Déjà Vu: Side-Channel Analysis of Mozilla's NSS

Sohaib ul Hassan
Tampere University
Tampere, Finland
sohaibulhassan@tuni.fi

Iaroslav Gridin
Tampere University
Tampere, Finland
iaroslav.gridin@tuni.fi

Ignacio M. Delgado-Lozano
Tampere University
Tampere, Finland
ignacio.delgadolozano@tuni.fi

Cesar Pereida García
Tampere University
Tampere, Finland
cesar.pereidagarcia@tuni.fi

Jesús-Javier Chi-Domínguez
Tampere University
Tampere, Finland
jesus.chidominguez@tuni.fi

Alejandro Cabrera Aldaya
Tampere University
Tampere, Finland
alejandro.cabreraaldaya@tuni.fi

Billy Bob Brumley
Tampere University
Tampere, Finland
billy.brumley@tuni.fi

ABSTRACT

Recent work on Side Channel Analysis (SCA) targets old, well-known vulnerabilities, even previously exploited, reported, and patched in high-profile cryptography libraries. Nevertheless, researchers continue to find and exploit the same vulnerabilities in old and new products, highlighting a big issue among vendors: effectively tracking and fixing security vulnerabilities when disclosure is not done directly to them. In this work, we present another instance of this issue by performing the first library-wide SCA security evaluation of Mozilla's NSS security library. We use a combination of two independently-developed SCA security frameworks to identify and test security vulnerabilities. Our evaluation uncovers several new vulnerabilities in NSS affecting DSA, ECDSA, and RSA cryptosystems. We exploit said vulnerabilities and implement key recovery attacks using signals—extracted through different techniques such as timing, microarchitecture, and EM—and improved lattice methods.

CCS CONCEPTS

• **Security and privacy** → **Cryptography**; **Public key (asymmetric) techniques**; **Digital signatures**; **Side-channel analysis and countermeasures**; **Software and application security**; *Cryptanalysis and other attacks.*

KEYWORDS

applied cryptography; public key cryptography; DSA; ECDSA; RSA; side-channel analysis; lattice-based cryptanalysis; software security; NSS; CVE-2020-12399; CVE-2020-12402; CVE-2020-6829; CVE-2020-12401

CCS '20, November 9–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7089-9/20/11.
https://doi.org/10.1145/3372297.3417891

ACM Reference Format:
Sohaib ul Hassan, Iaroslav Gridin, Ignacio M. Delgado-Lozano, Cesar Pereida García, Jesús-Javier Chi-Domínguez, Alejandro Cabrera Aldaya, and Billy Bob Brumley. 2020. Déjà Vu: Side-Channel Analysis of Mozilla's NSS. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20), November 9–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3372297.3417891

1 INTRODUCTION

Traditionally, SCA security research involves manual analysis of software libraries by actively triggering the execution paths of cryptographic schemes in isolation, and measuring information leaks. Constant-time software implementation is considered the most widely adapted countermeasure against these information leaks [43]. However, owing to a long list of SCA vulnerabilities in popular software libraries such as OpenSSL [2, 10, 13, 35, 36], we argue that for cryptographic library maintainers and developers, manual verification for constant-time behavior is a non-trivial task, and requires extensive knowledge about SCA security. This can result in SCA vulnerabilities being overlooked by *peer vendors*, especially when the issues are not directly reported to them.

A peer vendor is one that is "at the same horizontal level of the supply chain; peer vendors may be independent implementers of the same technology (e.g., OpenSSL and GnuTLS)" [20, p. 4]. In the context of our work, NSS, OpenSSL, GnuTLS, BoringSSL, LibreSSL, mbedTLS, WolfSSL, etc. all fit this definition of peer vendors, sharing the same market vertical, implementing (at least) several versions of the TLS standard and many of the implied cryptographic algorithms.

Inspired by documented multi-vendor security failures (detailed later in Appendix A), in this work we analyze the SCA security of Mozilla's NSS. OpenSSL's rich history provides a large corpus of security vulnerabilities, many of which are root caused to failure to use constant-time algorithms. We leverage this corpus to explore how well NSS has kept up with OpenSSL's significant improvements that accelerated after HeartBleed (CVE-2014-0160).

Taking NSS as our case study, we present a novel approach to SCA security, by developing a systematic methodology for library-wide automated identification of SCA leaks and flagging the vulnerable execution paths. Our findings reveal serious SCA deficiencies in NSS, which not only questions the current practice of vulnerability disclosure among peer vendors, but more importantly provide a general approach to automated SCA security validation for cryptographic library developers. Furthermore, we also perform multiple end-end attacks to highlight the severity of the discovered vulnerabilities and responsibly assist Mozilla to mitigate them.

Contributions. Briefly, the contributions of our work include: (i) combining the DATA [52] and TriggerFlow [24] frameworks to close the gap between identifying leakage and software / module / unit / regression testing, subsequently applied to NSS (Section 3) to discover, test, and exploit several novel vulnerabilities; (ii) an end-to-end network-based remote timing attack against NSS's DSA implementation (Section 4); (iii) discovery of a traditional timing attack vulnerability in NSS's ECDSA implementation (Section 5) in the context of nonce padding; (iv) an end-to-end ElectroMagnetic Analysis (EMA) attack on NSS's ECDSA implementation (Section 6) in the context of Elliptic Curve Cryptography (ECC) scalar multiplication; (v) an end-to-end microarchitecture attack on NSS's ECDSA implementation (Section 7) in the context of ECC scalar recoding; (vi) an end-to-end EMA attack on NSS's RSA key generation implementation (Section 8); (vii) improved lattice methods and empirical data for realizing several of these end-to-end attacks (Section 9).

2 BACKGROUND

2.1 Public Key Cryptography

DSA. Denote primes p, q such that q divides $(p-1)$, and a generator $g \in GF(p)$ of multiplicative order q. The user's private key α is an integer uniformly chosen from $\{1\,..\,q-1\}$ and the corresponding public key is $y = g^{\alpha} \bmod p$. With approved hash function Hash(), the DSA digital signature (r, s) on message m (denoting with $h < q$ the representation of Hash(m) as an integer) is

$$r = (g^k \bmod p) \bmod q, \quad s = k^{-1}(h + \alpha r) \bmod q \tag{1}$$

where k is a nonce chosen uniformly from $\{1\,..\,q-1\}$.

ECDSA. Denote an order-q generator $G \in E$ of an elliptic curve group E with cardinality fq and q a large prime and f the small cofactor. The user's private key α is an integer uniformly chosen from $\{1\,..\,q-1\}$ and the corresponding public key is $D = [\alpha]G$. With approved hash function Hash(), the ECDSA digital signature (r, s) on message m (denoting with $h < q$ the representation of Hash(m) as an integer) is

$$r = ([k]G)_x \bmod q, \quad s = k^{-1}(h + \alpha r) \bmod q \tag{2}$$

where k is a nonce chosen uniformly from $\{1\,..\,q-1\}$.

Point multiplication. During ECDSA signing, point multiplication $[k]G$ is the most computationally intensive part. For curves over prime fields, windowed non-adjacent form (wNAF) is a textbook method for performing point multiplication. Given a window size w, and a set of pre-computed points $\pm G, \pm[3]G, \cdots, \pm[2^{w-1}-1]G$, the ℓ-bit scalar can be recoded as

$$k = \sum_{i=0}^{\ell} k_i 2^i \qquad \text{where } k_i \in \{0, \pm 1, \pm 3, \cdots, \pm(2^{w-1}-1)\}$$

The wNAF point multiplication method (Algorithm 2) scans signed k_i digits performing a point *double* at each step, whereas the position of non-zero k_i decides on point *add*. The wNAF representation (Algorithm 1) reduces the number of non-zero scalar digits to about $\ell/(w+1)$, resulting in less point additions since it guarantees at most one out of w consecutive digits are non-zero.

RSA. According to PKCS #1 v2.2 (RFC 8017 [32]), 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:

$$d = e^{-1} \bmod \text{lcm}(p-1, q-1) \tag{3}$$

In addition, Chinese Remainder Theorem (CRT) parameters are stored for speeding up RSA computations:

$$d_p = d \bmod p, \quad d_q = d \bmod q, \quad i_q = q^{-1} \bmod p \tag{4}$$

Currently, the minimum recommended length for N is 2048 bits (i.e., p and q are 1024-bit primes) and e is fixed to a small value, typically 65537. Regarding RSA security, beyond traditional factoring Coppersmith [15] showed if we know half of the bits of either p or q it is possible to factor N in polynomial time, a fact we will utilize later in Section 8.

During key generation, a typical check is coprimality of e with both $p-1$ and $q-1$ often implemented with the binary extended Euclidean algorithm (BEEA) [40]. Algorithm 3 (and variants) computes the GCD of two integers a and b employing solely right-shift operations (SHIFTS) and subtractions (SUBS) instead of divisions. The BEEA control flow strongly depends on its inputs, and an SCA capable attacker differentiating between SUBS and SHIFT operations can recover information on a and b [1, 3, 5, 35, 51].

In the context of RSA, the integer arguments will be e (public) and $p-1$ or $q-1$, putting keys at risk. It is important to note that step 4 will never execute since e is always an odd number, and that during the first iterations $v > u$ as $p, q \gg e$.

2.2 Mozilla Network Security Services (NSS)

With its roots in Netscape Navigator and SSLv2, NSS is a Free and Open Source Software (FOSS) project tailored for Internet security and interoperating security (e.g., TLS) and crypto (e.g., PKCS #11) standards. The software suite consists of two libraries (`libnss` along with Netscape Portable Runtime `libnspr`) and over 70 CLI tools linking against them—e.g., certutil, pk1sign, p7sign, p7verify, signtool, etc.

Mozilla maintains the development infrastructure for NSS and is the main contributor due to the history of the project, as well as Firefox's significant browser market share. Yet currently Red Hat is also a major contributor due to their server-side enterprise use cases, and over the years other contributors include Sun Microsystems/Oracle Corporation, Google, and AOL.

Below the TLS layer, what differentiates NSS from other crypto-featured security software libraries is its abstraction of cryptographic operations. It features native PKCS #11 support for hardware and software security modules. In fact, at the API level, this is the interface at which linking applications drive the crypto—NSS serves crypto operations backed either externally through a PKCS #11 hardware token, or (by default) internally through its own PKCS #11 software token. This is different from all other major libraries, e.g., OpenSSL which provides access to crypto functionality either through its EVP interface (modern) or directly through low level APIs (legacy).

The comparison between NSS and OpenSSL is important due to the shared history and evolution of the projects. In particular, starting in 2001 Oracle (then Sun Microsystems) made fundamental FOSS contributions by integrating their ECC software into both projects[1], a new feature for both libraries. In that respect, the ECC parts of both libraries are forked from the same original code [26], yet have evolved independently over the last two decades.

NSS and SCA: previous work. Over the years NSS has had its fair share of cryptography implementation issues leading to several practical attacks on multiple primitives—some of these attacks are algebraic in nature such as RSA signature forging [33], DH small subgroup attack [45], and Lucky13 [7], just to name a few. In 2017, Yarom et al. [54] demonstrated that cache-bank conflicts leak timing information from an otherwise constant-time modular exponentiation function implemented in NSS, leading to RSA key recovery after observing 16000 RSA decryptions. In 2018, Ronen et al. [37] performed a padding oracle attack against RSA following the PKCS #1 v1.5 standard to recover long term login tokens used during TLS connections. Although this attack is well-known, the authors used recent SCA cache-based attack techniques, successfully reviving an old vulnerability. Finally, in 2019 Ryan [38] included NSS in his analysis of a new SCA attack enabled by a variable-time modular reduction function used during signature generation, allowing an attacker to recover ECDSA and DSA private keys.

2.3 Related Attacks

CVE-2016-2178. OpenSSL assigned this CVE based on work by Pereida García et al. [36]. The authors performed a cache-timing attack using the Flush+Reload technique against a variable-time sliding window exponentiation algorithm used during DSA signature generation, leading to full secret key recovery of OpenSSH and TLS servers co-located with an attacker. This vulnerability was present in the code base for more than 10 years and was enabled by a seemingly small software defect.

CVE-2018-0737. OpenSSL assigned this CVE based on work by Aldaya et al. [5]. The authors detected and identified several paths during RSA key generation potentially leaking information about the algorithm state. The authors performed a single trace cache-timing attack over the corresponding GCD function combining different techniques (including lattices) to achieve full secret key recovery.

CVE-2018-5407. OpenSSL assigned this CVE based on work by Aldaya et al. [2]. The authors discovered a novel timing SCA attack vector leveraging port contention in shared execution units on Simultaneous Multi Threading (SMT) architectures. With a spy process running in parallel, they targeted the variable-time wNAF point multiplication algorithm during ECDSA signature generation and recovered the `secp384r1` long term private key of a TLS server. Prior to the CVE and work done by Tuveri et al. [43], this implementation was the default choice for most prime curves, which was subsequently replaced by a timing resistant version.

2.4 Leakage Detection and Assessment

Differential Address Trace Analysis (DATA). This is a framework that detects potential side-channel leaks in program binaries; Weiser et al. [52] used the framework to analyze OpenSSL and PyCrypto. DATA works by observing the program execution with known and different inputs using Intel Pin[2], then analyzing the execution traces to detect differences in flow caused by different input, thus highlighting potential SCA vulnerabilities. This approach makes it mostly automated and universal with respect to SCA method. DATA led to the discovery of CVE-2018-0734 and CVE-2018-0735 issued by OpenSSL [50].

Triggerflow. This is a tool to selectively track code-path execution [24], facilitating testing-based SCA of cryptography libraries such as OpenSSL and mbedTLS [5, 22]. The power of Triggerflow comes from its simplicity, allowing a user to annotate source code by placing Points of Interest (POIs) and filtering rules, thus supporting false positive filtering. Then, Triggerflow compiles the source code, and runs a list of user-supplied binary invocations called "triggers", reporting context whenever a trigger reaches any of the user-defined POIs. Triggerflow can be adapted to Continuous Integration (CI) of the development pipeline for automated regression testing. Triggerflow does not support automatic POI detection, instead relying on other offensive methodologies and tools [18, 25, 52].

3 NSS: AN SCA SECURITY ASSESSMENT

In this section, we combine DATA [52] for SCA POI identification with Triggerflow [24] for extended POI testing. We apply this combination to assess the SCA security of NSS, in particular for its public key cryptography primitives.

DATA frameworks. DATA requires a "framework" for program analysis—a Bash script defining commands necessary to prepare the environment, run the program with given inputs and optionally supply a leakage model. The script uses a library included in DATA and supplies its own domain-specific callbacks. The end result is a script which accepts parameters such as algorithm, key size and processing phase, and runs DATA.

An NSS framework for DATA. For NSS, we created a framework analyzing signature creation with DSA, ECDSA, and RSA algorithms. First, we define DATA callback `cb_prepare_framework` which creates the NSS certificate storage if it does not exist yet. The storage includes an SQLite database storing all certificates generated by command-line tools. This callback runs in the beginning of every

[1] https://seclists.org/isn/2002/Sep/89

[2] https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

framework invocation. Second, we define `cb_genkey` which generates key pairs and certificates for a given algorithm using the NSS utility `certutil`. For RSA and DSA we use default key size, for DSA default parameters, and for ECDSA curves `secp256r1`, `secp384r1`, and `secp521r1`—the only legacy curves NSS features. The callback executes every time DATA needs a different key. Finally, for DATA analysis we sign a fixed small piece of data using the NSS utility `pk1sign` in the callback `cb_run_command` traced by DATA.

Supporting DATA code allows us to define all algorithms in a single file, using algorithm-specific code depending on arguments given in framework invocation. In our case, the only difference was the algorithm selection during creation of the certificate. The DATA software package includes example frameworks, as well as working frameworks for OpenSSL and PyCrypto.

Performance evaluation. We performed our experiments on an Intel Xeon Silver 4116 (Skylake) with 256 GB RAM. At peak, DATA framework consumed 120 GB memory to analyze the program. The exact amount depends on the stage and algorithm. High memory requirements and general resource consumption make it unsuitable for automated testing, but still an extremely useful offensive tool for vulnerability research.

Results. DATA output is a collection of potential leaks. It stores the leak data in XML, as well as in Python standard "pickle" serialization format. The included GUI can read this format, and includes tools for marking leak points for further review, as well as adding comments. Table 1 presents our aggregate DATA statistics, where the "Total" rows include also SQLite and/or other less relevant parts of NSS while (statically-linked, private) `libfreebl` handles the crypto arithmetic in NSS.

Combination of DATA and Triggerflow. DATA can help quickly determine areas of the code vulnerable to SCA, but it is—as shown before—expensive to run and this is unsuitable for automated testing. Thus, we combine DATA and Triggerflow in vulnerability research: first, we detect vulnerabilities once using DATA analysis, then we mark vulnerable areas with Triggerflow annotations and continuously and cheaply monitor the code for SCA vulnerabilities. This hybrid approach combines assisted vulnerability scanning of DATA and automatic inexpensive monitoring by Triggerflow. The approach is general and applicable to any library supported by the tools, so it can be applied to other cryptographic libraries as well.

Producing Triggerflow annotations from DATA results. Using the information from DATA GUI to guide manual code review, we determined the most critical areas of NSS potentially vulnerable to SCA. Next, we annotated each area with Triggerflow's `TRIGGERFLOW_POI`, further refined with `TRIGGERFLOW_IGNORE` when running multiple operations on annotated source code to eliminate false positives. This allows us to examine potential vulnerabilities in context, and led to several concrete vulnerabilities summarized in Table 2 and described in detail in the following sections. We further point out that in all the attacks, NSS library was compiled with debug symbol enabled while keeping the default configurations intact. Section 4–Section 8 present a more detailed description of the experiment environments and threat models.

Table 1: Statistics for our NSS framework in DATA.

Algorithm	Location	CF leaks	Data leaks
DSA	`libfreebl`	0	446
DSA	Total	2443	7435
ECDSA	`libfreebl`	0	1074
ECDSA	Total	2124	5890
RSA	`libfreebl`	666	804
RSA	Total	3593	11140

Table 2: Summary of SCA attacks.

SCA attack	Vulnerability	Target device	Application layer	Threat model
DSA timing (Section 4)	Nonce padding	Raspberry Pi3	Time Stamp Protocol	Remote
ECDSA timing (Section 5)	Nonce padding	Intel i7-7700	NSS `pk1sign`	Local
ECDSA Electromagnetic (Section 6)	Point multiplication	Allwinner Pine A64	Time Stamp Protocol	Physical proximity
ECDSA uarch (Section 7)	Scalar recoding	Intel i7-7700	NSS `pk1sign` (SGX)	Local, malicious OS
RSA Electromagnetic (Section 8)	Key generation	Allwinner Pine A64	NSS `certutil`	Physical proximity

Case study. A good example of this workflow is the vulnerability described later in Section 6. DATA correctly flagged the problematic line in `ec_compute_wNAF` (Figure 2), as well as 93 other potential data leaks. Inspecting the leak data in DATA GUI showed that all vulnerable places converge in the parent `ec_GFp_pt_mul_jm_wNAF`, which is suitable as a Triggerflow POI. After running Triggerflow NSS configuration, `pk1sign` triggered the annotation once for both curves `secp384r1` and `secp521r1`. This annotation could be included in an automatic SCA regression test for NSS.

4 DSA: LEAKAGE MEETS CONSTANTNESS

As mentioned previously, after [13] OpenSSL and several peer vendors decided to apply the nonce fixed bit length countermeasure to their code base. Unfortunately, this fix did not permeate to the DSA portion of NSS, leaving the library vulnerable to this flaw at least since 2011. We speculate that a very regular fixed window exponentiation (FWE) algorithm paired with constant-time cache access to pre-computed values provided a false sense of security, forgetting that the nonce requires its own protection against bit length leakage due to the fragility of the DSA algorithm w.r.t. SCA.

Analysis. Our tooling revealed the main root cause of the time leakage was directly attributed to the variable bit length of the nonce k during the computation $r = g^k \bmod p$ (Equation 1) in the upper level `dsa_SignDigest` function. Helped by leakage amplification in lower-level exponentiation functions, this flaw leaks a considerable amount of information on the MSBs of the nonce. To better understand the time leakage, we can highlight three important functions in the NSS library, from general to more specific: (i) The `dsa_SignDigest` function contains the logic to calculate the digital signature pair *(r, s)* by calling the corresponding high-level modular arithmetic functions; (ii) The `mp_exptmod` function is a wrapper function selecting a specific modular exponentiation function among several

available based on input values and flags set during compilation time—additionally it determines the window size to be used by the exponentiation function; (iii) The `mp_exptmod_safe_i` function computes and implements a cache-timing safe regular FWE algorithm based on the window size selected by the previous wrapper function.

After calculating the window size and just before calling the FWE function, the wrapper function modifies the bit length of the exponent by making it a multiple of window size. Thus, artificially increasing the amount of bits in the exponent and therefore the amount of windows to be processed by the FWE function. This means the leakage potentially occurs in multiples of the window size, reveling a total of $w \cdot i$ MSBs for each signature, where w is the window size, and i is the amount of windows skipped by the FWE due to shorter-than-average nonces. Therefore, the FWE function effectively amplifies the leakage and improves the resolution by widening the time gap it takes to process variable length exponents coming from the upper-level DSA signing function.

4.1 DSA: Remote Timing Attack

To demonstrate concretely the impact of the vulnerability, we exploit it remotely from the application layer through the Time Stamp Protocol defined in RFC 3161 [55] as implemented in `uts-server`[3]. The Time Stamp (TS) Protocol permits a trusted Time Stamp Authority (TSA) to digitally sign a piece of data—e.g., using DSA or ECDSA—confirming the data existed at that particular point in time, and allowing anyone with access to the TSA certificate to verify the timeliness of the data.

Target device. We used a Raspberry Pi 3 Model B plus board containing a 1.4 GHz 64-bit quad-core Cortex-A53 processor. The device runs stock Gentoo 17, and we set the board frequency governor to "powersave". We deployed `uts-server` on the target device, acting as the TSA, receiving TS requests over (the default) HTTP and generating TS responses.

The only supported backend cryptography library for `uts-server` is OpenSSL, therefore we use an OpenSSL loadable cryptographic module (engine) [42] to expose NSS DSA signature generation to the server. In general, the purpose of engines is to intercept OpenSSL low-level crypto functionality and carry out the operations internal to the engine, either HW or SW-backed. The `e_nss`[4] OpenSSL engine makes the use of NSS transparent to linking applications, in our case `uts-server`.

As an FOSS contribution, we submitted a PR to the `uts-server` project adding engine-backed key support, e.g., devices such as TPMs, HSMs, or generically PKCS #11 driven. In these instances, the key never leaves the device hence cannot be directly access by OpenSSL, only driven. Our contribution[5] addresses a three year old outstanding feature request on the project's issues page. In our case, this allows `uts-server` to transparently utilize the crypto functionality of NSS through `e_nss` and keys inside the NSS keystore through NSS's PKCS #11 software token view.

We patched, compiled, and deployed the latest `uts-server` v0.2.0, linking against an unmodified build of OpenSSL 1.1.1. We compiled and deployed an unmodified version of `e_nss`, linking to both—the previous OpenSSL build and an unmodified build of NSS v3.51, effectively transparently connecting the server to NSS through OpenSSL.

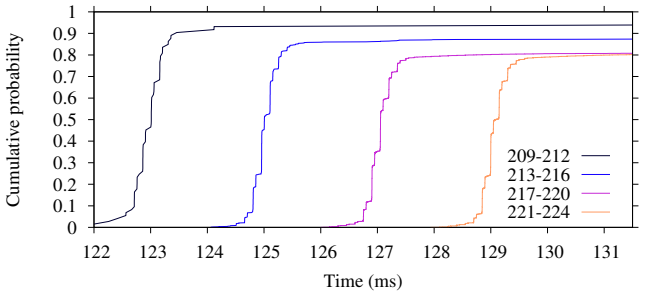

Figure 1: DSA time leakage in NSS on a remote scenario. Direct correlation between DSA signature generation time and bit length of the nonce.

Experiment setup. On one end we deployed the TS server on our RPi, on the other end we deployed a custom TS client on a workstation equipped with a 3.1 GHz 64-bit Intel i5-2400 CPU (Sandy Bridge), both communicating through a Cisco 9300 series enterprise switch over Gbit Ethernet. The workstation has 4x1 Gbit link aggregation to the switch and the RPi a single Gbit connection to the switch. Our custom client is a simple rust program that embeds a TS request in an HTTP request, establishing a TCP connection to the server, and starting a timer just before sending the request. The `uts-server` actively listens for TS requests and generates corresponding TS responses using NSS transparently through OpenSSL, replying back to the client as soon as the TS response is available. Once the TS response is received, the client stops the timing, closes the TCP connection, computes the latency, and finally stores the latency and the TS response pairs in a database. This operation is repeated as needed to gather more samples. Following the attack methodology from previous remote attacks [13, 22, 31], we divided our attack in two phases.

Collection phase. We collected 2^{18} samples using our custom client, and our timing analysis confirmed that our vulnerability analysis was correct—a direct correlation existed between the wall clock execution time of DSA signature generation and the bit length of the nonce used to compute the signature. We confirmed the nonce values using the ground truth private key from the NSS keystore. More importantly, Figure 1 shows that the time leakage is substantial, allowing an attacker to exploit the vulnerability even in a remote scenario.

Recovery phase and results. Section 9 describes in detail the lattice construction formalization, lattice parameters, lattice experiments, and results applied to our collected samples. In short, we observed a striking 99% (1536 samples) and 38% (1152 samples) success rate for recovering private keys in our remote timing attack scenario after performing a lattice attack.

[3] https://github.com/kakwa/uts-server
[4] https://roumenpetrov.info/e_nss
[5] https://github.com/kakwa/uts-server/pull/15

5 ARE YOUR NONCES REALLY PADDED?

In a nutshell, CVE-2011-1945 exploited the Montgomery ladder feature that it executes an iteration per scalar (i.e., ECDSA nonce k) bit, performing both *double* and *add* ECC operations regardless of said nonce bit value. This regularity offers protection against classical SCA that aims at recovering the nonce by tracking the sequence of ECC operations [29]. However as demonstrated in [13] this feature plays in favor of a timing attacker.

This highly regular feature combined with the fact that this algorithm executes $\lceil \lg k \rceil - 1$ iterations implies that the execution time is highly related to the effective bit length of k [13, 43]. Therefore, a timing attacker could learn information on k by measuring the execution time during ECDSA signature generation, with computing time dominated by the scalar multiplication.

In order to prevent this attack, Brumley and Tuveri [13] proposed a countermeasure that fixes the number of significant bits of the nonce using (5).

$$\hat{k} = \begin{cases} k + 2q & \text{if } \lceil \lg(k+q) \rceil = \lceil \lg(q) \rceil \\ k + q & \text{otherwise} \end{cases} \tag{5}$$

In response to this research and CVE-2011-1945, the *nonce padding* countermeasure based on (5) has been implemented not only in OpenSSL, but in other libraries as well. Mozilla NSS included the nonce padding countermeasure in 2011 in their high-level ECDSA function[6]. However, despite this intended fix, we uncovered the implemented countermeasure is ineffective. We found that after the padding, a lower-level scalar multiplication function reduces $\hat{k}$ modulo q, thus reverting the nonce to its original value (Line 25[7]).

This *nonce unpadding* opens the door to a timing attack against NSS. The library has different scalar multiplication algorithms implemented. For curve `secp256r1` it uses a constant-time scalar multiplication algorithm, yet a wNAF implementation for higher security curves `secp384` and `secp521r1`. This algorithm iterates through wNAF digits of the scalar, and the number of iterations depends on the wNAF representation length, eventually depending on $\lg(k)$. Therefore a timing attacker could learn information about k by measuring the duration of ECDSA signature generation.

Experimental validation. In order to validate this hypothesis, we developed a proof-of-concept to demonstrate that the execution time of NSS wNAF scalar multiplication is related to $\lg(k)$. For this experiment we build NSS v3.51 on an Ubuntu 18.04 LTS desktop workstation running on Intel i7-7700 3.60GHz (Kaby Lake). We measured the number of clock cycles consumed by the NSS library exported function for generating digital signatures: `SignData`. Our previous analysis (Section 3) reveals this function's call trace includes `ec_GFp_pt_mul_jm_wNAF` for $w = 5$ in the context of `secp384r1`. We collected 1M samples of `SignData` latency during ECDSA signature generation. Using the ground truth private key from the NSS keystore, we computed the secret nonces used in each signature, then estimated the cdf curves of the latency per effective bit length.

[6] https://hg.mozilla.org/projects/nss/rev/079cfc4710c7193ef73888394f4d4f935e03f241
[7] https://hg.mozilla.org/projects/nss/file/c06f22733446c6fb55362b9707fa714c15caf04e/lib/freebl/ecl/ecl_mult.c

Figure 4 shows these curves, aggregating those $\ell \leq 380$ in one single curve. This empirically demonstrates there is indeed a dependency between $\lg(k)$ and the time taken to produce an `secp384r1` ECDSA signature in NSS. With enough samples under the right conditions, we speculate this leak could be exploited using the lattice methods developed in Section 9, as demonstrated in [22].

6 LEAKING ECDSA KEYS THROUGH EMA

In general, wNAF has been subjected to a variety of SCA attacks on OpenSSL in the past—e.g., L1 and LLC cache timings [10, 12], EM [22, 23] and port contention [2]. As far as we know, we are the first to practically demonstrate an end-to-end attack on NSS wNAF implementation. To this end, we employed EMA to exfiltrate the ECDSA private key. Previous EM attacks on wNAF focused on only retrieving least significant bit positions [8, 22, 23]. In contrast, out attack uses an advanced multi-digit lattice formulation detailed in Section 9.2, making it possible to potentially use the entire EM trace to extract side-channel information, consequently, lowering the number of signatures required and reducing the data complexity of the attack.

Threat model. We assume an adversary is able to obtain a similar device to learn about the particular EM leakage (preparation phase), and furthermore gain access to close proximity of the target device while issuing ECDSA queries (attack phase). This model is consistent with the literature.

Experiment setup. Our setup includes a Pine A64-LTS powered by a 64-bit quad-core ARM Cortex A53 SoC. This target hardware device runs Ubuntu 16.04.5 LTS minimal with NSS v3.51. Similar to Section 4, we created a TS server instance using `uts-server` as our victim, this time with an `secp384r1` key. We measured the EM signals using a Langer LF-U 2.5 EM probe attached to a 40 db preamplifier, with the probe head positioned close to the target board to achieve good signal strength. We acquired the EM traces using a Picoscope 6404*C* USB digital oscilloscope with a maximum sampling rate of 5GSps supporting up to 500MHz bandwidth. To strike balance between lower computational cost and decent signal-to-noise ratio, we used a sampling rate of 150MSps instead.

Signal acquisition. We created a client responsible for sending TS requests over HTTP to our server and controlling the oscilloscope. The client first initiates the trace capture command followed by a TS request, then stopping the trace capture upon receiving the HTTP response from the server. We parsed the server response messages to retrieve DER encoded ECDSA signatures and the hash from the client request. The resulting EM traces along with their parsed ECDSA information were stored for offline signal processing and key recovery phase.

Signal processing. For a successful signing key recovery, the EM traces must go through signal processing to reliably extract the partial nonce information. From the signal analysis perspective, these partial nonces are encoded as the sequences of *double* and *add* operations during wNAF point multiplication as observed in the EM trace (Figure 2). Using this partial information from multiple signatures we can formulate a lattice attack as described in Section 9.2.

Since the captured trace contained the entire TS request window, the first step involved in locating and isolating the ECDSA point multiplication part. By performing a manual analysis, we found specific patterns in the trace pertaining to the start and end of the point multiplication. We used these as the templates (created by averaging over 20 EM traces) to cut the point multiplication window using squared Euclidean distances between root mean square values of the trace and template.

We then moved to the next phase, extracting the *double* and *add* sequences. This was a two-step process: finding the position of all the *add* operations and then finding all *double* operations between them. By performing a spectrum analysis we found clearly distinguishable low energy point multiplication *double* loop (*D*) and higher energy *double* and *add* loop (*DA*). To improve the detection we extracted two components of the signal: a band pass around 15 MHZ and a low pass at 5Mhz for the *DA* and *D* loops respectively. We demodulated them using a digital Hilbert transform and applied signal smoothing filter.

Using the first signal component, we extracted the *DA* loops using a similar approach to the point multiplication extraction, i.e., compute rolling squared Euclidean distances using the *DA* template. The *D* sequences were present in the signal as voltage peaks, however due to noisy artifacts in the trace simply extracting the peaks resulted in both false positive and negative peaks. Since each wNAF loop iteration performs only one *double* operation between two *DA* loops, they follow an almost periodic trend. Using this information together with the fact that *D* peaks can be approximated to rectangular pulses using root mean square, we computed pulse width to period ratios. By selecting an experimentally evaluated threshold for these ratios we were able to significantly increase the detection of the *D* loops with an overall error rate of less then 1%.

In practice, EM traces contain noise from various sources—OS preemption, acquisition noise, environmental and electrical noise—which can reduce the efficacy of signal processing phase. OS interrupts for instance are high energy signals and therefore easily distinguishable. We marked all such interrupts in the trace and recovered the sequence from the interrupt position till the end of the trace (i.e., interrupt to wNAF LSD). For other noise sources and small interrupts our sequence extraction resulted in around 6% of the total traces with less then 1% incorrect guess for *D* and/or *AD* loops. Additionally, we applied heuristics on the recovered sequences to filter those which violated the wNAF encoding rules.

In total we collected 300 signatures, which left us with 211 signatures after performing sequence extraction, containing 13 errors. We filtered out the sequences with a length of at least 384, which resulted in a total of 66 signatures. By using our lattice formulation, we were able to recover the private key with as few as 30 signatures as described Section 9.2. A clear advantage of using more information per trace is reflected in the low number of signatures required. To put this into perspective, the ECDSA attack presented by [23] utilized only LSDs of the nonce (last non-zero digit and trailing zeros), required 3060 signatures and even the `secp256k1` curve at a substantially lower security level.

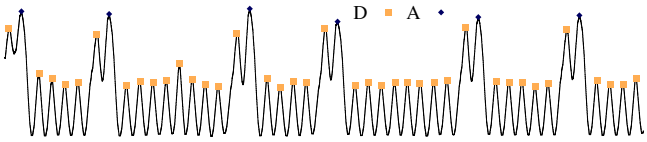

Figure 2: EMA trace showing part of the wNAF point multiplication with marked *double* (D) and *add* (A) operations. The filtered trace clearly shows two distinguishable loops: lower energy D and higher energy DA.

7 UARCH SCA ON SCALAR RECODING

This section presents an SCA attack that aims at recovering the wNAF representation of an ECDSA nonce, similar to Section 6 that targeted the ECC operation sequence. We instead target the wNAF recoding itself (Algorithm 1). This target, in addition to representing a novelty as it is not a common target in the literature, represents a challenge due to its low temporal and spatial granularity as detailed later. Moreover, we will be able to recover the same information as in Section 6. Additionally, employing the same channel we will gain extra information on the wNAF representation, recovering the sign of its non-zero coefficients.

Threat model. Although Section 6 and this section aim at recovering very related data, their threat models and targets differ significantly, therefore they are distinct attacks. Controlled channel attacks belong to a class of threat models when the adversary has control over the targeted computing platform except the targeted algorithm itself and its processed secrets [53]. One instance of this threat model is provided by trusted executed environments (TEEs), like Intel Software Guard Extensions (SGX) on Intel microprocessors [16]. In Intel SGX nomenclature, an *enclave* is software running on a secure space that provides confidentiality and integrity of *enclave* code and data, even in presence of a compromised OS (i.e., adversary). However, it delegates SCA protections to developers, allowing attackers to use privileged OS resources when gathering SCA signals that reveal information on enclave secrets.

Intel SGX leaves control of memory pages to the OS, which an adversary can use to track the sequence of executed memory pages by a targeted enclave [39, 47, 49, 51, 53]. An attacker first marks a memory page with SCA relevance as *non-executable* and launches the enclave. If the enclave executes that memory page, a page fault is generated and handled by the OS (i.e., adversary), hence the attacker learns the targeted memory page was executed [53]. Applying this process for a set of memory pages allows the adversary to track the sequence of executed memory pages, thus potentially leaking secret data processed by the enclave. This attack works at 4KB granularity, yet sufficient to recover some secrets on low granularity targets as detailed below.

Experiment setup. To track the sequence of executed memory pages of an enclave, we used the `SGX-Step` framework proposed by Van Bulck et al. [46] and integrated into the `Graphene-SGX` framework [41]. `Graphene-SGX` allows running unmodified code inside an Intel SGX enclave, providing a straightforward approach to execute NSS code in an enclave and assess its SCA resistance. It is worth noting the `Graphene-SGX` framework is not a requirement for the attack, it just

Table 3: Functions of interest and tracked pages.

Function	Memory page
ec_compute_wNAF	0x08000
mp_isodd	0x1f000
mp_add_d, mp_sub_d	0x22000
s_mp_cmp_d	0x24000

simplifies porting NSS to SGX. We performed our experiments on Ubuntu 18.04 LTS running on a desktop workstation featuring an Intel i7-7700 microprocessor (Kaby Lake) with Intel SGX enabled.

The NSS (private) function ec_compute_wNAF computes the wNAF encoding. Figure 5 shows a snippet of this function, consisting of a main loop that encodes k into its wNAF representation, which is stored in out. In our build this snippet compiles to 363 bytes, hence much smaller than a memory page; however its callees are located on different pages.

The execution flow of this function is related to the wNAF representation of k to different degrees. For instance, it is easy to verify the results of conditions at lines 3 and 4 allows retrieving the indices of the non-zero coefficients of the wNAF representation. A further analysis revealed it is also possible to extract the sign of these coefficients, by inferring the condition result at line 11. This additional information will reduce lattice computation time to recover ECDSA keys as shown in Section 9. Note that this *sign leakage* is due to NSS API mp_sub_d only supporting unsigned digits as commented in Figure 5. This is just another example that the implementation has the final word regarding SCA (cf. Algorithm 1).

Table 3 shows the relation between functions of interest for SCA and the tracked memory pages we used to record the execution flow of ec_compute_wNAF. The first three memory pages allow extracting the unsigned non-zero coefficients of the wNAF representation, while additionally tracking s_mp_cmp_d allows sign recovery.

The memory page sequence of mp_add_d and mp_sub_d are almost identical. However, a subtle difference allows distinguishing them. mp_sub_d call trace reveals s_mp_cmp_d executes more times in mp_sub_d than in mp_add_d, making it a good s_mp_cmp_d distinguisher to determine the condition result at line 11 in Figure 5.

We developed an SGX enclave that generates ECDSA signatures using curve secp384r1 through NSS pk1sign. We targeted this enclave collecting 1000 traces while tracking the memory pages in Table 3. Using the ground truth private key, we verified the non-zero coefficient signs of the nonce wNAF representation used to generate those signatures were perfectly recovered in all cases. As detailed in Section 9.2, after applying lattice cryptanalysis we were able to recover the private key with very high probability.

8 LEAKING RSA KEYS THROUGH EMA

We now present another EM attack: exploiting the BEEA algorithm during RSA key generation. During NSS RSA key generation, the function RSA_PrivateKeyCheck makes two calls to the vulnerable function mp_gcd to validate if the public exponent e is relatively prime to $p-1$ and $q-1$.

Threat model. Our attack utilizes a single EM trace to recover the private key, since the attacker only gets one shot at the key. Hence our model assumes the attacker can either trigger RSA key generation or knows when it occurs. The threat model is otherwise the same as in Section 6.

Experiment setup. For capturing EM traces during RSA key generation, we use the same setup and target device described in Section 6.

8.1 Signal Acquisition and Processing

Using the NSS certutil tool, we issued self signed certificates requesting a fresh 2048-bit RSA key pair each time, while ensuring sync with the oscilloscope's signal capture window. We logged the key metadata and the corresponding EM traces for further analysis and key recovery. We captured 1100 independent traces and used 100 as a *training set* to adjust signal processing and error correction phases of the attack. The remaining 1000 are left to present statistics of the proposed attack (Section 8.3).

To increase the success rate of key recovery, we preprocessed the traces to remove high frequency noise and detect noise sources such as interrupts. We applied low pass filter followed by a digital Hilbert transform. We then moved on to extracting p and q traces from the entire key generation trace. By applying templates obtained from two distinctive patterns, at the start of the first and the end of the second GCD computation, we extracted the signal of interest. Finally to cut the traces into p and q, we used the fact that the signal mean changes abruptly when the second GCD computation starts, creating a distinctive dip in the trace.

After the trace preprocessing, we selected a single trace to serve as a template. We divided this template trace into 17 windows (cf. Figure 3). For each window we used peak extraction to identify SUBS operations along the trace, while the distance between those peaks relates to the number of SHIFTS operations between two SUBS. The distance between peaks is decreasing along the trace, as operations require less and less time as the integers u and v from Algorithm 3 decrease in magnitude. For each window we create a linear regression model that relates the distance between peaks with the number of SHIFTS operations produced between two SUBS operations. This way, the unit distance for a SHIFT operation is not the same for different windows, and we empirically found 17 to be the minimum number of windows containing the maximum possible number of peaks, for linear regression models to accurately relate distances with SHIFT operations. Once the 17 models are generated, it is possible to select any trace, cut it in 17 windows with the same number of peaks within them and recover the entire sequence of operations made by the BEEA, applying the regression model corresponding to each window.

In a noise-free scenario, using the whole sequence of SHIFTS and SUBS it is possible to recover the primes [4, 5]. However, due to the noisy nature of EMA, errors possibly exist in the recovered sequences for p and q. Therefore, instead of trying to recover the full sequence we aimed to recover a partial one with sufficient information to retrieve a prime. Coppersmith [15] proposed an algebraic approach that allows to recover about half the bits of a prime knowing the other half employing lattice methods. Using [15], Aldaya et al. [5] showed it was practically possible to recover a 1024-bit prime knowing its 522 least significant bits with very high probability in less than 10 minutes. Therefore, we adopted a similar approach to [5], taking into account its lattice implementation is

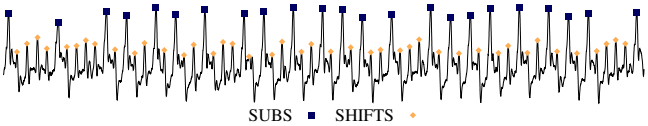

Figure 3: Window selected from a random trace. SUBS operations represented by peaks and SHIFTS by the distance among them.

open source. In what follows we treat this lattice-based cryptanalysis as a *lattice oracle* that takes 522 bits of a prime as input and outputs the remaining bits. Following established leakage models of the BEEA [3, 5], for recovering t bits we need to recover a sequence such that the number of total SHIFTS is at least t. This implies we need to gather about half a trace (i.e., $t = 522$), considerably reducing the influence of noise.

8.2 Error Correction of Noisy RSA Keys

In each key generation, the function `mp_gcd` executes for both $(e, p-1)$ and $(e, q-1)$, thus we have two traces per key generated. Hence two shots to recover one of the secret primes, p or q, needed to compute the private key. We address the problem of retrieving the bits from the recovered sequence using two different approaches. The first approach consists of trying to retrieve the prime using only the recovered sequence from the trace corresponding to a single prime, with a classical brute force mechanism.

Our linear regression models can make different mistakes during the task of retrieving the SHIFTS count between SUBS peaks, and a single error could lead to incorrect recovery of the secret prime. For that reason, it is necessary to detect components that give hints indicating error due to the way they were determined.

Error modeling. During our sequence recovery process we noticed there are three frequent error types: (i) The model finds a distance corresponding to 0 SHIFTS between two SUBS operations, which is a contradiction; thus, we delete that component from the vector, and consider the surrounding components susceptible to error. This normally happens because the acquisition of the peak is not clean, and we find in the trace two peaks extremely near to one another, that actually should be represented by only one peak. (ii) The model recovers the number of SHIFT operations from linear regression and requires rounding. As the fractional part nears 0.5 the model can decide incorrectly—we consider every component with its fractional part in the range $[0.3, 0.5]$ to be error-susceptible. Finally, (iii) we noticed that for high SHIFTS count the regression model accuracy decreases. This is due to the fact that a higher number of SHIFTS is increasingly improbable, so we had considerably less samples of high SHIFTS count during the procedure to generate our linear regression models. For this reason, we consider every component finding a SHIFTS count exceeding 8 to be error-susceptible.

To fix these potential errors, we propose the following corrections: (i) Modify the component by ±1, allowing three possibilities for each component (counting the original retrieved); (ii) Modify the component by ±1 depending if the rounding operation leads to the next or previous natural number (two possibilities); (iii) Modify the component by +1 and +2 (three possibilities) since, from an analysis done in the training set, we found the model tends to underestimate the real value when the SHIFT count between two peaks is relatively high.

Finally, it is important to note that our model is not able to retrieve the first iterations of the algorithm, i.e., the SHIFTS and SUBS operations are unknown during these iterations. However, we observed that no more than four SUBS operations were lost at the beginning of the sequence. For that reason, we consider all possible combinations of lost SUBS operations up to four and up to six SHIFTS[8] between SUBS, leading to a total of $\sum_{z=0}^{4} 6^z = 1555$ possibilities to recover the leading sequence of SHIFTS and SUBS operations.

Feasible exhaustive search error correction. Concerning the brute force procedure, we used the training set to recover statistics about the worst case computational cost of this approach. Note in this approach the attacker has two shots per trace to recover the first 522 bits of a prime. After our analysis, we verified that all traces suffer from missing iterations at the beginning, therefore the attacker needs 1555 calls to the lattice oracle to succeed in the worst case. At the same time, we successfully retrieved 14 out of 100 keys with this low computational cost—the recovered trace matched the ground truth sans the missing iterations. Generally, considering a *low* computational cost adversary (i.e., less than 17 error hints) it is feasible to recover 51 out of 100 private keys using only the bruteforce approach. In the first row, Table 4 gives the statistics about the associated worst case computational costs. However, beyond these numbers, we highlight it is feasible for an adversary to solve this problem using an exhaustive search approach.

Combined error correction. It is also possible to exploit the leak redundancy in the EM traces of p and q, combining them to fix errors. This is a common technique to recover noisy RSA keys, useful in our scenario considering the noisy nature of an EM side-channel attack. It was first introduced by Percival [34] and later extended and formalized by Heninger and Shacham [27]. For surveys about error correction in noisy RSA keys the reader can consult [5, 27, 30].

In our work, we use the approach proposed in [5] due to the similarity between the error types handled there and our observations. Their error correction approach belongs to the binary *extend-and-prune* algorithm class. Where, in our RSA context we obtain a binary sequence that represents the execution flow as explained in Section 8.1. However, despite reusing their algorithm we use it in a different scenario not originally considered, which we expand on during the experimental evaluation.

A binary *extend-and-prune* algorithm for fixing errors in RSA keys processes noisy binary sequences *expanding* them to a set of candidates considering possible error sources: (i) error at p_i, (ii) error at q_i, (iii) error at both p_i and q_i, (iv) no error at all. This expand process ensures that the relation $N = pq \bmod 2^i$ holds, where i represents the sequence index. Avoiding candidate space explosion requires a *prune* procedure.

The *prune* approach proposed in [5] is to use a set of filters over the candidates, e.g., hard limiting the number of candidates that will not be pruned, total number of errors since start (i.e., promoting those candidates with less errors since start). For a full description

[8] Higher values are possible, but increasingly improbable.

on the filters employed we refer the reader to [5]. In our work we used an unmodified version of the error correction algorithm employed in [5].

This algorithm could output a maximum of 150k candidates per trace, indicating that a ranking would be useful to reduce the number of calls to the lattice oracle. This algorithm incorporates a candidate enumeration approach that, according to the experiments in [5], ranks the correct solution at first position with high probability.

8.3 Extended Experiment Results

For this extended experiment we used 1000 EM traces as described in Section 8.1. From each of these traces we extracted the sequences of SHIFTS and SUBS corresponding to the processing of p and q. We used the training set to adjust the combined error correction algorithm, comparing observed trace recovery with the ground truth. Hence we decided to use the same configuration employed in [5].

As described during the exhaustive search approach for solving errors in p or q, it is common that traces lack information about the first iterations, also in some of them recovering an additional iteration. These errors cannot be handled by this error correction algorithm, therefore implying a failed recovery [5].

In our training set, the number of missing iterations ranges from zero to four. A straightforward approach to solve this was explained in Section 8.2, generating 1555 candidates per prime trace. Therefore, considering the traces of p and q the number of candidates exploded to $1555^2 \approx 2^{21}$. It is then possible to launch the error correction algorithm and see which one gives a solution. However, while this method is feasible, we decided to explore another approach that considerably reduces the number of candidates.

Instead of exhaustive searching the number of missing leading iterations and the SHIFTS count inside them, we bruteforce only the iteration count and fix SHIFTS in all missing iterations to one. It is likely these filled iterations contain errors, but our hypothesis is the error correction algorithm will be able to fix them automatically. With this approach the number of candidates reduces from 2^{21} to 25.

Due to this considerable reduction, in addition to attempting to solve missing iterations we decided to handling cases where traces have an additional leading iteration. Removing the first iteration and treating it as a missing one gives a total of 36 traces per original trace. This approach was not considered in [5], where the authors confirmed that 30% of their traces have missing information at the beginning, but did not solve it—a gap our work fills.

We tested this approach on the training set, obtaining a success rate of 64%. After manually inspecting the training set, we observed a maximum success rate of 67/100 using this combined error correction algorithm. In that sense, 64/67 is sufficiently high for our purposes.

One interesting feature is the algorithm spends considerably less time processing a trace with the correct number of iterations at the beginning compared to incorrect. This is important because the computing time is one good indicator if there is a solution in the processed trace. This allows us to quickly detect which trace out of the 36 has the correct fix for the missing iterations.

Table 4: Number of calls to the lattice oracle.

Method	Min	Median	Mean	Max
Bruteforce	1555	2^{22}	2^{26}	2^{30}
Combined	1	1	3	720

After this training, we applied this procedure with the same configuration parameters in a large set to estimate the success rate of the full attack. Fixing the number of bits to recover to 522, we expanded 1k traces on their 36 candidates each and executed the error correction algorithm on all of them (36k) limiting the computing time to 15 min per trace. The results are as follows: 587 traces terminated in time and after recovering the remaining bits using well-known lattice techniques, we recovered 565 independent RSA-2048 private keys. The number of calls to the lattice oracle had an impressive median of one while achieving a success rate of 56.5%. Table 4 shows more statistics about this method.

Data show in Table 4 demonstrates that an EM attack on binary GCD algorithms is a real threat to SoC devices. This table is not about comparing two approaches: rather providing experimental data of attack feasibility using different methods, where both have room for improvements.

9 ENDGAME: LATTICE-BASED ANALYSIS

ECDSA and DSA signing both return a pair (r, s) such that

$$s \cdot k \equiv h + \alpha \cdot r \bmod q \tag{6}$$

where α, h, and k correspond with the private key, hash, and nonce. Additionally, the integer r coincides with the output of a procedure that performs an exponentiation (DSA, Equation 1) or point multiplication (ECDSA, Equation 2). The vulnerabilities in Section 4–Section 7 provide varying degrees of leakage for each nonce, and in this section we utilize that for lattice-based cryptanalysis.

The private key recovery problem reduces to a Shortest Vector Problem (SVP) or Closest Vector Problem (CVP) instance of a given lattice. In both SVP and CVP instances, one proceeds by reducing the lattice with the LLL or BKZ procedures; the next step is looking for a short or close vector on the lattice, respectively. However, any CVP instance with input lattice B and vector u can be mapped into an SVP instance by looking for a short lattice basis vector in the lattice

$$\hat{B} = \left[\begin{array}{c|c} B & \vec{0} \\ \hline \vec{u} & q \end{array}\right] \tag{7}$$

In this section, we focused on constructing a suitable CVP instance, i.e., the lattice B and vector $\vec{u}$.

9.1 DSA Endgame: Lattice Attack

In Section 4 we analyzed a timing vulnerability and then we collected traces containing a variable amount of bits leaked during DSA signature generation—we now show the lattice formalization enabling private key recovery.

Lattice construction. Assume we have a sample of size N and elements of the form $(\omega_i, r_i, s_i, h_i)$ where ω_i is the elapsed time of signing a message with hash h_i and signature (r_i, s_i) satisfying

Equation 6. Next, one proceeds by sorting the sample according to ω_i and retaining the $f \ll N$ fastest ones, expected to correspond with shorter-than-average nonces. After filtering, the next step is to construct a suitable lattice that allows private key recovery.

Recall, in the filtered sample each nonce k_i *should* have bit length smaller than or equal to $m = \lg(q)$, and then $k_i < q/2^{\ell_i}$ for some positive integer ℓ_i. Moreover, the inequality $h_i/s_i + \alpha \cdot r_i/s_i \bmod q = k_i < q/2^{\ell_i}$ is crucial because it ensures the existence of integers $\lambda_i \in \{-q \,..\, q\}$ such that $\alpha \cdot (2W_i \cdot t_i) - (2W_i \cdot \hat{u}_i + q) - (2W_i \cdot \lambda_i \cdot q) \leq q$ where $t_i = r_i/s_i \bmod q$, $\hat{u}_i = -h_i/s_i \bmod q$, and $W_i = 2^{\ell_i}$. To be more precise, with $d \ll N$ a positive integer, the dimensional-$(d+1)$ lattice

$$B = \begin{bmatrix} 2W_1 \cdot q & 0 & \cdots & \cdots & 0 \\ 0 & 2W_2 \cdot q & \ddots & \cdots & \vdots \\ \vdots & \ddots & \ddots & 0 & \vdots \\ 0 & \cdots & 0 & 2W_d \cdot q & 0 \\ 2W_1 \cdot t_1 & 2W_2 \cdot t_2 & \cdots & 2W_d \cdot t_d & 1 \end{bmatrix} \tag{8}$$

and the integer vectors $\vec{u} = (2W_1 \cdot \hat{u}_1 + q, \ldots, 2W_d \cdot \hat{u}_d + q, 0)$, $\vec{z} = (\lambda_1, \ldots, \lambda_d, \alpha)$, and $\vec{y} = (2W_1 \cdot \nu_1, \ldots, 2W_d \cdot \nu_d, \alpha)$ satisfy $\vec{z}B - \vec{u} = \vec{y}$ with $\nu_i \in \{-(q-1)/2 \,..\, (q-1)/2\}$ the signed modular reduction of $\hat{u}_i + q/(2W_i) \bmod q$.

Lattice parameters. Feasible private key recovery depends on lattice parameters N, f, and d. Recall, N determines the number of measured signatures, and each signature requires a nonce k_i randomly and uniformly drawn from $\{1 \,..\, q-1\}$. Notice the expected number of nonces with ℓ MSBs clear is approximately $N/2^{\ell}$. Our goal is to determine the expected number of MSBs clear for each nonce corresponding with the filtered sample size f.

Denote ℓ the number of clear MSBs for the dimensional-$(d+1)$ lattice construction. Each column of the lattice has ℓ correlated bits to the private key and, assuming the private key has m bits, the expected number of independent bits in a sub-sample of size d is $m \cdot (1 - (1 - \ell/m)^d)$ [9]. Then setting $d = c \cdot m/\ell$, the probability that a random sub-sample of size d has m independent correlated bits is $1 - (1 - \ell/m)^d \approx 1 - e^{-c}$ and $c = 1.25$ is enough to ensure a "small" lattice dimension and success probability of at most 0.71. Hence we set $c = 1.25$ in what follows.

Increasing f decreases the expected number of clear MSBs, hence the best configuration is to set $f = \delta \cdot d \approx d$ with $\delta \in (1, 2]$ and $f = N/2^{\theta}$ where $\theta = \lg(N) + \lg(\ell) - \lg(m) - \lg(1.25 \cdot \delta)$. Finally, the existence of noise in the measurements necessitates a certain degree of freedom for the number of clear MSBs. Recall, θ is the expected number of clear MSBs (for noise-free measurements) and ℓ is the fixed number of clear MSB's to be used, and therefore the quantity $\theta - \ell$ gives an idea of how much large ℓ can be for a fixed sample size N. Figure 6 illustrates the degree of freedom assuming $\ell \in \{4 \,..\, 11\}$ and different sample sizes. Section 9.3 shows experiment results for the lattice applied to our collected samples during the remote timing attack scenario.

[9] Each correlation can be viewed as an ℓ-tuple with entries determined by the bit positions of the private key that are correlated. Thus, the number of different positions in a sample of d tuples with entries in $\{0 \,..\, m-1\}$ will determine the number of independent bits.

9.2 ECDSA Endgame: Lattice Attack

This section describes two kinds of lattice constructions, depending on the SCA nature. *Unsigned* applies to signals where point subtractions cannot be distinguished from point additions during wNAF point multiplication (cf. Section 6). *Signed* is a stronger attack in the sense that point subtractions *can* be distinguished, yielding the sign of each wNAF representation coefficient (cf. Section 7). The lattice constructions utilizing said leakage are based on those of van de Pol et al. [48] and Allan et al. [6], respectively, summarized as follows. With wNAF window width w, knowing the position of two consecutive non-zero coefficients κ_j and $\kappa_{j+\ell}$ leads to an equation with z α-correlated bits. Moreover, each column of the lattice B is determined by the pair $(\kappa_j, \kappa_{j+\ell})$ along with public values h and (r, s).

In addition, Equation 8 describes B but having $t = r \cdot s^{-1} \cdot 2^{m-j-\ell-1} \bmod q$, $W = 2^z$ where $z = \ell - w$ and $z = \ell - w + 1$ for the unsigned and signed approaches, respectively. In both approaches, $\vec{u}$ has the same shape like in the timing attack but without the term q, and either: (i) $\tilde{u} = 2^{m+w-\ell-1} - (h \cdot s^{-1} + 2^{j+w} - 2^{j+\ell}) \cdot 2^{m-j-\ell-1} \bmod q$ for the unsigned case; or (ii) $\tilde{u} = (2b+1) \cdot 2^{m+w-\ell-2} - (h \cdot s^{-1} \cdot 2^{m-j-\ell-1}) \bmod q$, where b denotes the sign of the coefficient $\kappa_{j+\ell}$ for the signed case.

Remark. The above equations are assuming each coefficient κ_j in the wNAF representation of k satisfies $-2^w < \kappa_j < 2^w$. However, our case study uses a modified wNAF representation such that $-2^{w-1} < \kappa_j < 2^{w-1}$; that is, replacing w with $w-1$.

9.3 Lattices at Work

To illustrate the practical implications of SVP instances used for private key recovery, we implemented our lattice-based cryptanalysis in Python 3. After constructing the dimensional-$(d+2)$ lattice $\hat{B}$ from Equation 7, we proceed by applying the method by Gama et al. [21]. The main idea is to reduce (using BKZ) another dimensional-$(d+2)$ lattice $\tilde{B}$ which is computed by shuffling the columns of $\hat{B}$ and multiplying it by a unimodular matrix with low density approximately equal to $(d + \sqrt{d})$. The goal is looking for a short lattice basis vector in the reduced lattice of $\tilde{B}$: if private key recovery is unsuccessful, update $\hat{B}$ as the reduced matrix of $\tilde{B}$ and repeat the procedure. Finally, our implementation solves the corresponding SVP instance with help of the BKZ reduction included in `fpylll`[10], a Python wrapper for the `fplll` C++ library [17].

We focused our experiments on private key recovery for DSA and ECDSA procedures with a 224-bit and 384-bit q, respectively. Each experiment consisted of the following steps: (i) select a random sample of size N from the SCA data; (ii) construct a random dimensional-$(d+2)$ lattice $\hat{B}$; (iii) look for a short lattice basis vector in $\hat{B}$ allowing private key recovery; goto (ii) if unsuccessful.

We labeled an experiment *successful* when recovering the private key according to the above steps and fixing the maximum lattice constructions to correspond to roughly 4 h of wall clock single core CPU time. The experiments with 224-bit DSA instances assume $\ell = 4$ clear MSBs and use the suitable lattice dimension proposed in Section 9.1. On the another hand, experiments corresponding with 384-bit ECDSA have a different lattice nature not determined

[10] https://github.com/fplll/fpylll

in terms of clear MSBs but by the distances between two non-zero wNAF coefficients. However, our lattice dimension choices are based on the analysis given in Section 9.1 and, because of the reduced sample size compared to the DSA case, we increased the lattice dimension until reaching a (possible) high success probability. The signed wNAF trace approach ensures one more leakage bit and smaller sample sizes with a higher success probability than the (unsigned) wNAF trace approach. For both 224-bit DSA and 384-bit ECDSA, the small density of about $(d + \sqrt{d})$ permits (at each step in the method by Gama et al. [21]) the use of a "random" lattice with small difference from the reduced one, and thus the BKZ reduction cost is minimized. In other words, we observe our choice $(d + \sqrt{d})$ allows a large number of different lattices in a fixed yet reasonable amount of time, and therefore the success probability increase.

Additionally, the random samples used in 224-bit DSA instances correspond with the data obtained by the remote timing attacks presented in Section 4. Conversely, the random samples used in the (unsigned) wNAF traces of 384-bit ECDSA instances are the ones obtained by the EM-based analysis presented in Section 6. As far as we known, this is first application of multi-digit lattice methods to EM-based signals. And finally, the random samples corresponding to signed wNAF traces of 384-bit ECDSA instances were obtained by the analysis from Section 7.

To have a better understanding how practice meets theory, we measured the elapsed time and number of random lattices required in each successful experiment. Table 5 summarizes the average elapsed time (in minutes) and the portion of successful experiments over all 1000 runs. Additionally, Table 5 also ilustrates the minimum, maximum, median, mean, and standard deviation of the number of lattice constructions performed. Moreover, from Table 5 we can see that our improvements bring us a success probability of 0.38 (with sample size 1152) for 224-bit DSA, 0.14 and 0.83 (both with sample size 30) for wNAF traces and wNAF signed traces of 384-bit ECDSA respectively.

Attack	N	$d+2$	Min	Max	Median	Mean	Stdev	Time	Ratio
Timing (Section 4)	2048	78	1	1374	25	59.9	102.4	0.3	0.99
	1536		1	4826	6	48.9	255.7	0.1	0.99
	1280		1	10169	6	67.4	450.1	0.1	0.74
	1152		1	10726	7	148.3	839.9	0.1	0.38
wNAF (Section 6)	40	92	1	2522	77	273.5	492.4	2.1	0.11
	40	132	1	50	2	4.9	7.9	0.9	0.09
	40	172	1	158	2	7.1	19.4	1.3	0.08
	30	92	5	3673	522	781.4	872.7	14.5	0.05
	30	132	1	1855	130	306.3	407.1	9.8	0.14
	30	172	1	2008	173	322.6	425.0	22.1	0.13
wNAF, error-free (Section 6)	40	92	1	2725	35	197.4	425.4	1.0	0.92
	30	92	1	2814	357	641.9	711.6	7.7	0.23
	30	132	1	1969	178	370.5	442.8	13.6	0.73
	20	132	47	1616	646	671.8	524.8	47.5	0.02
	20	172	2	1617	771	792.6	517.1	97.5	0.02
wNAF, signed (Section 7)	40	92	1	2817	3	80.2	277.4	0.3	0.94
	30	92	1	2854	101	430.5	672.6	2.4	0.44
	30	132	1	840	13	76.2	145.1	1.9	0.83
	20	132	1	1893	363	569.3	622.3	18.5	0.03
	20	172	4	2127	663	782.7	643.7	64.2	0.04

Table 5: Minimum, maximum, median, mean, and standard deviation of the number of lattice constructions performed. Timings are in minutes, and correspond with the median of the successful experiments.

10 CONCLUSION

In this work, we presented an extensive SCA security evaluation of Mozilla's NSS—from discovering vulnerabilities to performing key recovery attacks. Our methodology combines an automated leakage assessment framework DATA to identify potential SCA leaks, and the Triggerflow tool to track the code paths that lead to them. Applied during NSS invocations of DSA, ECDSA and RSA, the results led to the discovery of some serious SCA security flaws in DSA and ECDSA nonce padding, ECDSA point multiplication and scalar encoding as well as RSA key generation. To demonstrate real-world security impact, we also performed several end-to-end attacks at the application level—remote timing attack on DSA (research data released [44] in support of Open Science), microarchitecture attack on ECDSA nonce encoding, EM attack on ECDSA point multiplication and EM attack on RSA key generation. Finally, we summarized the results of different lattice formulations used during key recovery phase of the attacks. Interestingly, the discovered vulnerabilities are known to the research community and previously reported across multiple vendors (e.g., OpenSSL), which highlights a gap in the practice of CVE coordination among peer vendors.

Cui bono? NSS is certainly neither the first nor last security library to fall prey to SCA and failure to use constant-time implementations. Why is this a recurring event? Who should be held culpable? We note the break, fix, break cycle benefits several stakeholders due to perverse incentives—to mention a few: (i) it keeps software engineers in demand since these libraries are not "deploy and forget"; (ii) it keeps security engineers in demand since there is a steady stream of security issues to assess and address; (iii) it keeps security researchers busy with a perpetual flow of research topics to write papers about—including us. During judgment, the ancient Romans inquired *Cui bono?* or "Who benefits?" to identify suspects. Perhaps the incomplete list of key players above in this self-perpetuating meta-system is a good start.

Mitigations. During responsible disclosure to Mozilla, we made several FOSS contributions to assist in mitigating these issues and testing the fixes—all of which are now merged (further details in Appendix B).

ACKNOWLEDGMENTS

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

The first author was supported in part by the Tuula and Yrjö Neuvo Fund through the Industrial Research Fund at Tampere University of 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

[1] Onur Acıiçmez, 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, Vol. 4887)*, Steven D. Galbraith (Ed.). Springer, 185–203. https://doi.org/10.1007/978-3-540-77272-9_12

[2] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. 2019. Port Contention for Fun and Profit. In

2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019. IEEE, 870–887. https://doi.org/10.1109/SP.2019.00066

[3] 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

[4] Alejandro Cabrera Aldaya, Raudel Cuiman Márquez, Alejandro J. Cabrera Sarmiento, and Santiago Sánchez-Solano. 2017. Side-channel analysis of the modular inversion step in the RSA key generation algorithm. *I. J. Circuit Theory and Applications* 45, 2 (2017), 199–213. https://doi.org/10.1002/cta.2283

[5] Alejandro Cabrera Aldaya, Cesar Pereida García, Luis Manuel Alvarez Tapia, and Billy Bob Brumley. 2019. Cache-Timing Attacks on RSA Key Generation. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019, 4 (2019), 213–242. https://doi.org/10.13154/tches.v2019.i4.213-242

[6] 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

[7] Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. 2015. Lucky 13 Strikes Back. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, AsiaCCS 2015, Singapore, April 14-17, 2015*, Feng Bao, Steven Miller, Jianying Zhou, and Gail-Joon Ahn (Eds.). ACM, 85–96. https://doi.org/10.1145/2714576.2714625

[8] Pierre Belgarric, Pierre-Alain Fouque, Gilles Macario-Rat, and Mehdi Tibouchi. 2016. Side-Channel Analysis of Weierstrass and Koblitz Curve ECDSA on Android Smartphones. 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, Vol. 9610)*, Kazue Sako (Ed.). Springer, 236–252. https://doi.org/10.1007/978-3-319-29485-8_14

[9] Dmitry Belyavsky, Billy Bob Brumley, Jesús-Javier Chi-Domínguez, Luis Rivera-Zamarripa, and Igor Ustinov. 2020. Set It and Forget It! Turnkey ECC for Instant Integration. *CoRR* abs/2007.11481 (2020). https://arxiv.org/abs/2007.11481

[10] Naomi Benger, 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, Vol. 8731)*, Lejla Batina and Matthew Robshaw (Eds.). Springer, 75–92. https://doi.org/10.1007/978-3-662-44709-3_5

[11] Daniel J. Bernstein and Bo-Yin Yang. 2019. Fast constant-time gcd computation and modular inversion. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019, 3 (2019), 340–398. https://doi.org/10.13154/tches.v2019.i3.340-398

[12] 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, Vol. 5912)*, Mitsuru Matsui (Ed.). Springer, 667–684. https://doi.org/10.1007/978-3-642-10366-7_39

[13] 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, Vol. 6879)*, Vijay Atluri and Claudia Díaz (Eds.). Springer, 355–371. https://doi.org/10.1007/978-3-642-23822-2_20

[14] Ken A.L. Coar and David Robinson. 2004. *The Common Gateway Interface (CGI) Version 1.1.* RFC 3875. RFC Editor. 1–36 pages. https://doi.org/10.17487/RFC3875

[15] Don Coppersmith. 1996. 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 (Lecture Notes in Computer Science, Vol. 1070)*, Ueli M. Maurer (Ed.). Springer, 155–165. https://doi.org/10.1007/3-540-68339-9_14

[16] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 2016, 86 (2016). http://eprint.iacr.org/2016/086

[17] The FPLLL development team. 2016. fplll, a lattice reduction library. (2016). https://github.com/fplll/fplll

[18] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. 2015. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. *ACM Trans. Inf. Syst. Secur.* 18, 1 (2015), 4:1–4:32. https://doi.org/10.1145/2756550

[19] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. 2019. Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 1202–1219. https://doi.org/10.1109/SP.2019.00005

[20] Forum of Incident Response and Security Teams, Inc. 2017. *Guidelines and Practices for Multi-Party Vulnerability Coordination and Disclosure.* Forum of Incident Response and Security Teams, Inc. https://www.first.org/global/sigs/vulnerability-coordination/multiparty/FIRST-Multiparty-Vulnerability-Coordination-latest.pdf

[21] Nicolas Gama, Phong Q. Nguyen, and Oded Regev. 2010. Lattice Enumeration Using Extreme Pruning. In *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Monaco / French Riviera, May 30 - June 3, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6110)*, Henri Gilbert (Ed.). Springer, 257–278. https://doi.org/10.1007/978-3-642-13190-5_13

[22] Cesar Pereida García, Sohaib ul Hassan, Nicola Tuveri, Iaroslav Gridin, Alejandro Cabrera Aldaya, and Billy Bob Brumley. 2020. Certified Side Channels. In *Proceedings of the 29th USENIX Security Symposium, Boston, MA, USA, August 12-14, 2020*. USENIX Association. https://arxiv.org/abs/1909.01785

[23] 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

[24] Iaroslav Gridin, Cesar Pereida García, Nicola Tuveri, and Billy Bob Brumley. 2019. 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 (Lecture Notes in Computer Science, Vol. 11543)*, Roberto Perdisci, Clémentine Maurice, Giorgio Giacinto, and Magnus Almgren (Eds.). Springer, 330–350. https://doi.org/10.1007/978-3-030-22038-9_16

[25] 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

[26] Vipul Gupta, Douglas Stebila, and Sheueling Chang Shantz. 2004. Integrating elliptic curve cryptography into the web's security infrastructure. In *Proceedings of the 13th international conference on World Wide Web - Alternate Track Papers & Posters, WWW 2004, New York, NY, USA, May 17-20, 2004*, Stuart I. Feldman, Mike Uretsky, Marc Najork, and Craig E. Wills (Eds.). ACM, 402–403. https://doi.org/10.1145/1013367.1013496

[27] Nadia Heninger and Hovav Shacham. 2009. 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 (Lecture Notes in Computer Science, Vol. 5677)*, Shai Halevi (Ed.). Springer, 1–17. https://doi.org/10.1007/978-3-642-03356-8_1

[28] Jan Jancar, Vladimir Sedlacek, Petr Svenda, and Marek Sýs. 2020. Minerva: The curse of ECDSA nonces. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020, 4 (2020). https://eprint.iacr.org/2020/728

[29] 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, Vol. 1666)*, Michael J. Wiener (Ed.). Springer, 388–397. https://doi.org/10.1007/3-540-48405-1_25

[30] Noboru Kunihiro. 2018. Mathematical Approach for Recovering Secret Key from Its Noisy Version. In *Mathematical Modelling for Next-Generation Cryptography*. Math. Ind. (Tokyo), Vol. 29. Springer, Singapore, 199–217. https://doi.org/10.1007/978-981-10-5065-7

[31] Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, and Nadia Heninger. 2020. TPM-FAIL: TPM meets Timing and Lattice Attacks. In *Proceedings of the 29th USENIX Security Symposium, Boston, MA, USA, August 12-14, 2020*. USENIX Association. https://www.usenix.org/conference/usenixsecurity20/presentation/moghimi

[32] Kathleen Moriarty, Burt Kaliski, Jakob Jonsson, and Andreas Rusch. 2016. *PKCS #1: RSA Cryptography Specifications Version 2.2.* RFC 8017. RFC Editor. 1–78 pages. https://doi.org/10.17487/RFC8017

[33] Yutaka Oiwa, Kazukuni Kobara, and Hajime Watanabe. 2007. A New Variant for an Attack Against RSA Signature Verification Using Parameter Field. In *Public Key Infrastructure, 4th European PKI Workshop: Theory and Practice, EuroPKI 2007, Palma de Mallorca, Spain, June 28-30, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4582)*, Javier López, Pierangela Samarati, and Josep L. Ferrer (Eds.). Springer, 143–153. https://doi.org/10.1007/978-3-540-73408-6_10

[34] 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

[35] Cesar Pereida García and Billy Bob Brumley. 2017. Constant-Time Callees 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

[36] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. 2016. "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*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM, 1639–1650. https:

//doi.org/10.1145/2976749.2978420
[37] Eyal Ronen, Robert Gillham, Daniel Genkin, Adi Shamir, David Wong, and Yuval Yarom. 2019. The 9 Lives of Bleichenbacher's CAT: New Cache ATtacks on TLS Implementations. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 435–452. https://doi.org/10.1109/SP.2019.00062
[38] Keegan Ryan. 2019. Return of the Hidden Number Problem: A Widespread and Novel Key Extraction Attack on ECDSA and DSA. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019, 1 (2019), 146–168. https://doi.org/10.13154/tches.v2019.i1.146-168
[39] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. 2016. Preventing Page Faults from Telling Your Secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016*, Xiaofeng Chen, XiaoFeng Wang, and Xinyi Huang (Eds.). ACM, 317–328. https://doi.org/10.1145/2897845.2897885
[40] Josef Stein. 1967. Computational problems associated with Racah algebra. *J. Comput. Phys.* 1, 3 (1967), 397–405. https://doi.org/10.1016/0021-9991(67)90047-2
[41] Chia-che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, Dilma Da Silva and Bryan Ford (Eds.). USENIX Association, 645–658. https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai
[42] Nicola Tuveri and Billy Bob Brumley. 2019. Start Your ENGINEs: Dynamically Loadable Contemporary Crypto. In *2019 IEEE Cybersecurity Development, SecDev 2019, Tysons Corner, VA, USA, September 23-25, 2019*. IEEE, 4–19. https://doi.org/10.1109/SecDev.2019.00014
[43] 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 *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. ACM, 147–160. https://doi.org/10.1145/3274694.3274725
[44] Sohaib ul Hassan, Iaroslav Gridin, Ignacio M. Delgado-Lozano, Cesar Pereida García, Jesús-Javier Chi-Domínguez, Alejandro Cabrera Aldaya, and Billy Bob Brumley. 2020. *CVE-2020-12399: research data and tooling*. Zenodo. https://doi.org/10.5281/zenodo.3982270
[45] Luke Valenta, David Adrian, Antonio Sanso, Shaanan Cohney, Joshua Fried, Marcella Hastings, J. Alex Halderman, and Nadia Heninger. 2017. Measuring small subgroup attacks against Diffie-Hellman. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/measuring-small-subgroup-attacks-against-diffie-hellman/
[46] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2017. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution, SysTEX@SOSP 2017, Shanghai, China, October 28, 2017*. ACM, 1–6. https://doi.org/10.1145/3152701.3152706
[47] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, Engin Kirda and Thomas Ristenpart (Eds.). USENIX Association, 1041–1056. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/van-bulck
[48] 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, Vol. 9048)*, Kaisa Nyberg (Ed.). Springer, 3–21. https://doi.org/10.1007/978-3-319-16715-2_1
[49] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. 2017. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. 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, 2421–2434. https://doi.org/10.1145/3133956.3134038
[50] Samuel Weiser, David Schrammel, Lukas Bodner, and Raphael Spreitzer. 2020. Big Numbers - Big Troubles: Systematically Analyzing Nonce Leakage in (EC)DSA Implementations. In *Proceedings of the 29th USENIX Security Symposium, Boston, MA, USA, August 12-14, 2020*. USENIX Association. https://www.usenix.org/conference/usenixsecurity20/presentation/weiser
[51] 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
[52] Samuel Weiser, Andreas Zankl, Raphael Spreitzer, Katja Miller, Stefan Mangard, and Georg Sigl. 2018. 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*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 603–620. https://www.usenix.org/conference/usenixsecurity18/presentation/weiser
[53] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. 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, 640–656. https://doi.org/10.1109/SP.2015.45
[54] Yuval Yarom, Daniel Genkin, and Nadia Heninger. 2016. 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 (Lecture Notes in Computer Science, Vol. 9813)*, Benedikt Gierlichs and Axel Y. Poschmann (Eds.). Springer, 346–367. https://doi.org/10.1007/978-3-662-53140-2_17
[55] Robert Zuccherato, Patrick Cain, Carlisle Adams, and Denis Pinkas. 2001. *Internet X.509 Public Key Infrastructure Time-Stamp Protocol (TSP)*. RFC 3161. RFC Editor. 1–26 pages. https://doi.org/10.17487/RFC3161

A PEER VENDORS AND SECURITY DISCLOSURE

Multi-vendor security disclosure is a practical challenge with peer vendor competing products. Quoting [20, p. 20]: "Missing or poor communication between peer vendors can negatively impact coordination efforts." The document goes on to give two illustrative examples of security failures due to lack of peer vendor communication, which we extend with a third.

HTTP proxy poisoning. First publicly disclosed by R. L. Schwartz in 2001, by setting the (undefined) `Proxy` header in an HTTP request to a malicious URL, per RFC 3875 [14] server-side CGI scripts will translate this to the `HTTP_PROXY` environmental variable. This is unfortunately a common environmental variable used by subsequent server-side scripts to configure outgoing proxy connections, redirecting traffic to the malicious URL. Rediscovered several times since, in 2016 "httpoxy" by D. Scheirlinck et al. led to over 14 CVE assignments across different vendors[11].

DNS cache poisoning. In 1999, D. J. Bernstein implemented UDP source port randomization to harden dnscache (part of djbdns) against DNS spoofing attacks[12]. Around that time, Bernstein made a very clear and public argument that transaction ID randomization was insufficient. In 2008, D. Kaminsky developed an exploit around the concept[13], resulting in CVE-2008-1447 that affected several widely-deployed DNS solutions at the time, such as BIND and Windows DNS.

Bit lengths can be secrets, too. In 2011, Brumley and Tuveri [13] described a timing vulnerability present in OpenSSL's implementation of ECDSA that used Montgomery's ladder as a scalar multiplication algorithm for binary curves, exploited to steal the secret key of a remote TLS server. The general message from the work was that in nonce-based digital signature schemes, the effective bit length of the nonce must also be kept secret. OpenSSL responded by issuing CVE-2011-1945 and peer vendor Mozilla ported the patch that landed in OpenSSL to NSS[14]. In 2019, "Minerva" by Jancar et al. [28] revealed the vulnerability persists in many modern implementations leading to at least seven CVE assignments across vendors, and similarly the recent "TPM-FAIL" by Moghimi et al.

[11] https://httpoxy.org/
[12] https://cr.yp.to/djbdns/forgery.html
[13] https://dankaminsky.com/2008/07/09/an-astonishing-collaboration/
[14] https://hg.mozilla.org/projects/nss/rev/079cfc4710c7193ef73888394f4d4f935e03f241

[31] with more of a hardware focus and CVE assignments by Intel (CVE-2019-11090) and STMicroelectronics (CVE-2019-16863).

The three examples above are an illustrative range of porting attack concepts to related products that implement similar standards. The first being closer to the traditional software security side, the second more of an algorithmic attack, and the third (strongly motivating our work) a side-channel attack. At a high level, the root cause of all three examples is vendors failing to follow the advice of security professionals and being blind to peer vendor activity regarding security hardening.

B MITIGATIONS

CVE-2020-12399. To solve the vulnerability in Section 4, we proposed a patch[15] to NSS that randomizes the nonce by $\hat{k} = k + b \cdot q$ where b is a random word with a fixed bit length, i.e., top bit set. Our empirical evaluation of the patch indicates this aligns the curves in Figure 1, mitigating the issue.

CVE-2020-12402. For the vulnerability in Section 8, we implemented[16] the constant-time GCD and modular inversion by Bernstein and Yang [11].

CVE-2020-6829. For the Section 6 and Section 7 vulnerabilities, we decided against patching wNAF and, similar to the `secp256r1` code in NSS, proposed two custom `ECGroupStr` for `secp384r1`[17] and `secp521r1`[18]. We leveraged ECCKiila[19] for this task [9], built on top of `fiat-crypto`[20] to take advantage of its formally verified and constant-time GF layer [19].

CVE-2020-12401. With these constant-time versions in place and NSS not featuring any other vulnerable curves, the broken fix in Section 5 is no longer needed—hence we submitted a patch to remove the padding[21].

C REFERENCE ALGORITHMS

Algorithm 1: Compute wNAF representation of k

Input: Integer k and width w
Output: $wNAF(k, w)$

$i = 0$
while $k \neq 0$ **do**
  if $\mathsf{odd}(k)$ **then**
    $d = k \bmod 2^w$
    if $d > 2^{w-1}$ **then** $d = d - 2^w$
    $k = k - d$
  else $d = 0$
  $wNAF[i] = d, k = k/2, i = i + 1$
return $wNAF$

Algorithm 2: wNAF-based scalar multiplication

Input: Integer k, width w and elliptic curve point G
Output: $R = kG$

Compute $wNAF(k, w)$ using Algorithm 1
$P[i] = iG\ ; i \in \{-2^{w-1} + 1, \ldots, -3, -1, 1, \ldots, 2^{w-1} - 1\}$
$R = O$
for $i = \lfloor \lg(k) \rfloor + 1$ **downto** 0 **do**
  $R = 2R$
  if $wNAF[i] \neq 0$ **then** $R = R + P[wNAF[i]]$
return R

Algorithm 3: Binary extended Euclidean algorithm (BEEA)

Input: Integers a and b such that $0 < a < b$
Output: Greatest common divisor of a and b

begin
  $u \leftarrow a, v \leftarrow b, i \leftarrow 0$
  while $\mathsf{even}(u)$ **and** $\mathsf{even}(v)$ **do**
    $u \leftarrow u/2, v \leftarrow v/2, i \leftarrow i + 1$
  while $u \neq 0$ **do**
    while $\mathsf{even}(u)$ **do** $u \leftarrow u/2$
    while $\mathsf{even}(v)$ **do** $v \leftarrow v/2$
    if $u \geq v$ **then** $u \leftarrow u - v$
    else $v \leftarrow v - u$
  return $v \cdot 2^i$

[15] https://hg.mozilla.org/projects/nss/rev/daa823a4a29bcef0fec33a379ec83857429aea2e
[16] https://hg.mozilla.org/projects/nss/rev/699541a7793bbe9b20f1d73dc49e25c6054aa4c1
[17] https://hg.mozilla.org/projects/nss/rev/d19a3cd451bbf9602672fdbba8d6a817a55bfc69
[18] https://hg.mozilla.org/projects/nss/rev/ca068f5b5c176c503ddce969e78dd326cc5fd29a
[19] https://gitlab.com/nisec/ecckiila
[20] https://github.com/mit-plv/fiat-crypto
[21] https://hg.mozilla.org/projects/nss/rev/aeb2e583ee957a699d949009c7ba37af76515c20

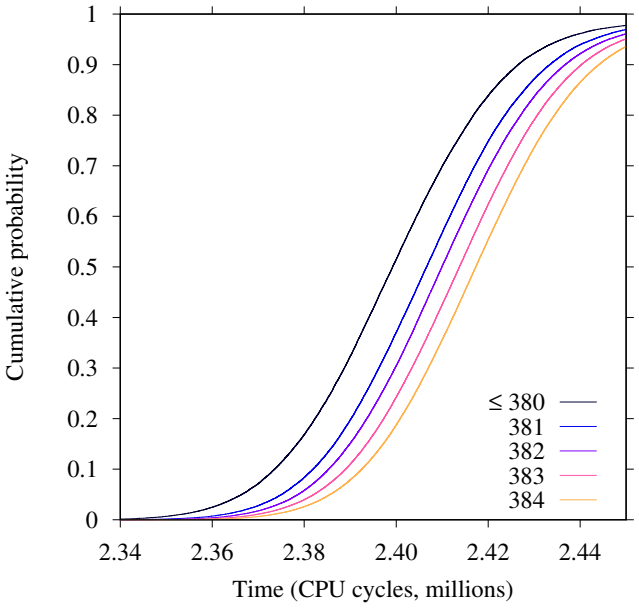

Figure 4: ECDSA timing cdf per nonce bit-length.

```
    i = 0;
    /* Compute wNAF form */
    while (mp_cmp_z(&k) > 0) {
        if (mp_isodd(&k)) {
            out[i] = MP_DIGIT(&k, 0) & mask;
            if (out[i] >= twowm1)
                out[i] -= 2 * twowm1;

            /* Subtract off out[i].  Note
             ↪  mp_sub_d only works with
              * unsigned digits */
            if (out[i] >= 0) {
                mp_sub_d(&k, out[i], &k);
            } else {
                mp_add_d(&k, -(out[i]), &k);
            }
        } else {
            out[i] = 0;
        }
        mp_div_2(&k, &k);
        i++;
    }
```

Figure 5: NSS wNAF encoding function snippet.

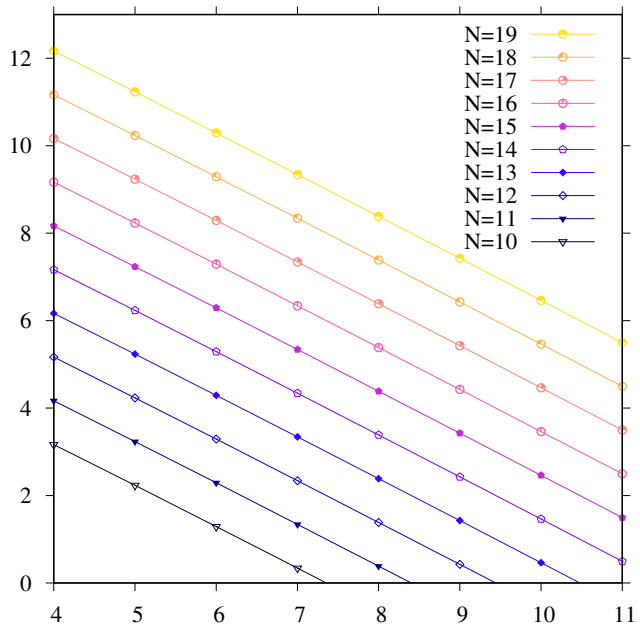

Figure 6: The x-axis corresponds with the number ℓ clear MSBs used, while the y-axis determines the quantity $\theta - \ell$ where $\theta = \lg(N) + \lg(\ell) - \lg(m) - \lg(1.25 \cdot \delta)$ is the expected number of clear MSBs in the filtered sample. Sample sizes are log scale.