective, higher W is desirable but requires more leakage from the victim.

Since in this formulation the attacker does not use a per-equation W_i as the distributions are partially overlapping, the question remains how to set W . Underestimating W is technically accurate for approximating zero-MSBs for most of the filtered traces, but forces higher lattice dimensions and slower computation for each job. Using a larger set of training samples, analyzing the ground truth w.r.t. the actual nonce of each sample, we empirically determined that the distribution of nonce bitlengths on the average set filtered by our collection phase is a Gaussian distribution with mean $\lg(k_i) = 247.80$ and s.d. 3.81, which suggests $W = 2^8$ ($8 = 256 - 248$) is a better approximation of the bound on most nonces. Given the s.d. magnitude, by trial-and-error we set $W = 2^7$ as a good trade-off for lattice attack execution time vs. success rate.

Combining the better resilience to errors of this particular lattice formulation and the higher amount of information carried by each trace included in the lattice instance by pushing W , we fixed the lattice attack parameters to $d = 60$ and $j = 55K$ and limit the maximum number of attempted lattice reductions per job to 100 (in practice on our cluster, less than a single minute), as we observed the overwhelming majority of instances returned success within this time frame or not at all.

Attack results. With these parameters, and repeating the attack 100 times, we observed a 91% success rate in our remote timing attack over Ethernet. The median number of jobs needed over all attack instances was 1377 (i.e. $j = 1377$ was sufficient for key recovery in the majority of cases). Those reductions that led to successful key recovery (i.e. 91 in number) had $\lg(k_i) = 246.85$ and s.d. 3.13, while the $j = 55K$ reductions per each of the 9 failed overall attack instances had $\lg(k_i) = 247.96$ and s.d. 3.87. This difference suggests: (i) the better resilience to errors in this lattice formulation is empirically valid, as given the stated s.d. not all k_i satisfied the bound W ; (ii) in our environment, even the failed instances would likely succeed by tweaking lattice parameters (i.e. decreasing W and increasing d) and providing more parallel computation power (i.e. increasing j).

In case of success, the attacker obtains the long-term secret key. On failure she can repeat the collection phase (accumulating more traces and improving the filtering output and the probability of success of another lattice phase) or iteratively tune the lattice parameters (decreasing W and increasing d) to adapt to the features of the specific output of the collection phase, thus improving the lattice attack's success probability.

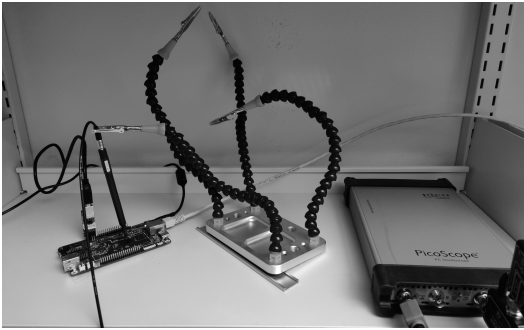


Figure 5: Experiment setup capturing EM traces using Picoscope USB oscilloscope with the Langer EM probe positioned on the Pine64-LTS SoC: a TSP server connected via Ethernet serving requests over HTTP.

4.2 ECDSA: EM Attack

In a much stronger (yet still SCA-classical) attack model assuming physical proximity, we now perform an EM attack on OpenSSL ECDSA. As far as we are aware, we are the first to exploit this code path in the context of OpenSSL and NIST P-256: [38] target the 256-bit Bitcoin curve, and [72] the 256-bit SM2 curve. The reason for this is our Section 3.1 vulnerability allows us to bypass the dedicated `EC_METHOD` instance on this architecture, `EC_GFp_nistz256_method` which is constant time and optimized for AVX and ARMv8 architectures. The wNAF Double and Add operations have a different set of underlying finite field operations—square, multiply, add, sub, inversion—resulting in distinguishable EM signatures.

Experiment setup. To capture the EM traces, we positioned the Langer LF-U 2.5 near field probe head on the SoC where it resulted in the highest signal quality. For digitizing the EM emanations, we used Picoscope 6404C USB digital oscilloscope with a bandwidth of 500 MHz and maximum sampling rate of 5 GSps. However, we used a lower sampling rate of 125 MSps as the best compromise between the trace quality and processing overhead. To acquire the traces while ensuring that the entire ECDSA trace was captured, we synchronized the oscilloscope capture with the time stamp request message: initiate the oscilloscope to start acquiring traces, query a time stamp request over HTTP to the server and wait for the server response, and finally stop the trace acquisition. We stored the EM traces along with the DER-encoded server response messages. We parsed the messages to retrieve the hash from the client request and the DER-encoded ECDSA signatures, used to generate metadata for the key recovery phase. Figure 5 shows the setup we used for our attack.

Signal analysis. After capturing the traces, we moved to offline post processing of the EM traces for recovering the partial nonce information. This essentially means identifying

the position of the last Add operation. The problem is twofold: finding the end of the point multiplication (end trigger), then identifying the last Add operation therein. We divided the complete signal processing phase mainly into four steps: (i) Remove traces with errors due to acquisition process; (ii) Find the end of the ECDSA point multiplication; (iii) Remove traces encountering interrupts; (iv) Identifying the position of the last Add operation. We started by selecting only those traces which had peak magnitude to the root mean square ratio within an emphatically selected confidence interval, evidently removing traces where the point multiplication operation was not captured or trace was too noisy to start with.

In the next step, we used a specific pattern at the end of ECDSA point multiplication as our soft end trigger. To isolate this trigger pattern from the rest of the signal, we first applied a low pass FIR filter followed by a phase demodulation using the digital Hilbert transform. We further enhanced this pattern while suppressing the rest of the operations by applying root mean square envelope with a window size roughly half its sample size. We created a template by extracting this pattern from 20 random traces and taking their average. We used the Euclidean distance between the trace and template to find the end of point multiplication. We dropped all traces where the Euclidean distance was above an experimental threshold value, i.e. no soft trigger found. The traces also encountered random interrupts due to OS scheduling clearly identifiable as high amplitude peaks. Any traces with an interrupt at the end of point multiplication were also discarded to avoid corrupting the detection of the Add operation.

To recover the position of the last Add operation, we applied a different set of filters on the raw trace, keeping the end of point multiplication as our starting reference. Since the frequency analysis revealed most of the Add operations energy is between 40 MHz and 50 MHz, we applied a band pass FIR filter around this band. Performing a digital Hilbert transform, additional signal smoothing and peak envelope detection, the Add operations were clearly identifiable (Figure 6).

To automatically extract the Add operation, we first used peak extraction. However it was not as reliable since the signals occasionally encountered noisy peaks or in some instances the Add peaks were distorted. We again resorted to the template matching method used in the previous step, i.e. create an Add template and use Euclidean distance for pattern matching. For each peak identified, we also applied the template matching and measured the resulting Euclidean distance against a threshold value. Anything greater than the threshold was considered a false positive peak.

These steps ensured that the error rate stays low, consequently increasing the success rate of the key recovery lattice attack. We estimated the number of Double operations using the total sample length from the middle of the last Add operation to the end of trace as illustrated in Figure 6. To effectively reduce the overlap between the sample length metric of different Double and Add sequences, we applied K-means

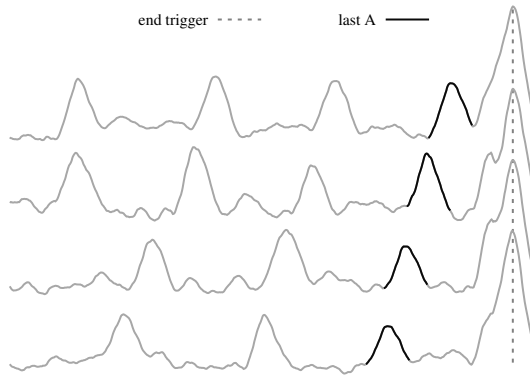


Figure 6: Four different EM traces showing the last Add (A) operations relative to the soft end trigger. The distance in terms of samples between the last Add and trigger gives the number of Double (D) operations. Top to Bottom: Trace ends with an A, AD, ADD, ADDD.

clustering to keep sequences which were close to the cluster mean.

Attack results. We acquired a total of 500 signatures, and after performing the signal processing steps we were left with 422 traces. Additionally, after filtering out signatures categorized as “A” and hence not useful lattice-wise, we were left with $t = 172$ signatures suitable for building lattice problem instances. We chose $d = 120$ as the number of signatures to populate the lattice basis. We then constructed $j = 48$ instances of the lattice attack, randomly selecting d -size subsets from the t signatures for each instance. We then ran these instances in parallel on a 2.10 GHz dual CPU Intel Xeon Silver 4116 (24 cores, 48 threads across 2 CPUs). The first instance to succeed in recovering the private key did so in just over three minutes. Checking the ground truth afterwards, $e = 4$ out of the t signatures were categorized incorrectly, for a suitably small error rate of about 2.3%.

5 Conclusion

In this work, we evaluated how different choices of private key formats and various optional parameters supported by them can influence SCA security. We employed the automated tool Triggerflow to analyze vulnerable code paths in well known cryptographic libraries for various combinations of key formats and optional parameters. The results uncovered several *Certified Side Channels*, circumventing SCA hardened code paths in OpenSSL (ECC with explicit parameters, DSA with MSBLOB and PVK formats, RSA during key validation) and mbedTLS (RSA with missing parameters). To demonstrate the severity of these vulnerabilities, we performed microarchitecture leakage analysis on RSA and DSA and also presented

end-to-end key recovery attacks on OpenSSL ECDSA using traditional timing and EM side channels. We publish our data set for the remote timing attack to support Open Science [63].

In the OpenSSL case, Pereira García et al. [62] conclude the fundamental design issue around `BN_FLG_CONSTTIME` is due to its insecure default nature, hypothesizing inverted logic with secure-by-default behavior provides superior assurances. While that would indeed have prevented CVE-2016-2178, our work shows that runtime secure-by-default is still not enough: simply the presence of known SCA-vulnerable code alongside SCA-hardened code poses a security threat. For example, in this light, in our Section 3.1 vulnerability the zero cofactor masquerades as a virtual `BN_FLG_CONSTTIME`, since the exploited code path is oblivious to the flag’s value by design.

Mitigations. As part of the responsible disclosure process, we notified OpenSSL and mbedTLS of our findings. At the same time, we made several FOSS contributions to help mitigate these issues in OpenSSL, who assigned CVE-2019-1547 based on our work. For the Section 3.1 vulnerability, we implemented a fix that manually computes the cofactor from the field cardinality and generator order using the Hasse Bound. This works for all standards-compliant curves—named or with explicit parameters. To mitigate the vulnerabilities in Section 3.2 and Section 3.3, we submitted simple patches that set the `BN_FLG_CONSTTIME` correctly, steering the computations to existing SCA-hardened code. Moreover, we replaced the variable-time GCD function in OpenSSL by a constant-time implementation¹⁵ based on the work by Bernstein and Yang [16]. To further reduce the SCA attack surface, we implemented changes¹⁶ in the way OpenSSL creates EC key abstractions when the associated curve is defined by explicit parameters. The explicit parameters are matched against the internal table of known curves, switching to an internal named curve representation for matches, ultimately enabling the use of specialized implementations where available. Finally, we integrated the new Triggerflow unit tests (Figure 1). Applying all these fixes across non-EOL OpenSSL branches as well as the development branch, no Triggerflow POIs are subsequently triggered, indicated the patches are effective.

Astute readers may notice the above fixes do not remove the vulnerable functions in question. In general, indeed this is one option, but such a strategy requires analysis on a case-by-case basis. These are real-world products that come with real-world performance constraints. For example, an SCA-secure modular exponentiation function that protects both the length and values of the exponent can likely meet the performance requirements for e.g. DSA signing, but not RSA verification with a short, low-weight, and public exponent. This is the main reason why libraries often feature multiple versions of the same functionality with different security characteristics.

¹⁵<https://github.com/openssl/openssl/pull/10122>

¹⁶<https://github.com/openssl/openssl/pull/9808>

Future work. In Section 4.1, discussing the lattice formulation for our attack, we highlight an increased resilience to lattice errors compared to similar previous works. We note here that an analysis of error resilience of different lattice constructions and of strategies to increase the overall success rate of lattice attacks in the presence of errors in collected traces would constitute a valuable future contribution to this area of research.

Our vulnerabilities in Section 3 cover only a very small subset of possible inputs, combinations, architectures, and settings. Another interesting research question is how to extend test coverage in a reasonable way, even considering other libraries.

Acknowledgments. We thank Tampere Center for Scientific Computing (TCSC) for generously granting us access to computing cluster resources. The second author was supported in part by the Tuula and Yrjö Neuvo Fund through the Industrial Research Fund at Tampere University of Technology. The third author was supported in part by a Nokia Scholarship from the Nokia Foundation. This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No 804476).

References

- [1] Security requirements for cryptographic modules. FIPS PUB 140-2, National Institute of Standards and Technology, May 2001. URL <https://doi.org/10.6028/NIST.FIPS.140-2>.
- [2] SEC 1: Elliptic Curve Cryptography. SEC 1, Standards for Efficient Cryptography Group, May 2009. URL <http://www.secg.org/sec1-v2.pdf>.
- [3] Rodrigo Abarzúa, Claudio Valencia Cordero, and Julio López Hernandez. Survey for performance & security problems of passive side-channel attacks countermeasures in ECC. *IACR Cryptology ePrint Archive*, 2019(10), 2019. URL <https://eprint.iacr.org/2019/010>.
- [4] Onur Acıçmez, Shay Gueron, and Jean-Pierre Seifert. New branch prediction vulnerabilities in OpenSSL and necessary software countermeasures. In *IMACC*, volume 4887 of *LNCS*, pages 185–203. Springer, 2007. URL https://doi.org/10.1007/978-3-540-77272-9_12.
- [5] Dakshi Agrawal, Bruce Archambeault, Josyula R. Rao, and Pankaj Rohatgi. The EM side-channel(s). In *CHES*, volume 2523 of *LNCS*, pages 29–45. Springer, 2002. URL https://doi.org/10.1007/3-540-36400-5_4.
- [6] Alejandro Cabrera Aldaya and Billy Bob Brumley. When one vulnerable primitive turns viral: Novel single-trace attacks on ECDSA and RSA. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(2):196–221, 2020. URL <https://doi.org/10.13154/tches.v2020.i2.196-221>.
- [7] Alejandro Cabrera Aldaya, Alejandro J. Cabrera Sarmiento, and Santiago Sánchez-Solano. SPA vulnerabilities of the binary extended Euclidean algorithm. *J. Cryptographic Engineering*, 7(4):273–285, 2017. URL <https://doi.org/10.1007/s13389-016-0135-4>.
- [8] Alejandro Cabrera Aldaya, Raudel Cuiman Márquez, Alejandro J. Cabrera Sarmiento, and Santiago Sánchez-Solano. Side-channel analysis of the modular inversion step in the RSA key generation algorithm. *I. J. Circuit Theory and Applications*, 45(2):199–213, 2017. URL <https://doi.org/10.1002/cta.2283>.
- [9] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. In *IEEE S&P*, pages 870–887. IEEE, 2019. URL <https://doi.org/10.1109/SP.2019.00066>.
- [10] Alejandro Cabrera Aldaya, Cesar Pereida García, Luis Manuel Alvarez Tapia, and Billy Bob Brumley. Cache-timing attacks on RSA key generation. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(4):213–242, 2019. URL <https://doi.org/10.13154/tches.v2019.i4.213-242>.
- [11] Thomas Allan, Billy Bob Brumley, Katrina E. Falkner, Joop van de Pol, and Yuval Yarom. Amplifying side channels through performance degradation. In *ACSAC*, pages 422–435. ACM, 2016. URL <http://doi.acm.org/10.1145/2991079.2991084>.
- [12] Aurélie Bauer, Éliane Jaulmes, Emmanuel Prouff, Jean-René Reinhard, and Justine Wild. Horizontal collision correlation attack on elliptic curves – extended version. *Cryptography and Communications*, 7(1):91–119, 2015. URL <https://doi.org/10.1007/s12095-014-0111-8>.
- [13] Pierre Belgarric, Pierre-Alain Fouque, Gilles Macario-Rat, and Mehdi Tibouchi. Side-channel analysis of Weierstrass and Koblitz curve ECDSA on Android smartphones. In *CT-RSA*, volume 9610 of *LNCS*, pages 236–252. Springer, 2016. URL https://doi.org/10.1007/978-3-319-29485-8_14.
- [14] Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. “Ooh Aah... Just a Little Bit”: A small amount of side channel can go a long way. In *CHES*, volume 8731 of *LNCS*, pages 75–92. Springer, 2014. URL https://doi.org/10.1007/978-3-662-44709-3_5.
- [15] Daniel J. Bernstein. Cache-timing attacks on AES, 2005. URL <http://cr.yyp.to/papers.html#cachetiming>.
- [16] Daniel J. Bernstein and Bo-Yin Yang. Fast constant-time gcd computation and modular inversion. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(3):340–398, 2019. URL <https://doi.org/10.13154/tches.v2019.i3.340-398>.
- [17] Daniel J. Bernstein, Joachim Breitner, Daniel Genkin, Leon Groot Bruinderink, Nadia Heninger, Tanja Lange, Christine van Vredendaal, and Yuval Yarom. Sliding right into disaster: Left-to-right sliding windows leak. In *CHES*, volume 10529 of *LNCS*, pages 555–576. Springer, 2017. URL https://doi.org/10.1007/978-3-319-66787-4_27.
- [18] Dan Boneh and Ramarathnam Venkatesan. Hardness of computing the most significant bits of secret keys in Diffie-Hellman and related schemes. In *CRYPTO*, volume 1109 of *LNCS*, pages

- 129–142. Springer, 1996. URL https://doi.org/10.1007/3-540-68697-5_11.
- [19] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In *CHES*, volume 3156 of *LNCS*, pages 16–29. Springer, 2004. URL https://doi.org/10.1007/978-3-540-28632-5_2.
- [20] Billy Bob Brumley. Faster software for fast endomorphisms. In *COSADE*, volume 9064 of *LNCS*, pages 127–140. Springer, 2015. URL https://doi.org/10.1007/978-3-319-21476-4_9.
- [21] Billy Bob Brumley and Risto M. Hakala. Cache-timing template attacks. In *ASIACRYPT*, volume 5912 of *LNCS*, pages 667–684. Springer, 2009. URL https://doi.org/10.1007/978-3-642-10366-7_39.
- [22] Billy Bob Brumley and Nicola Taveri. Remote timing attacks are still practical. In *ESORICS*, volume 6879 of *LNCS*, pages 355–371. Springer, 2011. URL https://doi.org/10.1007/978-3-642-23822-2_20.
- [23] David Brumley and Dan Boneh. Remote timing attacks are practical. In *USENIX Sec.* USENIX Association, 2003. URL <https://www.usenix.org/conference/12th-usenix-security-symposium/remote-timing-attacks-are-practical>.
- [24] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on meltdown-resistant CPUs. In *ACM CCS*, pages 769–784. ACM, 2019. URL <https://doi.org/10.1145/3319535.3363219>.
- [25] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template attacks. In *CHES*, volume 2523 of *LNCS*, pages 13–28. Springer, 2002. URL https://doi.org/10.1007/3-540-36400-5_3.
- [26] Christophe Clavier, Benoit Feix, Georges Gagnerot, Mylène Roussellet, and Vincent Verneuil. Horizontal correlation analysis on exponentiation. In *ICICS*, volume 6476 of *LNCS*, pages 46–61. Springer, 2010. URL https://doi.org/10.1007/978-3-642-17650-0_5.
- [27] Jean-Luc Danger, Sylvain Guilley, Philippe Hoogvorst, Cédric Murdica, and David Naccache. A synthesis of side-channel attacks on elliptic curve cryptography in smart-cards. *J. Cryptographic Engineering*, 3(4):241–265, 2013. URL <https://doi.org/10.1007/s13389-013-0062-6>.
- [28] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. CacheAudit: A tool for the static analysis of cache side channels. *ACM Trans. Inf. Syst. Secur.*, 18(1):4:1–4:32, 2015. URL <https://doi.org/10.1145/2756550>.
- [29] Margaux Dugardin, Louiza Papachristodoulou, Zakaria Najm, Lejla Batina, Jean-Luc Danger, and Sylvain Guilley. Dismantling real-world ECC with horizontal and vertical template attacks. In *COSADE*, volume 9689 of *LNCS*, pages 88–108. Springer, 2016. URL https://doi.org/10.1007/978-3-319-43283-0_6.
- [30] Shuqin Fan, Wenbo Wang, and Qingfeng Cheng. Attacking OpenSSL implementation of ECDSA with a few signatures. In *ACM CCS*, pages 1505–1515. ACM, 2016. URL <https://doi.org/10.1145/2976749.2978400>.
- [31] Jeffrey Friedman. Tempest: A signal problem. *NSA Cryptologic Spectrum*, 2(3):26–30, 1972. URL <https://www.nsa.gov/Portals/70/documents/news-features/decclassified-documents/cryptologic-spectrum/tempest.pdf>.
- [32] Nicolas Gama, Phong Q. Nguyen, and Oded Regev. Lattice enumeration using extreme pruning. In *EUROCRYPT*, volume 6110 of *LNCS*, pages 257–278. Springer, 2010. URL https://doi.org/10.1007/978-3-642-13190-5_13.
- [33] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *J. Cryptographic Engineering*, 8(1):1–27, 2018. URL <https://doi.org/10.1007/s13389-016-0141-6>.
- [34] Daniel Genkin, Adi Shamir, and Eran Tromer. RSA key extraction via low-bandwidth acoustic cryptanalysis. In *CRYPTO*, volume 8616 of *LNCS*, pages 444–461. Springer, 2014. URL https://doi.org/10.1007/978-3-662-44371-2_25.
- [35] Daniel Genkin, Lev Pachmanov, Itamar Pipman, and Eran Tromer. Stealing keys from PCs using a radio: Cheap electromagnetic attacks on windowed exponentiation. In *CHES*, volume 9293 of *LNCS*, pages 207–228. Springer, 2015. URL https://doi.org/10.1007/978-3-662-48324-4_11.
- [36] Daniel Genkin, Itamar Pipman, and Eran Tromer. Get your hands off my laptop: physical side-channel key-extraction attacks on PCs - extended version. *J. Cryptographic Engineering*, 5(2):95–112, 2015. URL <https://doi.org/10.1007/s13389-015-0100-7>.
- [37] Daniel Genkin, Lev Pachmanov, Itamar Pipman, and Eran Tromer. ECDH key-extraction via low-bandwidth electromagnetic attacks on PCs. In *CT-RSA*, volume 9610 of *LNCS*, pages 219–235. Springer, 2016. URL https://doi.org/10.1007/978-3-319-29485-8_13.
- [38] Daniel Genkin, Lev Pachmanov, Itamar Pipman, Eran Tromer, and Yuval Yarom. ECDSA key extraction from mobile devices via nonintrusive physical side channels. In *ACM CCS*, pages 1626–1638. ACM, 2016. URL <http://doi.acm.org/10.1145/2976749.2978353>.
- [39] Oded Goldreich, Shafi Goldwasser, and Shai Halevi. Public-key cryptosystems from lattice reduction problems. In *CRYPTO*, volume 1294 of *LNCS*, pages 112–131. Springer, 1997. URL <https://doi.org/10.1007/BFb0052231>.
- [40] Iaroslav Gridin, Cesar Pereida García, Nicola Taveri, and Billy Bob Brumley. Triggerflow: Regression testing by advanced execution path inspection. In *DIMVA*, volume 11543 of *LNCS*, pages 330–350. Springer, 2019. URL https://doi.org/10.1007/978-3-030-22038-9_16.
- [41] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Sec.*, pages

- 897–912. USENIX Association, 2015. URL <https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/gruss>.
- [42] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+flush: A fast and stealthy cache attack. In *DIMVA*, volume 9721 of *LNCS*, pages 279–299. Springer, 2016. URL https://doi.org/10.1007/978-3-319-40667-1_14.
- [43] Shay Gueron and Vlad Krasnov. Fast prime field elliptic-curve cryptography with 256-bit primes. *J. Cryptographic Engineering*, 5(2):141–151, 2015. URL <https://doi.org/10.1007/s13389-014-0090-x>.
- [44] M. Jason Hinek. *Cryptanalysis of RSA and its variants*. Chapman & Hall/CRC Cryptography and Network Security. CRC Press, 2010. ISBN 978-1-4200-7518-2. URL <https://doi.org/10.1201/9781420075199>.
- [45] Simon Josefsson and Sean Leonard. Textual encodings of PKIX, PKCS, and CMS structures. RFC 7468, RFC Editor, April 2015. URL <https://datatracker.ietf.org/doc/rfc7468/>.
- [46] Emilia Käsper. Fast elliptic curve cryptography in OpenSSL. In *Financial Cryptography Workshops*, volume 7126 of *LNCS*, pages 27–39. Springer, 2011. URL https://doi.org/10.1007/978-3-642-29889-9_4.
- [47] Vlastimil Klíma, Ondrej Pokorný, and Tomás Rosa. Attacking RSA-based sessions in SSL/TLS. In *CHES*, volume 2779 of *LNCS*, pages 426–440. Springer, 2003. URL https://doi.org/10.1007/978-3-540-45238-6_33.
- [48] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *CRYPTO*, volume 1109 of *LNCS*, pages 104–113. Springer, 1996. URL https://doi.org/10.1007/3-540-68697-5_9.
- [49] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *CRYPTO*, volume 1666 of *LNCS*, pages 388–397. Springer, 1999. URL https://doi.org/10.1007/3-540-48405-1_25.
- [50] Jake Longo, Elke De Mulder, Dan Page, and Michael Tunstall. SoC it to EM: ElectroMagnetic side-channel attacks on a complex System-on-Chip. In *CHES*, volume 9293 of *LNCS*, pages 620–640. Springer, 2015. URL https://doi.org/10.1007/978-3-662-48324-4_31.
- [51] Accredited Standards Committee X9, editor. *ANSI X9.62-2005: Public Key Cryptography For The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)*. ANSI American National Standards Institute, 2005.
- [52] Thomas S. Messerges, Ezzy A. Dabbish, and Robert H. Sloan. Examining smart-card security under the threat of power analysis attacks. *IEEE Trans. Computers*, 51(5):541–552, 2002. URL <https://doi.org/10.1109/TC.2002.1004593>.
- [53] Christopher Meyer, Juraj Somorovsky, Eugen Weiss, Jörg Schwenk, Sebastian Schinzel, and Erik Tews. Revisiting SSL/TLS implementations: New Bleichenbacher side channels and attacks. In *USENIX Sec.*, pages 733–748. USENIX Association, 2014. URL <https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/meyer>.
- [54] Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, and Nadia Heninger. TPM-FAIL: TPM meets timing and lattice attacks. In *USENIX Sec.* USENIX Association, 2020. URL <https://www.usenix.org/conference/usenixsecurity20/presentation/moghimi>.
- [55] Kathleen Moriarty, Burt Kaliski, Jakob Jonsson, and Andreas Rusch. PKCS #1: RSA cryptography specifications version 2.2. RFC 8017, RFC Editor, November 2016. URL <https://datatracker.ietf.org/doc/rfc8017/>.
- [56] Samuel Neves and Mehdi Tibouchi. Degenerate curve attacks: extending invalid curve attacks to Edwards curves and other models. *IET Information Security*, 12(3):217–225, 2018. URL <https://doi.org/10.1049/iet-ifs.2017.0075>.
- [57] Phong Q. Nguyen and Igor E. Shparlinski. The insecurity of the Digital Signature Algorithm with partially known nonces. *J. Cryptology*, 15(3):151–176, 2002. URL <https://doi.org/10.1007/s00145-002-0021-3>.
- [58] Phong Q. Nguyen and Igor E. Shparlinski. The insecurity of the Elliptic Curve Digital Signature Algorithm with partially known nonces. *Des. Codes Cryptogr.*, 30(2):201–217, 2003. URL <https://doi.org/10.1023/A:1025436905711>.
- [59] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *CT-RSA*, volume 3860 of *LNCS*, pages 1–20. Springer, 2006. URL https://doi.org/10.1007/11605805_1.
- [60] Colin Percival. Cache missing for fun and profit. In *BSD-Can*, 2005. URL <http://www.daemonology.net/papers/cachemissing.pdf>.
- [61] Cesar Pereida García and Billy Bob Brumley. Constant-time callees with variable-time callers. In *USENIX Sec.*, pages 83–98. USENIX Association, 2017. ISBN 978-1-931971-40-9. URL <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/garcia>.
- [62] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. “Make sure DSA signing exponentiations really are constant-time”. In *ACM CCS*, pages 1639–1650. ACM, 2016. URL <http://doi.acm.org/10.1145/2976749.2978420>.
- [63] Cesar Pereida García, Sohaib ul Hassan, Nicola Tuveri, Iaroslav Gridin, Alejandro Cabrera Aldaya, and Billy Bob Brumley. *CVE-2019-1547: research data and tooling*. Zenodo, April 2020. URL <https://doi.org/10.5281/zenodo.3736311>.
- [64] Vladimir Popov, Serguei Leontiev, and Igor Kurepkin. Additional cryptographic algorithms for use with GOST 28147-89, GOST R 34.10-94, GOST R 34.10-2001, and GOST R 34.11-94 algorithms. RFC 4357, RFC Editor, January 2006. URL <https://datatracker.ietf.org/doc/rfc4357/>.
- [65] Thomas Popp, Stefan Mangard, and Elisabeth Oswald. Power analysis attacks and countermeasures. *IEEE Design & Test of Computers*, 24(6):535–543, 2007. URL <https://doi.org/10.1109/MDT.2007.200>.

[66] Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (EMA): measures and counter-measures for smart cards. In *E-smart*, volume 2140 of *LNCSS*, pages 200–210. Springer, 2001. URL https://doi.org/10.1007/3-540-45418-7_17.

[67] Michael O. Rabin. Probabilistic algorithm for testing primality. *J. Number Theory*, 12(1):128–138, 1980. ISSN 0022-314X. URL [https://doi.org/10.1016/0022-314x\(80\)90084-0](https://doi.org/10.1016/0022-314x(80)90084-0).

[68] Niels Samwel, Lejla Batina, Guido Bertoni, Joan Daemen, and Ruggero Susella. Breaking Ed25519 in WolfSSL. In *CT-RSA*, volume 10808 of *LNCSS*, pages 1–20. Springer, 2018. URL https://doi.org/10.1007/978-3-319-76953-0_1.

[69] ITU-T Telecommunication standardization sector of ITU, editor. *ITU-T X.690 Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)*. ITU International Telecommunication Union, August 2015. URL <http://handle.itu.int/11.1002/1000/12483>.

[70] Akira Takahashi and Mehdi Tibouchi. Degenerate fault attacks on elliptic curve parameters in OpenSSL. In *EuroS&P*, pages 371–386. IEEE, 2019. URL <https://doi.org/10.1109/EuroSP.2019.00035>.

[71] Michael Tunstall. Smart card security. In *Smart Cards, Tokens, Security and Applications*, pages 217–251. Springer, second edition, 2017. URL https://doi.org/10.1007/978-3-319-50500-8_9.

[72] Nicola Tuveri, Sohaib ul Hassan, Cesar Pereida García, and Billy Bob Brumley. Side-channel analysis of SM2: A late-stage featurization case study. In *ACSAC*, pages 147–160. ACM, 2018. URL <https://doi.org/10.1145/3274694.3274725>.

[73] Joop van de Pol, Nigel P. Smart, and Yuval Yarom. Just a little bit more. In *CT-RSA*, volume 9048 of *LNCSS*, pages 3–21. Springer, 2015. URL https://doi.org/10.1007/978-3-319-16715-2_1.

[74] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: rogue in-flight data load. In *IEEE S&P*, pages 88–105. IEEE, 2019. URL <https://doi.org/10.1109/SP.2019.00087>.

[75] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. CacheD: Identifying cache-based timing channels in production software. In *USENIX Sec.*, pages 235–252. USENIX Association, 2017. URL <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/wang-shuai>.

[76] Samuel Weiser, Raphael Spreitzer, and Lukas Bodner. Single trace attack against RSA key generation in Intel SGX SSL. In *AsiaCCS*, pages 575–586. ACM, 2018. URL <http://doi.acm.org/10.1145/3196494.3196524>.

[77] Samuel Weiser, Andreas Zankl, Raphael Spreitzer, Katja Miller, Stefan Mangard, and Georg Sigl. DATA - differential ad-

dress trace analysis: Finding address-based side-channels in binaries. In *USENIX Sec.*, pages 603–620. USENIX Association, 2018. URL <https://www.usenix.org/conference/usenixsecurity18/presentation/weiser>.

[78] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Sec.*, pages 719–732. USENIX Association, 2014. ISBN 978-1-931971-15-7. URL <https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom>.

[79] Robert Zuccherato, Patrick Cain, Carlisle Adams, and Denis Pinkas. Internet X.509 public key infrastructure time-stamp protocol (TSP). RFC 3161, RFC Editor, August 2001. URL <https://datatracker.ietf.org/doc/rfc3161/>.

A mbedTLS vulnerable RSA keys

Table 2: RSA keys that follow leaking mbedTLS code paths. Missing parameters are marked with •. Note, the first row belongs to a key with all included parameters indicating that it leaks through CRT computation.

Group	N	e	p	q	d	d_p	d_q	i_q
CRT						•	•	•
						•	•	
						•		•
						•		
							•	•
							•	
								•
	•							
	•					•	•	•
	•					•	•	
	•					•		•
	•					•		
	•						•	•
	•						•	
CRT & d					•			
					•	•	•	•
						•	•	
						•	•	
						•		•
						•		
						•	•	•
						•		
	•					•		•
	•					•	•	•
	•					•		•

