

Iaroslav Gridin

SOFTWARE FOR ADVANCED EXECUTION PATH INSPECTION

Master of Science Thesis
Information Technology and Communication Sciences
Examiners: Billy Bob Brumley, Tampere University
Cesar Pereida García, Tampere University
August 2020

ABSTRACT

Iaroslav Gridin: Software for Advanced Execution Path Inspection
Master of Science Thesis
Tampere University
Information Technology, MSc
August 2020

Execution path is a subset of code that gets executed during operation of software. Inspection of the execution path is often required when analysing software for vulnerabilities. This thesis describes Triggerflow, a tool for tracking execution paths, that can be used to facilitate such inspection. Triggerflow works by leveraging debugger to dynamically analyze code execution and filtering results using source code annotations. The thesis describes the tool interface, engineering choices made during its development, techniques it uses, and supporting software and methodology of deploying continuous integration using this software. Triggerflow was originally developed for detecting side-channel vulnerabilities in OpenSSL. The work on Triggerflow led to a conference publication at DIMVA 2019, main author being the author of this thesis. The conference paper is included as appendix A.

Keywords: software testing, regression testing, continuous integration, dynamic program analysis

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

CONTENTS

1	Introduction	2
1.1	Motivation	2
1.2	Goals	2
1.3	Contributions	2
1.4	Structure	3
2	Background	4
2.1	Side Channel Attacks	4
2.2	Challenges in Defense Against Side Channel Attacks	4
2.3	Verifying Constant-time Functions	4
2.4	OpenSSL	5
2.5	Constant-time Functions in OpenSSL	5
2.6	Attacks on BN_FLG_CONSTTIME	5
2.7	Inspecting OpenSSL With Triggerflow	6
3	Methodology	7
3.1	Tool Demand	7
3.2	Related Work	7
3.2.1	Dynamic Analysis	7
3.3	Programming Language	8
3.4	Code Analysis Strategies	8
3.5	Debugger	9
3.6	Unit Testing	9
4	Software description	10
4.1	Triggerflow At A Glance	10
4.2	GDB	10
4.3	Annotations	10
4.4	Triggers	12
4.5	Output	12
5	Application	14
5.1	Auditing OpenSSL For Side-Channel Vulnerabilities	14
5.2	Coupling With Statistical Analysis	14
5.3	Automatic Testing	15
6	Conclusion	17
6.1	Impact	17

6.2 Future Work	18
References	20
Appendix A Triggerflow paper	24

LIST OF FIGURES

4.1	Example of Triggerflow output in graph form	13
5.1	Output of performing vulnerable OpenSSL operation under Triggerflow, before filtering	15
5.2	Output of performing vulnerable OpenSSL operation under Triggerflow, after filtering	15

LIST OF SYMBOLS AND ABBREVIATIONS

CI	Continuous Integration
CPU	Central Processing Unit
GDB	GNU Debugger
GDB/MI	GNU Debugger Machine Interface
SCA	Side Channel Attack

PREFACE

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).

Thanks

- My supervisor Dr. Billy Bob Brumley for the guidance and feedback throughout, as well as inspiration for the thesis.
- My examiner Cesar Pereida García, for examination and feedback on the thesis.
- And everybody else in the NISEC group for collaboration and their expertise.
- Sofia Startceva, for the support.

1 INTRODUCTION

1.1 Motivation

Researchers investigating cryptographic software for side-channel vulnerabilities often know the vulnerable operations, for example, from another library, or from research done on the operation itself, but do not know if these operations are being run on sensitive data. The problem is to detect certain areas of code, known to be vulnerable, being executed as a result of operation on a specific value, known to be sensitive. Thus the researchers need a tool, which, given a code area, allows the user to trace its invocations and filter only those that involve a sensitive input. While there are many options for this developed for similar purposes, examples given in chapter 3, none provided advanced filtering features that Triggerflow implements.

1.2 Goals

The goals of the thesis are:

- To summarize the prior research in detecting side-channel vulnerabilities, describing tools already made for similar tasks and history of vulnerabilities they need to detect.
- To demonstrate the demand for better software that can facilitate the search for vulnerabilities.
- To produce the new tool, capable for advanced execution path inspection and suitable for automatic continuous testing of cryptographic software.
- To describe an example of the automated continuous testing process for finding side-channel vulnerabilities in cryptographic software using the tool.

1.3 Contributions

The contributions of the thesis are

- The Triggerflow software, open source, allowing researchers to use it to find side-channel vulnerabilities.

- Review of the prior research.
- The methodology to organize continuous automated testing of cryptographic libraries.

1.4 Structure

The chapter 2 presents the demand for Triggerflow and situation it needs to help. The chapter 3 contains the approach taken during development of Triggerflow and reasoning behind engineering choices. The chapter 4 describes Triggerflow interface and features. Finally, chapter 5 suggests some patterns of Triggerflow use and describes an example of automatic testing setup using Triggerflow and GitLab. In Appendix A, the original conference paper describing the Triggerflow is included.

2 BACKGROUND

This chapter presents the demand for Triggerflow and situation it needs to help.

2.1 Side Channel Attacks

Side Channel Attack, or SCA, in the context of cryptographic software, is an attack that utilizes information leaked during code execution [1]. Successful SCAs on cryptographic systems can lead to leaking data that must be kept secret from third parties but has to be involved in an operation. Timing attack is a type of a side-channel attack based on observing time taken by a program and deriving characteristics of a secret value based on observations [2]. Timing attacks are becoming more and more common, in part, due to popularity of web-based applications, which are fundamentally vulnerable to timing attacks due to their architecture [3].

2.2 Challenges in Defense Against Side Channel Attacks

To protect against SCAs, when working with secret values, software engineers make software perform **in constant time**, for example, by padding or batching computations, so that operation timing does not depend on properties of private values. While it is often reasonably easy to make such computations in constant time, such measures usually lower the software performance [4], so cryptographic libraries typically don't use these special versions of routines when operating on public values. The challenge here is to make sure the choice of constant-time/non-constant-time operation is made correctly in all cases, otherwise either security or speed is sacrificed.

2.3 Verifying Constant-time Functions

Once code that does not run in constant time is located, either by locating well-known non-constant-time operations manually, or by tools like DATA [5], the researcher needs to track if this code is being executed with sensitive input, thus creating a SCA vulnerability.

Detection can be performed simply by using GDB (see chapter 3 for information on it and similar tools) to stop execution and inform the user in vulnerable code points. However, in

practice this straightforward approach is not enough, since during a cryptographic operation multiple calls to the vulnerable code might be made, and this would lead to multiple false positives. Advanced filtering can solve this issue, for example excluding completely the code path that only happens with non-sensitive data (like a function call with only the public key as a parameter). Sometimes it is necessary to perform even more complicated filtering to exclude false positives, excluding a code path if it includes several code points at once.

These requirements led to development of Triggerflow, the tool to perform advanced execution path inspection.

2.4 OpenSSL

OpenSSL is a cryptographic library written in C, providing implementations of popular cryptographic protocol TLS as well as numerous cryptographic primitives. OpenSSL is very popular, to the point of its many [6] vulnerabilities, like in case of HeartBleed, causing wide efforts to limit the impact. For example, the widespread adoption of OpenSSL warranted using machine learning to detect vulnerable installations [7]. Thus, problems in OpenSSL have great impact on information networks, and the software is a prime target for security analysis.

2.5 Constant-time Functions in OpenSSL

BN_FLG_CONSTTIME is a Boolean property set on big numbers in OpenSSL code, and functions decide to run constant time or high performance code based on BN_FLG_CONSTTIME presence. By default, BN_FLG_CONSTTIME is set to false, and operations are performed not in constant time. The OpenSSL developers chose this approach because it requires minimal code changes and prioritizes performance, but it poses a security risk since all private values must be explicitly marked. BN_FLG_CONSTTIME was introduced in 2005 as a result of RSA cache-timing attack [8].

2.6 Attacks on BN_FLG_CONSTTIME

However, BN_FLG_CONSTTIME turned out to be vulnerable to mishandling and to have bugs of its own. In 2007, Aciicmez et al. [9] described several side-channel vulnerabilities in OpenSSL and suggested modifications to algorithms that eliminated the leaks. The vulnerabilities included calls to modular inversion function, which timing is known to be sensitive to the input.

In 2016 Pereida García et al. [10] have discovered that mechanism itself does not work

properly: if a number marked with `BN_FLG_CONSTTIME` is copied, the flag is lost and the result is handled as public. More issues with flag being mishandled were discovered over time [11] [12], with comprehensive overview available in section 2.2 of the Appendix A.

2.7 Inspecting OpenSSL With Triggerflow

To detect such issues, Gridin et al. [13] have added annotations to OpenSSL that mark the vulnerable areas and filter out the known false negatives. This method led to discovery of more OpenSSL issues over time, the process is documented in more detail in the paper attached to the thesis as Appendix A.

3 METHODOLOGY

This chapter contains the approach taken during development of Triggerflow and reasoning behind engineering choices.

3.1 Tool Demand

Analysing the OpenSSL code for side-channel vulnerabilities required a tool that can produce a call graph, filter it, present the results to user and output it in machine-readable form.

First, the tool needs to produce a **call graph**: a graph of function calls, that leads to code sections known to be vulnerable. The graph is required for further processing, and by itself contains useful information about the execution flow.

Second, the tool needs to filter the call graph to remove inputs that are known to be not private. This serves to remove the false positives and has been described above in chapter 1.

Third, the tool needs to present the results in visual form. The user needs to be able to quickly assess the output and share the data further in easily-understandable graph.

Finally, the tool needs to output machine-readable results, to serve either as input to further processing, or as part of an automatic testing setup, as shown in chapter 5.

3.2 Related Work

Numerous solutions for similar problems exist, but none quite combined the formulated requirements. Listed next are the prominent existing tools.

3.2.1 Dynamic Analysis

Callgrind [14] is an addon to Valgrind [15], debugging and profiling software suite, that records function call history. It allows to see call graph, but does not support filtering the code paths to exclude uninteresting data, so it is not useful for our purposes.

Gprof [16] is a tool that determines the amount of CPU time spent processing different

functions. Similar to Callgrind, it allows to output the call graph, but lacks any sophisticated filtering features.

DATA [17] is a framework for detecting side-channel leaks. Side-channel leak detection is the original application of Triggerflow, but DATA occupies a different niche: it is a statistical analysis tool, which discovers the leakage-prone code areas, then Triggerflow can be applied to trace all code paths that lead to the areas. Additionally, while DATA produces extensive information about the codebase, statistical analysis requires a lot of resources, so application of DATA to automatic monitoring is unfeasible.

3.3 Programming Language

Initially, the set of scripts to detect known side-channel vulnerabilities was written in Bash [18]. While portable and easy to write, Bash scripts are hard to maintain and extend. Project was small, but not clearly defined and scope was likely to change, so Ruby [19] was chosen as a programming language. Ruby is a weakly typed, dynamic programming language with advanced object model. Roman et al. in [20] reports that Ruby features a very relaxed and English-like syntax. These characteristics make development of small scale projects easy, but may lead to problems with development of complex software.

3.4 Code Analysis Strategies

There are two main types of code analysis: dynamic and static. Both approaches have different pros, cons and usage cases.

Static analysis works with binary or source code, and attempts to determine potential execution flows without executing the code. Since basic static analysis is generally easy to set up and effective, as many as 30 percent of open source projects [21] use it to detect code problems. However, static analysis often suffers from over-estimation, producing many false positives [17], making it unsuitable for fully automatic testing.

In contrast, dynamic analysis runs the code and analyzes the flow. This way, the analyzing software doesn't have to interpret the source code by itself, and works with existing tools to observe. With diverse inputs there can be a high probability of encountering information leaks [17], thus dynamic analysis can be relied to produce no false positives and accurate results in automated way.

Triggerflow uses dynamic analysis because of this ease of application, and its general sufficiency for Triggerflow targets.

3.5 Debugger

GDB [22] is the most popular and universal software debugging tool. It works under various UNIX systems, Mac OS X and Windows. GDB supports a variety of programming languages, with focus on C. GDB includes GDB/MI, machine-readable interface, which is instrumental to implementing Triggerflow. These characteristics are basically unique to GDB, so it was chosen to inspect the execution path. GDB supports **breakpoints**, locations in code in which execution should stop and context become available for inspection. Breakpoints can be **conditional**: only triggering when certain expression evaluates to true. Breakpoints are the main feature of GDB used by Triggerflow.

3.6 Unit Testing

The project uses sharness [23] for unit testing. Sharness is a small shell library which runs shell-based tests. Since Triggerflow is a single application with easily-parseable output, Sharness is perfect for testing it. While full code coverage is not required due to small project scale, unit tests are present to ensure basic workflows produce correct results. Triggerflow test suite includes testcases for all general use cases (all kinds of annotations and their combinations, patching), as well as some corner cases to ensure specific features work (segmentation fault in the program, program not terminating normally).

4 SOFTWARE DESCRIPTION

This chapter describes Triggerflow interface and features.

4.1 Triggerflow At A Glance

1. The inputs to Triggerflow are: a directory with annotated source code, instructions to build it, commands to run and debug, and optionally patches to apply before building.
2. Triggerflow scans the source code for special keywords, which are typically placed in comments near related lines of code, and builds a database of annotations. Annotations include **points of interest**, or POIs, marking parts of code that should be reported if executed, and ignore annotations, which cause certain points of interest to be ignored.
3. Triggerflow commences the build, then runs the given commands (**triggers**) under the debugger (GDB), instructed to set breakpoints at all points of interest.
4. When GDB reports hitting a breakpoint, Triggerflow inspects the backtrace supplied by GDB, makes decisions based on the backtrace and stored annotations, and possibly logs the code path that led to it.

4.2 GDB

GDB includes GDB/MI, a machine interface for interacting with running GDB instance. While full GDB library for Ruby did not exist at the moment, there already was a GDB/MI parser [24] available. The parser, available under MIT license, was incorporated into `gdb-ruby` library, which supports GDB functions required by Triggerflow. The library has been released along with other Triggerflow code, also under MIT license.

4.3 Annotations

GDB allows breakpoints to be set by a line number in source code. While line numbers of points of interest can be stored and used to batch-set breakpoints, if code is modified (by the researcher or the upstream source), these numbers will have to be changed. To

address this, Triggerflow uses **annotations**: special words in code, used by Triggerflow as anchors.

```

1  /* code before */
2  if(a % 2 == 0) // TRIGGERFLOW_POI
3  /* code after */

```

The annotation above means this line is a point of interest: if execution hits that code, path that leads to it is logged.

```

Breakpoint hit at math_function() test.c:9
    #0 math_function() test.c:9
    #1 crypto_function() test.c:17
    #2 main() test.c:23

```

Sometimes it is necessary to ignore certain code paths which are known to produce false positives (i.e. they call cryptographic functions but only handle public data). Triggerflow uses annotation `TRIGGERFLOW_IGNORE` for that.

```

1  /* code before */
2  sensitive_op(public_key); // TRIGGERFLOW_IGNORE
3  /* code after */

```

In this example, all code paths that that particular invocation of `sensitive_op()` triggers will be ignored.

If even more specific code paths need to be ignored, ignoring can be further restricted to group ignoring with `TRIGGERFLOW_IGNORE_GROUP`. Such annotations only ignore POI if all `TRIGGERFLOW_IGNORE_GROUP` annotations with that group name present in code are included in code path.

```

1  /* code before */
2  // conditionally sensitive operation
3  void c_s_op(public_key) {
4      /* code */
5      c_s_op2(public_key); // TRIGGERFLOW_IGNORE_GROUP S_OPS
6      /* code */
7  }
8  /* code */
9  c_s_op(public_key); // TRIGGERFLOW_IGNORE_GROUP S_OPS
10 /* code after */

```

In this example, only code paths that invoke `conditionally_sensitive_op2()` through this particular invocation of `conditionally_sensitive_op()` will be ignored.

If ignoring needs to depend on program state, `TRIGGERFLOW_IGNORE_IF` provides condi-

tional ignore that checks condition expressed in code expression in runtime, and ignores the path if the condition evaluates to true.

```

1  /* code before */
2  sensitive_op(key); // TRIGGERFLOW_IGNORE_IF key.is_private
3  /* code after */

```

In that case, when a code path invokes `sensitive_op()` in that line **and** `key.is_private` evaluates to true, path will be ignored.

Because annotations are dependant on the code structure beyond their line, it is best when annotations are maintained in the original codebase, and updated by the author of related changes. However, if the software being analyzed includes the system that protects against wrong codepaths, like OpenSSL's `BN_FLG_CONSTTIME` mentioned in chapter 2, then annotations are basically doing the same job, and would be useful only, for example, if they were maintained by separate teams, like in pair programming [25]. This way, two systems would verify each other. For the purposes of code analysis by a third party, Triggerflow also supports storing annotations separately, in form of patches that define annotation context. Context makes it usually trivial to adapt the patches to changed code, but manual inspection still has to be performed sometimes and complicated changes will require manual intervention.

4.4 Triggers

After building an internal database of annotations, Triggerflow builds the program being analyzed using user-supplied command from configuration file and starts running **triggers file**, a text file that contains commands that can possibly trigger the annotated code. If `debug` is prepended to a command, it is launched under GDB with breakpoints set according to annotations, and Triggerflow analyzes the output. Otherwise, it is interpreted as **maintenance command** and just runs in shell. These commands can be used to prepare the environment for launching triggers, or to clean up.

4.5 Output

Triggerflow supports simple console output, shown above, as well as outputting a diagram of observed execution paths. Graph is output in dot format [26], which can be visualized to popular formats like PDF with tools like `dot`. While console output is more compact and sometimes easier to read, graph form can visualize the structure and help to quickly grasp the call graph structure. A small test application example can be seen in Figure 4.1.

Graph output allows to visually quickly determine main flow directions, identify threat sources and false positives.

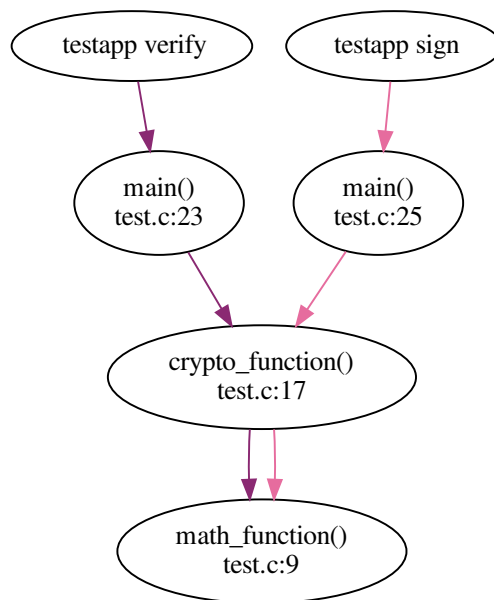


Figure 4.1. Example of Triggerflow output in graph form

5 APPLICATION

This chapter suggests some patterns of Triggerflow use and describes an example of automatic testing setup using Triggerflow and GitLab.

5.1 Auditing OpenSSL For Side-Channel Vulnerabilities

The section 3.1 of paper in Appendix A gives several detailed examples of detecting and monitoring OpenSSL Vulnerabilities with Triggerflow.

The current set of annotations started with known vulnerable areas in OpenSSL's implementation of big number division, exponentiation and finding a greatest common divisor, as well as elliptic curve multiplication. The POIs were filtered with ignore annotations in points that led to vulnerable areas, but only concerned public parts of initial data.

Over the time, the annotations were automatically adapted, as well as manually. Changes in OpenSSL internal APIs required researchers to edit the annotations in a way that could not be inferred by algorithm, and add new ignore points and move POIs.

Basically, the process to establish the set of annotations is as follows:

- Mark well-known vulnerable places with `TRIGGERFLOW_POI` .
- Runs operations using secret data under Triggerflow.
- Inspect the output. If there are paths known to be secure (e.g. the input is not secret, or it is transformed to a safe form in constant time before), mark them with `TRIGGERFLOW_IGNORE` or in complicated cases with `TRIGGERFLOW_IGNORE_GROUP`.
- Repeat the previous steps until only vulnerabilities are left.

5.2 Coupling With Statistical Analysis

Triggerflow can trace the execution of known vulnerable code, but finding the code is outside the scope. Thus, a researcher can apply a statistical analysis tool like DATA [17] to detect the vulnerable areas automatically and then use Triggerflow annotations to trace references to them.

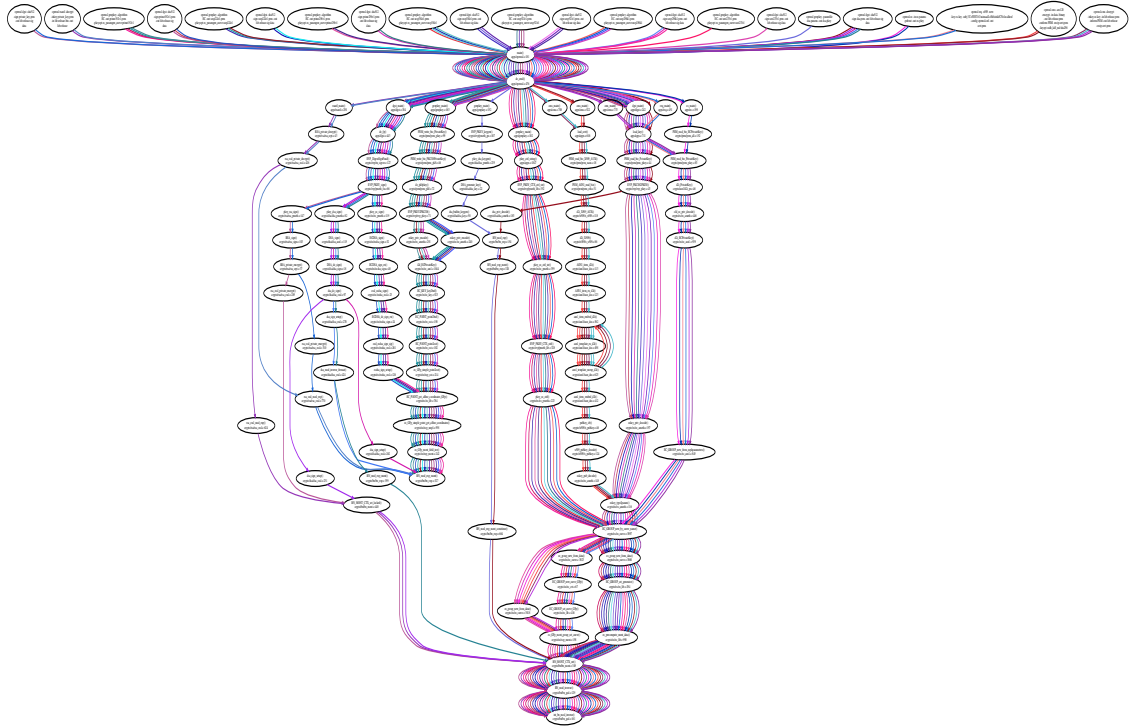


Figure 5.1. Output of performing vulnerable OpenSSL operation under Triggerflow, before filtering

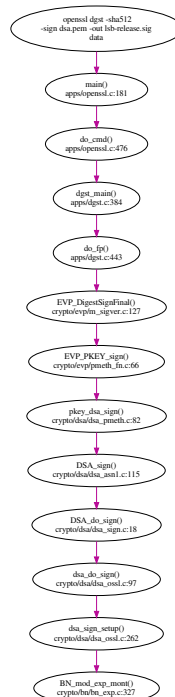


Figure 5.2. Output of performing vulnerable OpenSSL operation under Triggerflow, after filtering

5.3 Automatic Testing

Triggerflow code annotations are easy to maintain in updating codebase. If stored as **patches** [27], format for description of changes to text which includes context, annotations

can be automatically adapted to changed code. This lends itself to creating a system that can monitor code for known issues. Here we describe an automatic testing system built for monitoring OpenSSL code.

After determining sensitive areas, researcher marks them up with `TRIGGERFLOW_POI` annotations and annotates paths that lead to known false positives. Then, using quilt [28], they extract annotations to **patch files**, where line numbers are stored, along with context. Context is needed for automatic patch updates. Analyst also writes Triggerflow configuration file that describes steps to build software and possibly other options, and trigger file(s).

All these files, along with original code as git [29] **submodule** is added to git repository. Additionally, a CI description file (in our case, Gitlab's `.gitlab-ci.yml`) is added to define CI operations. CI system reads the repository, runs Triggerflow as described and, since no POIs are triggered, build is passed.

Separate system runs special program, **repatcher**. Repatcher regularly checks original code repository for updates. When a newer version of software is available, repatcher checks patches for compatibility and updates them if necessary. If patches cannot be automatically updated, repatcher pauses work and signals human operator to resolve situation. If patches apply cleanly, repatcher pushes code to repository handled by CI. CI runs Triggerflow on patched code. If any points of interest are hit, build is marked as broken.

This system relies on upstream software to build on every commit: if software fails to build, resulting build is also broken in CI and appears as false positive. However, it is generally accepted good practice to only commit code that builds and passes tests, since that assumption facilitates many kinds of code analysis.

One notable limitation is that since Triggerflow uses a debugger, under widely used Docker containerization tool [30] it needs extra permission `SYS_PTRACE` which allows to receive info about other host processes, and thus Triggerflow is not suitable for typical public automatic testing setups.

6 CONCLUSION

6.1 Impact

Timing attacks on cryptographic software, while well-known and largely mitigated, are still widespread. OpenSSL includes a non-perfect measure to ensure all sensitive values are handled in non-vulnerable fashion, but it is possible to verify its performance automatically using external tools. Gridin et al. developed such tool, called Triggerflow, first successfully applied in continuous testing of OpenSSL [13]. Triggerflow helped track down deficiencies in OpenSSL handling of sensitive values in asymmetric cryptography code and provided testing system to ensure security in the future. The authors established an automated testing system to monitor three major branches of OpenSSL for vulnerabilities.

Building on that, Pereida García et al. used Triggerflow to find several vulnerabilities in OpenSSL and MbedTLS [31]:

- Wrong handling of certain ECDSA certificates caused a timing vulnerability in signature code. Attackers managed to remotely recover the secret key, both via the network and by capturing EM traces. The vulnerability has been assigned CVE-2019-1547 [32].
- Vulnerable algorithm used when handling certain certificate formats endangered secret keys using RSA and DSA algorithms.
- Researchers found algorithm used for RSA operations in MbedTLS to be vulnerable, allowing for side-channel attack endangering services using MbedTLS and RSA certificates, such as the once issued by popular *Let's Encrypt* [33] organization.

In 2020 Hassan et al. used Triggerflow to analyze the cryptographic library NSS, used in Mozilla web browser. In that work, researchers combined Triggerflow with DATA [17] to be able to find the possible vulnerable places using DATA's statistical analysis and then trace their execution using Triggerflow. This new combined approach resulted in discovery of three new vulnerabilities:

- CVE-2020-12399 [34]: DSA signature code did not include the countermeasure used by other vendors, exposing secret key to the timing attacks. The vulnerability

was remotely exploited over network.

- CVE-2020-12402 [35]: Using vulnerable functions in RSA key generation has led to recovery of the secret key. The vulnerability was remotely exploited via EM traces.
- CVE-2020-6829 [36]: ECDSA nonce padding was found to be ineffective, leaking nonce size through timing attacks and facilitating nonce recovery. The vulnerability was remotely exploited via EM traces.

Additionally, Triggerflow testing setup discovered some non-security issues due to testing more wide and deep than OpenSSL testing suite:

- <https://github.com/openssl/openssl/issues/12102> breakage of a specific feature used by testing setup
- <https://github.com/openssl/openssl/issues/10114> regression not caught by OpenSSL testcases

These results demonstrate that Triggerflow is useful for finding side-channel vulnerabilities, especially when combined with statistic analysis to assist in locating potential vulnerable locations.

Triggerflow can be applied to any cryptographic library or any other program that would benefit from execution path inspection, in any language supported by GDB.

The obvious candidates for analysis might be other popular cryptographic libraries, like MbedTLS <https://github.com/ARMmbed/mbedtls>, NSS <https://wiki.mozilla.org/NSS>, BoringSSL <https://boringssl.googlesource.com/boringssl/>, or de-facto standards in other languages, like ring <https://github.com/briansmith/ring>.

6.2 Future Work

Since Triggerflow only performs non-negligible computations when building annotation database, its own performance of Triggerflow is generally not a bottleneck in practice. Nevertheless, it might benefit from reimplementation in a statically-typed programming language if used in a project with a lot of filters and points of interest. Strict typing, as well as tighter interface with GDB, might increase reliability of the program as well.

One other possible future direction is development of embedded Triggerflow version that can be compiled into the program being researched, thus allowing to avoid the requirement of a debugger described in chapter 5. Such a library could work by expanding annotations to function calls that monitor execution path from inside the program itself. However, the code would need to be written in every language, a considerable drawback versus the current language-agnostic leveraging of GDB.

The automatic testing setup, currently implemented with minimal footprint over the Git-Lab's CI framework, could be expanded with better interface and notification, enabling researchers to filter through problems, solve them in easier way and to monitor the past status of the testing system with relation to the state of software being monitored, for example, to be able to quickly see the analysis for a specific commit.

REFERENCES

- [1] Brumley, B. B. and Taveri, N. Remote Timing Attacks Are Still Practical. *Computer Security - ESORICS 2011 - 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12-14, 2011. Proceedings*. Ed. by V. Atluri and C. Díaz. Vol. 6879. Lecture Notes in Computer Science. Springer, 2011, 355–371. DOI: 10.1007/978-3-642-23822-2_20. URL: https://doi.org/10.1007/978-3-642-23822-2_20.
- [2] Brumley, D. and Boneh, D. Remote timing attacks are practical. *Computer Networks* 48.5 (2005), 701–716. DOI: 10.1016/j.comnet.2005.01.010. URL: <https://doi.org/10.1016/j.comnet.2005.01.010>.
- [3] Chen, S., Wang, R., Wang, X. and Zhang, K. Side-Channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow. *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berkeley/Oakland, California, USA*. IEEE Computer Society, 2010, 191–206. DOI: 10.1109/SP.2010.20. URL: <https://doi.org/10.1109/SP.2010.20>.
- [4] Bernstein, D. J. *Cache-timing attacks on AES*. Tech. rep. 2005, 13.
- [5] Hassan, S. ul, Gridin, I., Delgado-Lozano, I. M., Pereida García, C., Chi-Domínguez, J., Aldaya, A. C. and Brumley, B. B. Déjà Vu: Side-Channel Analysis of Mozilla's NSS. *CoRR abs/2008.06004* (2020). arXiv: 2008.06004. URL: <https://arxiv.org/abs/2008.06004>.
- [6] Meyer, C. and Schwenk, J. SoK: Lessons Learned from SSL/TLS Attacks. *Information Security Applications - 14th International Workshop, WISA 2013, Jeju Island, Korea, August 19-21, 2013, Revised Selected Papers*. Ed. by Y. Kim, H. Lee and A. Perrig. Vol. 8267. Lecture Notes in Computer Science. Springer, 2013, 189–209. DOI: 10.1007/978-3-319-05149-9_12. URL: https://doi.org/10.1007/978-3-319-05149-9_12.
- [7] Zhu, M., Hu, Z. and Liu, P. Reinforcement Learning Algorithms for Adaptive Cyber Defense against Heartbleed. *Proceedings of the First ACM Workshop on Moving Target Defense, MTD '14, Scottsdale, Arizona, USA, November 7, 2014*. Ed. by S. Jajodia and K. Sun. ACM, 2014, 51–58. DOI: 10.1145/2663474.2663481. URL: <https://doi.org/10.1145/2663474.2663481>.
- [8] Percival, C. Cache Missing for Fun and Profit. *BSDCan 2005, Ottawa, Canada, May 13-14, 2005, Proc.* 2005. URL: <http://www.daemonology.net/papers/cachemissing.pdf>.

- [9] Aciicmez, O., Gueron, S. and Seifert, J. New Branch Prediction Vulnerabilities in OpenSSL and Necessary Software Countermeasures. *Cryptography and Coding, 11th IMA International Conference, Cirencester, UK, December 18-20, 2007, Proceedings*. Ed. by S. D. Galbraith. Vol. 4887. Lecture Notes in Computer Science. Springer, 2007, 185–203. DOI: 10.1007/978-3-540-77272-9_12. URL: https://doi.org/10.1007/978-3-540-77272-9_12.
- [10] Pereida García, C., Brumley, B. B. and Yarom, Y. "Make Sure DSA Signing Exponentiations Really are Constant-Time". *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. Ed. by E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers and S. Halevi. ACM, 2016, 1639–1650. DOI: 10.1145/2976749.2978420. URL: <https://doi.org/10.1145/2976749.2978420>.
- [11] Pereida García, C. and Brumley, B. B. Constant-Time Callees with Variable-Time Callers. *IACR Cryptology ePrint Archive 2016 (2016)*, 1195. URL: <http://eprint.iacr.org/2016/1195>.
- [12] Weiser, S., Spreitzer, R. and Bodner, L. Single Trace Attack Against RSA Key Generation in Intel SGX SSL. *Proceedings of the 2018 on Asia Conference on Computer and Communications Security, AsiaCCS 2018, Incheon, Republic of Korea, June 04-08, 2018*. Ed. by J. Kim, G. Ahn, S. Kim, Y. Kim, J. López and T. Kim. ACM, 2018, 575–586. DOI: 10.1145/3196494.3196524. URL: <https://doi.org/10.1145/3196494.3196524>.
- [13] Gridin, I., Pereida García, C., Tuveri, N. and Brumley, B. B. Triggerflow: Regression Testing by Advanced Execution Path Inspection. *Detection of Intrusions and Malware, and Vulnerability Assessment - 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19-20, 2019, Proceedings*. Ed. by R. Perdisci, C. Maurice, G. Giacinto and M. Almgren. Vol. 11543. Lecture Notes in Computer Science. Springer, 2019, 330–350. DOI: 10.1007/978-3-030-22038-9_16. URL: https://doi.org/10.1007/978-3-030-22038-9_16.
- [14] *Callgrind: a call-graph generating cache and branch prediction profiler*. URL: <http://valgrind.org/docs/manual/cl-manual.html>.
- [15] Nethercote, N. and Seward, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. Ed. by J. Ferrante and K. S. McKinley. ACM, 2007, 89–100. DOI: 10.1145/1250734.1250746. URL: <https://doi.org/10.1145/1250734.1250746>.
- [16] *GNU gprof*. URL: <https://sourceware.org/binutils/docs/gprof/>.
- [17] Weiser, S., Zankl, A., Spreitzer, R., Miller, K., Mangard, S. and Sigl, G. DATA - Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries. *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA,*

- August 15-17, 2018*. Ed. by W. Enck and A. P. Felt. USENIX Association, 2018, 603–620. URL: <https://www.usenix.org/conference/usenixsecurity18/presentation/weiser>.
- [18] Free Software Foundation. *GNU Bash*. URL: <https://www.gnu.org/software/bash/>.
- [19] *Ruby Programming Language*. URL: <https://www.ruby-lang.org/en/>.
- [20] Roman, B., Scholin, C., Jensen, S., Massion, E., Roman Marin, I., Preston, C., Greenfield, D., Jones, W. and Wheeler, K. Controlling a Robotic Marine Environmental Sampler with the Ruby Scripting Language. *JALA: Journal of the Association for Laboratory Automation* 12.1 (2007), 56–61. DOI: 10.1016/j.jala.2006.07.013. eprint: <https://doi.org/10.1016/j.jala.2006.07.013>. URL: <https://doi.org/10.1016/j.jala.2006.07.013>.
- [21] Beller, M., Bholanath, R., McIntosh, S. and Zaidman, A. Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software. *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016*. IEEE, 2016, 470–481. URL: <http://dx.doi.org/10.1109/SANER.2016.105>.
- [22] *GDB: The GNU Project Debugger*. URL: <https://www.gnu.org/software/gdb/>.
- [23] *Sharness*. URL: <https://github.com/chriscool/sharness>.
- [24] Aschermann, C. *gdb-mi-parser*. 2019. URL: <https://github.com/eqv/gdb-mi-parser/>.
- [25] Williams, L. A. Integrating Pair Programming into a Software Development Process. *14th Conference on Software Engineering Education and Training, 19-21 February 2001, Charlotte, North Carolina, USA*. IEEE Computer Society, 2001, 27. DOI: 10.1109/CSEE.2001.913816. URL: <https://doi.org/10.1109/CSEE.2001.913816>.
- [26] Gansner, E., Koutsofios, E. and North, S. *Drawing graphs with dot*. Tech. rep. 2006. URL: <http://www.graphviz.org/Documentation/dotguide.pdf>.
- [27] *patch - apply changes to files*. URL: <http://pubs.opengroup.org/onlinepubs/9699919799/utilities/patch.html>.
- [28] *Quilt - Summary [Savannah]*. URL: <http://savannah.nongnu.org/projects/quilt>.
- [29] *Git*. URL: <https://git-scm.com/>.
- [30] Merkel, D. Docker: Lightweight Linux containers for consistent development and deployment. 2014 (2014), 2. URL: <https://www.linuxjournal.com/content/docker-lightweight-linux-containers-consistent-development-and-deployment>.
- [31] Pereida García, C., Hassan, S. ul, Tuveri, N., Gridin, I., Aldaya, A. C. and Brumley, B. B. Certified Side Channels. *29th USENIX Security Symposium, USENIX*

- Security 2020, August 12-14, 2020*. Ed. by S. Capkun and F. Roesner. USENIX Association, 2020, 2021–2038. URL: <https://www.usenix.org/conference/usenixsecurity20/presentation/garcia>.
- [32] *CVE-2019-1547*. Available from MITRE, CVE-ID CVE-2019-1547. 2019. URL: <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-1547>.
- [33] Aas, J., Barnes, R., Case, B., Durumeric, Z., Eckersley, P., Flores-López, A., Halderman, J. A., Hoffman-Andrews, J., Kasten, J., Rescorla, E., Schoen, S. D. and Warren, B. Let's Encrypt: An Automated Certificate Authority to Encrypt the Entire Web. *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. Ed. by L. Cavallaro, J. Kinder, X. Wang and J. Katz. ACM, 2019, 2473–2487. DOI: 10.1145/3319535.3363192. URL: <https://doi.org/10.1145/3319535.3363192>.
- [34] *CVE-2020-12399*. Available from MITRE, CVE-ID CVE-2020-12399. 2020. URL: <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-12399>.
- [35] *CVE-2020-12402*. Available from MITRE, CVE-ID CVE-2020-12402. 2020. URL: <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-12402>.
- [36] *CVE-2020-6829*. Available from MITRE, CVE-ID CVE-2020-6829. 2020. URL: <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-6829>.

A TRIGGERFLOW PAPER

This is a post-peer-review, pre-copyedit version of an article published in Lecture Notes in Computer Science, Volume 11543. The final authenticated version is available online at: https://doi.org/10.1007/978-3-030-22038-9_16

Triggerflow: Regression Testing by Advanced Execution Path Inspection

Iaroslav Gridin, Cesar Pereida García, Nicola Tuveri, and Billy Bob Brumley

Tampere University, Tampere, Finland

{iaroslav.gridin,cesar.pereidagarcia,nicola.tuveri,billy.brumley}@tuni.fi

Abstract. Cryptographic libraries often feature multiple implementations of primitives to meet both the security needs of handling private information and the performance requirements of modern services when the handled information is public. OpenSSL, the de-facto standard free and open source cryptographic library, includes mechanisms to differentiate the confidential data and its control flow, including run-time flags, designed for hardening against timing side-channels, but repeatedly accidentally mishandled in the past. To analyze and prevent these accidents, we introduce Triggerflow, a tool for tracking execution paths that, assisted by source annotations, dynamically analyzes the binary through the debugger. We validate this approach with case studies demonstrating how adopting our method in the development pipeline would have promptly detected such accidents. We further show-case the value of the tooling by presenting two novel discoveries facilitated by Triggerflow: one leak and one defect.

Keywords: software testing · regression testing · continuous integration · dynamic program analysis · applied cryptography · side-channel analysis · OpenSSL

1 Introduction

Attacks based on Side-Channel Analysis (SCA) are ubiquitous in microarchitectures and recent research [22, 20] suggest that they are much harder to mitigate than originally believed due to flawed system microarchitectures. Constant-time programming techniques are arguably the most effective and cheapest countermeasure against SCA. Functions implemented following this approach, execute and compute results time-independent from the secret inputs, thus avoiding information leakage.

Implementing constant-time code requires a highly specialized and ever growing skill set such as SCA techniques, operating systems, compilers, signal processing, and even hardware architecture; thus it is a difficult and error-prone task. Unfortunately, code is not always easily testable for SCA flaws due to code complexity and the difficulty of creating the tests themselves. Moreover, cryptography libraries tend to offer several versions of a single algorithm to be used in particular cases depending on the users' needs, thus amplifying the confusion and the possibility of using SCA vulnerable functions.

To that end, we present Triggerflow, a tool that allows to selectively track code paths during program execution. The approach used by Triggerflow is elegant in its simplicity: it reports code paths taken by a given program according to the annotations defined by the user. This enables designing simple regression tests to track control flow skew. Moreover, the tool is extendable and can be integrated in the Continuous Integration (CI) development pipeline, to automatically test code paths in new builds. Triggerflow can be used both as a stand-alone tool to continuously test for known flaws, and as a support tool for other SCA tools when the source code is available. It easily allows examining code execution paths to pinpoint code flaws and regressions.

We motivate our work and demonstrate Triggerflow’s effectiveness by adapting it to work with OpenSSL due to its rich history of known SCA attacks, its wide usage in the Internet, and its rapid and constant development stage. We start by back-testing OpenSSL’s previously known and exploited code flaws, where our tool is able to easily find and corroborate the vulnerabilities. Additionally, using Triggerflow we identify new bugs and SCA vulnerabilities affecting the most recent OpenSSL 1.1.1a version.

In summary, Section 2 discusses previous problems and pitfalls in OpenSSL that led to side-channel attacks. Section 3 describes the Triggerflow tool and Section 4 its application in a CI setting. We analyze in Section 5 the new bugs and vulnerabilities affecting OpenSSL, and in Section 6 we back-test known OpenSSL SCA vulnerabilities to validate the tool’s effectiveness. Section 7 looks at related work. In Section 8 we discuss the limitations of our tool, and finally we conclude in Section 9.

2 Background

2.1 The OpenSSL `BN_FLG_CONSTTIME` Flag

In 2005, OpenSSL started considering SCA in their threat model, introducing code changes in OpenSSL version 0.9.7. The (then new) RSA cache-timing attack by Percival [25] allowed an attacker to recover secret exponent bits during the sliding-window exponentiation algorithm on systems supporting simultaneous multi-threading (SMT). As a countermeasure to this attack, the OpenSSL team adopted two important changes: Commit 3 introduced the constant-time exponentiation flag and `BN_mod_exp_mont_consttime`, a fixed-window modular exponentiation function; and Commit 4 implemented exponent padding. By combining these countermeasures, OpenSSL aimed for SCA resistant code path execution when performing secret key operations during DSA, RSA, and Diffie-Hellman (DH) key exchange, with the goal of performing exponentiation reasonably independent of the exponent weight or length.

The concept is to set the `BN_FLG_EXP_CONSTTIME` flag on `BIGNUM` variables containing secret information: e.g. private keys, secret prime values, nonces, and integer scalars. Once set, the flag drives access to the constant-time security critical modular exponentiation function supporting the flag. Due to performance

reasons, OpenSSL kept both functions: the constant-time version and the non constant-time version of the modular exponentiation operation. The library defaults to the non constant-time function since it assumes most operations are not secure critical, thus they can be done faster, but upon entry to the non constant-time function the input BN variables are checked for the flag and if the program detects the flag is set, it takes an early exit to the constant-time function, otherwise it continues the insecure code path.

As research and attacks on SCA improved, Aciçmez et al. [1] demonstrated new SCA vulnerabilities in OpenSSL. More precisely, the authors showed that the default BN division function, and the Binary Extended Euclidean algorithm (BEEA) function—used in OpenSSL to perform modular inversion operations—are highly dependent on their input values, therefore they leak enough information to perform a cache-timing attack. This discovery forced the introduction of Commit 14, implementing the `BN_div_no_branch` and `BN_mod_inverse_no_branch` functions, offering a constant-time implementation for the respective operations. Moreover, `BN_FLG_EXP_CONSTTIME` was renamed to `BN_FLG_CONSTTIME` to reflect the fact that it offered protection not only to the modular exponentiation function, but to other functions as well.

2.2 Flag Exploitation

During the last three years, the `BN_FLG_CONSTTIME` flag has received a fair amount of attention due to its flawed effectiveness as an SCA countermeasure in OpenSSL. Pereida García et al. [27] showed the issues of having an insecure-by-default approach in OpenSSL by exploiting a flaw during DSA signature generation due to a flag propagation issue. Performing a `FLUSH+RELOAD` [39] attack, the authors fully recover DSA private keys.

Following the previous work, Pereida García and Brumley [26] identified yet another flaw in OpenSSL, this time involving the `BN_mod_inverse` function. Failure to set the flag allowed the authors to successfully perform a cache-timing attack using `FLUSH+RELOAD` to recover secret keys during ECDSA P-256 signature generation in SSH and TLS protocols.

Building on top of the previous works, two research teams [35, 3] discovered independently several SCA flaws in OpenSSL. On the one hand, Aldaya et al. [3] developed and used a simple but effective methodology to find vulnerable code paths in OpenSSL. The authors tracked SCA vulnerable functions in OpenSSL using GDB by placing breakpoints on them. They executed the RSA key generation command, hitting the breakpoints and thus revealing flaws in OpenSSL’s RSA key generation implementation. On the other hand, [35] analyzed the RSA key generation implementation and also discovered calls to the SCA vulnerable GCD function. In both cases, the authors noticed a combination of non constant-time functions in use, failure to set flags, and flags not propagated to `BIGNUM` variables caused OpenSSL to leak key bits. Moreover, both works demonstrate that it is possible to retrieve enough key bits to fully recover an RSA key after a single SCA trace using different cache techniques and threat models (page-level or `FLUSH+RELOAD`).

The previous works highlight a clear and serious issue surrounding the constant-time flag. The developers need to identify all the possible security critical cases in OpenSSL where the flag must be set in order to prevent SCA attacks, which has proven to be a laborious and clearly error-prone task. Even if done thoroughly and correctly, the developers must still ensure code changes do not introduce regressions surrounding the flag.

3 Tracking Execution Paths with Triggerflow

OpenSSL’s regression-testing framework has significantly improved over time, notably following the HeartBleed vulnerability. Nevertheless, the framework has its limitations, with real-world constraints largely imposed by portability requirements weighed against engineering effort. With respect to the `BN_FLG_CONSTTIME` flag, the testing framework does not provide a mechanism to track function calls or examine the call stack. This largely contributes to the root cause of the previously discussed vulnerabilities surrounding the `BN_FLG_CONSTTIME` flag: the testing framework cannot accommodate a reasonable regression test in these instances.

With this motivation, our work began by designing Triggerflow¹: a tool for tracking execution paths. After marking up the source code with special comments, its purpose is to detect when code hits paths of interest. We wrote Triggerflow in Ruby² and it uses GDB³ for inspecting code execution. In support of Open Science [18], Triggerflow is free and open source, distributed under MIT license.

We chose GDB since it provides all the required functionality: an established interface for choosing trace points and inspecting the program execution, as well as a machine-readable interface⁴. Additionally, GDB supports a wide variety of platforms, architectures, and languages.

Architecture. The high level concept of Triggerflow is as follows.

1. The inputs to Triggerflow are: a directory with annotated source code, instructions to build it, commands to run and debug, and optionally patches to apply before building.
2. Triggerflow scans the source code for special keywords, which are typically placed in comments near related lines of code, and builds a database of annotations.
3. Triggerflow commences the build, then runs the given commands (*triggers*) under GDB, instructed to set breakpoints at all points of interest.
4. When GDB reports hitting a breakpoint, Triggerflow inspects the backtrace supplied by GDB, makes decisions based on the backtrace and stored annotations, and possibly logs the code path that led to it.

¹ <https://gitlab.com/nisec/triggerflow>

² <https://www.ruby-lang.org/en/>

³ <https://www.gnu.org/software/gdb/>

⁴ https://sourceware.org/gdb/onlinedocs/gdb/GDB_002fMI.html

In addition to verbose raw logging, Triggerflow provides output in Graphviz DOT format, allowing easy conversion to PDF, image, and other formats.

Annotations. Using marked up source code allows leveraging existing tools for merging code changes to (semi)automatically update annotations to reflect codebase changes. It is best when annotations are maintained in the original code, and updated by the author of related changes, but for the purposes of code analysis by a third party, Triggerflow also supports storing annotations separately, in form of patches that define annotation context. Our tool currently supports four different annotations, described below and illustrated in Figure 1.

1. `TRIGGERFLOW_POI` is a point of interest and it is always tracked. The Triggerflow tool reports back every time the executing code steps into it.
2. `TRIGGERFLOW_POI_IF` is a conditional point of interest, thus it is conditionally tracked. The Triggerflow tool reports back every time the code annotated is stepped into and the given expression evaluates to true.
3. `TRIGGERFLOW_IGNORE` is an ignore annotation that allows to safely ignore specific code lines resulting in code execution paths that are not interesting (false positives).
4. `TRIGGERFLOW_IGNORE_GROUP` is a group ignore annotation that allows to safely ignore a specific code execution path if and only if every line marked with the same group ID is stepped into.

<pre> 1 /* code before */ 2 if(a % 2 == 0) // TRIGGERFLOW_POI 3 /* code after */ 1 if(something) { 2 a = publickey; // ↪ TRIGGERFLOW_IGNORE_GROUP ↪ ec_publickey 3 } 4 call_suspicious_code(a) // ↪ TRIGGERFLOW_IGNORE_GROUP ↪ ec_publickey </pre>	<pre> 1 /* code before */ 2 call_suspicious_code(a) // ↪ TRIGGERFLOW_POI_IF a.private() 3 /* code after */ 1 int call_suspicious_code(int a) { 2 // TRIGGERFLOW_POI 3 /* something interesting with a */ 4 } 5 call_suspicious_code(public_key) // ↪ TRIGGERFLOW_IGNORE </pre>
--	---

Fig. 1. Annotations currently supported by Triggerflow.

3.1 Annotating OpenSSL

Using the known vulnerable code paths previously discussed in Section 2.2, we created a set of annotations for OpenSSL with the intention to track potential leakage during secure critical operations in different public key cryptosystems such as DSA, ECDSA, RSA, as well as high-level CMS routines.

Following a direct approach, as Figure 2 illustrates we placed `TRIGGERFLOW_POI` annotations to track the code path execution of the most prominent information-leaking functions previously exploited. We placed an annotation in the

`BN_mod_exp_mont` function immediately after the early exit to its constant-time counterpart. In the `BN_mod_inverse` function, we placed a similar annotation after the early exit. We added an annotation at the top of the non constant-time `BN_gcd` function since it is known for being previously used during security critical operations but this function does not have an exit to a constant-time implementation, i.e., it is oblivious to the `BN_FLG_CONSTTIME`.

On the ECC code we annotated the `ec_wNAF_mul` function. This function implements *wNAF* scalar multiplication, a known SCA vulnerable function exploited several times in the past [12, 8, 28, 4, 2]. Similar to the previous cases, upon entry to this function, an early exit is available to a more SCA secure Montgomery ladder scalar multiplication `ec_scalar_mul_ladder`, thus we added the annotation immediately after the early exit.

The strategy to annotate `BN_div` varies depending on the OpenSSL branch. For branches up to and including 1.1.0, the function checks the flag on BN operands and assigns `no_branch = 1` if it detects the flag. Hence we annotate with a `no_branch != 1` conditional breakpoint. The master and 1.1.1 branches recently applied SCA hardening to its callee `bn_div_fixed_top` to make it oblivious to the flag. The corner case is when the number of words in BN operands are not equal, and inside the resulting data-dependent control flow we add an unconditional point of interest annotation.

Ideally, the previous annotations should never be reached, since we assume OpenSSL follows a constant-time code path during the execution of these secure critical operations. Yet one of the most security-critical parts of the process is marking false positive annotations. To give an idea of the scope of such marking, with the above described point of interest annotations applied to the OpenSSL 1.1.0 branch, and no ignore annotations, Triggerflow identifies 84 potentially errant code paths, provided with only a basic set of 25 triggers.

4 Continuous Integration

As previously discussed, our main motivation for Triggerflow is the need to test for regressions in OpenSSL surrounding the `BN_FLG_CONSTTIME` flag. From the software quality perspective, and given the previously exploited vulnerabilities discussed later in Section 6, there is a clear need for an automated approach that accounts for the time dimension and a rapidly changing codebase. Seemingly small and insignificant changes can suddenly shift codepaths, and when PRs are proposed and merged we want to be automatically informed. Using code marked up for Triggerflow allows establishing CI, automatically testing code for introducing unsafe codepaths. We propose (and deploy) the following approach to establish an automatic CI pipeline using Triggerflow and GitLab’s infrastructure, illustrated in Figure 3.

- Create a special Git repository containing Triggerflow configuration, trigger list, annotations in form of Quilt⁵ patch queue, and a submodule containing

⁵ <https://savannah.nongnu.org/projects/quilt>

```

1  int ec_wNAF_mul(const EC_GROUP *group, EC_POINT *r,
↳ const BIGNUM *scalar,
2  ↳ size_t num, const EC_POINT *points[],
↳ const BIGNUM *scalars[],
3  ↳ BN_CTX *ctx)
4  {
5  /* ... */
6  if ((scalar == NULL) && (num == 1)) {
7  return ec_scalar_mul_ladder(group, r,
↳ scalars[0], points[0], ctx);
8  }
9  }
10
11 if (scalar != NULL) { /* TRIGGERFLOW_POI */
}

1  int bn_div_fixed_top(BIGNUM *dv, BIGNUM *rm, const
↳ BIGNUM *num,
2  ↳ const BIGNUM *divisor, BN_CTX *ctx)
3  {
4  /* ... */
5  div_n = sdiv->top;
6  num_n = snum->top;
7
8  if (num_n <= div_n) {
9  /* TRIGGERFLOW_POI */
10 /* caller didn't pad dividend -> no
↳ constant-time guarantee... */
}

1  int BN_gcd(BIGNUM *r, const BIGNUM *in_a, const BIGNUM
↳ *in_b, BN_CTX *ctx)
2  {
3  BIGNUM *a, *b, *t; /* TRIGGERFLOW_POI */
}

1  BIGNUM *BN_mod_inverse(BIGNUM *in,
2  ↳ const BIGNUM *a, const BIGNUM
↳ *n, BN_CTX *ctx)
3  {
4  BIGNUM *A, *B, *X, *Y, *M, *D, *T, *R = NULL;
5  BIGNUM *ret = NULL;
6  int sign;
7
8  if ((BN_get_flags(a, BN_FLG_CONSTTIME) != 0)
9  || (BN_get_flags(n, BN_FLG_CONSTTIME) != 0)) {
10 return BN_mod_inverse_no_branch(in, a, n, ctx);
11 }
12
13 bn_check_top(a); /* TRIGGERFLOW_POI */
}

1  int BN_mod_exp_mont(BIGNUM *rr, const BIGNUM *a, const
↳ BIGNUM *p,
2  ↳ const BIGNUM *m, BN_CTX *ctx,
↳ BN_MONT_CTX *in_mont)
3  {
4  /* ... */
5  if (BN_get_flags(p, BN_FLG_CONSTTIME) != 0
6  || BN_get_flags(a, BN_FLG_CONSTTIME) != 0
7  || BN_get_flags(m, BN_FLG_CONSTTIME) != 0)
8  {
9  return BN_mod_exp_mont_consttime(rr, a, p, m,
↳ ctx, in_mont);
10 }
11
12 bn_check_top(a); /* TRIGGERFLOW_POI */
}

```

Fig. 2. Top left: a TRIGGERFLOW_POI annotation in the wNAF scalar multiplication function after the early exit. Middle left: a TRIGGERFLOW_POI annotation during BN_div execution. Bottom left: a TRIGGERFLOW_POI annotation in OpenSSL’s insecure BN_gcd function. Top right: a TRIGGERFLOW_POI annotation in OpenSSL’s BN_mod_inverse function after the early exit. Bottom right: a TRIGGERFLOW_POI annotation in BN_mod_exp_mont after the early exit.

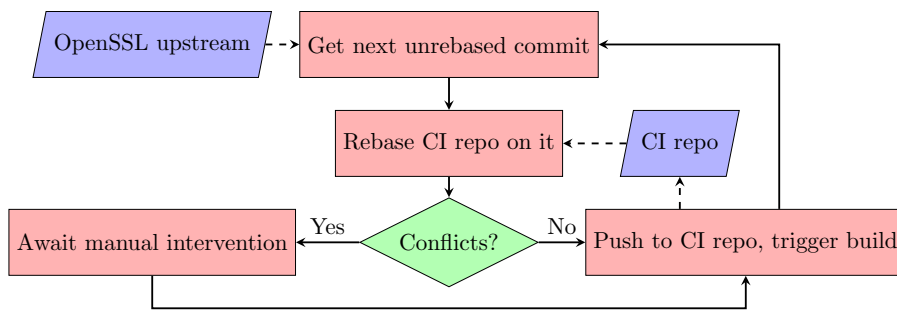


Fig. 3. CI flow illustrated.

code to test (in our case, OpenSSL). This repository is hosted on a GitLab instance and includes the description of the testing process in GitLab format, `.gitlab.yml`.

- Two *runners* are established on separate machines, connected to the GitLab instance. A runner is automated testing software which creates a container and runs testing routines according to rules in `.gitlab.yml`. We maintain two runners with different architectures, `x86_64` and `aarch64`. The runners are based in our infrastructure. When new code is pushed into the GitLab repository and `.gitlab.yml` is present, runners execute the tests and report status back to GitLab, where results are then reviewed.
- A separate software (*repatcher*) is continuously monitoring main OpenSSL code repository for updates and adapting annotations to changed code. If changes can be applied automatically, *repatcher*⁶ pushes updated code to GitLab where it is tested. Otherwise, a human is notified to resolve conflicts and update the patches manually. After that, *repatcher*'s work automatically continues. Repatcher is based in our infrastructure.

This process is independent of any support from the original developers. Of course, a better approach is to have developers themselves integrate and maintain Triggerflow annotations upstream, or potentially enforce them at compile time.

Unfortunately, successful deployment of such a CI pipeline depends on code being buildable on every upstream commit, which is sometimes not the case with OpenSSL. Still, with minimal manual inspection it makes a great automatic testing setup: Figure 4 illustrates our CI testing OpenSSL's master branch using Triggerflow. The results of our CI system instance are public⁷, monitoring `master`, `1.1.1` and `1.1.0` branches of OpenSSL.

Average build of OpenSSL on our runners takes 85 s on `x86_64` (440 s on `aarch64`), and Triggerflow takes average of 26 s to run our set of triggers on `x86_64` (92 s on `aarch64`).
















Status	Pipeline	Commit	Stages
 passed	#1495 by  latest	 patched/mas... -> f3b5c690  [master:c8147d37ccaaf28c...]	 00:07:42 📅 1 hour ago
 passed	#1494 by 	 patched/mas... -> 81d96fbd  [master:fe16ae5f95fa86ddb...]	 00:07:52 📅 1 hour ago
 passed	#1493 by 	 patched/mas... -> 9fb8e7df  [master:0b76ce99aaa5678b...]	 00:07:46 📅 1 hour ago

Fig. 4. GitLab CI running: Triggerflow testing OpenSSL code.

⁶ <https://gitlab.com/nisec/repatcher>

⁷ <https://gitlab.com/nisec/openssl-triggerflow-ci>

5 New Bugs and Vulnerabilities

With the tooling in place, our first task was to examine functionality issues that could arise with applying the annotation patches to a shifting codebase. The EC module recently underwent a quite heavy overhaul regarding SCA security [33]. We used that as a case study, and in this section we present two discoveries facilitated by Triggerflow: one leak and one software defect.

5.1 A New Leak

We started from Commit 1 and the Triggerflow unit test in question is ECDSA signing in `ecdsa_oss1.c`. The test passed at that commit, hence the tooling proceeded with subsequent commits. They all passed unit testing, until reaching Commit 2. The purpose of said commit was to fix a regression in the padding of secret scalar inputs in the timing-resistant elliptic curve scalar multiplication, using the group cardinality rather than the generator order, supporting cryptosystems where the distinction is relevant (e.g., ECDH and cofactor variants). Figure 5 illustrates the failed unit test.

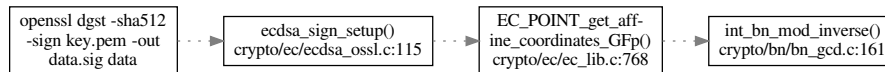


Fig. 5. Insecure flow: projective to affine point conversion (abridged).

The fix. In this case, what the tooling is telling us is that the code is traversing the insecure modular inversion path when converting from projective to affine coordinates. Examining this function, it has always been oblivious to the constant-time flag, yet academic results suggest that said conversion should be protected [24, 23]. Put another way, Commit 2 is not the culprit—the function is insecure by design. Instead of simply enabling the flag, we chose⁸ to add a `field_inv` function pointer inside the `EC_METHOD` structure, alongside existing pointers for other finite field operations such as `field_mul` and `field_sqr`. This allowed us to unify the finite field inversion across the EC module, instead of each function meticulously enabling the constant-time flag when calling `BN_mod_inverse`. Once unified, we can ensure default SCA hardening through a single interface. We provided three different implementations for this pointer for three different `EC_METHOD` instances:

1. `EC_GFp_mont_method` is the default for prime curves and pre-computes a Montgomery arithmetic structure for finite field arithmetic. This is convenient for inversion via FLT, which is modular exponentiation with a fixed exponent and variable base—benefiting generously from the Montgomery

⁸ <https://github.com/openssl/openssl/pull/8254>

arithmetic. Hence our `field_inv` implementation is a straightforward version of FLT in this case.

2. `EC_GFp_simple_method` is a fallback method that contains much of the boilerplate code pointed to by several other `EC_METHOD` implementations. For example, those that implement their own custom arithmetic, such as NIST curves that use Mersenne-like primes. Here, no Montgomery structure is guaranteed to exist. Hence our `field_inv` implementation is blinding, computing $a^{-1} = b/(ab)$ with b chosen uniformly at random and the ab term inverted via `BN_mod_inverse`.
3. `EC_GF2m_simple_method` is the only method for binary curves present in the OpenSSL codebase. Here `field_inv` is a simple wrapper around `BN_GF2m_mod_inv`, which is already SCA-hardened with blinding.

With these SCA-hardened `field_inv` function pointers in place, we then transitioned all finite field inversions in the EC module from `BN_mod_inverse` and `BN_GF2m_mod_inv` to our new pointer, including that of the projective to affine conversion. After these changes, Triggerflow unit tests were successful.

5.2 A New Defect

The previous unit test failure is curious in the sense that Commit 2 was essentially unrelated to projective to affine conversion. As stated above, that conversion has always been oblivious to the constant-time flag. We were left with the question of how such a change could trigger an insecure behavior in an unrelated function.

Using the debugger to compare the internal state when executing `EC_POINT_get_affine_coordinates_GFp` in Commit 2 and its parent, we discovered that, until the latter, a temporary variable storing one of the inputs to `BN_mod_inverse` was flagged as constant-time even if the flag was not explicitly set with the dedicated function. The temporary variable in question was obtained through a `BN_CTX` object, a buffer shared among various functions that simulates a hardware stack to store `BIGNUM` variables, minimizing costly memory allocations—we defer to [13] for more details on the internals of the `BN_CTX` object.

In this case, the `BN_CTX` object is created in the top level function implementing signature generation for the ECDSA cryptosystem, and is shared among most of its callees and descendants; the analysis led to discover that the `BN_CTX` buffer retained the state of `BN_FLG_CONSTTIME` for each stored `BIGNUM` variable, allowing functions to alter the value of `BN_FLG_CONSTTIME`, and thus occasionally the execution flow, of subsequently called functions sharing the same `BN_CTX`.

The fix. This long-standing defect raises several concerns:

- as in the case that led to its discovery, retrieving a `BIGNUM` variable from the `BN_CTX` with `BN_FLG_CONSTTIME` unexpectedly set, might lead to unintentional execution of a timing-resistant code-path. This could be perceived as a benign effect, but hides unexpected risks as it generates false negatives

during security analysis. Moreover, changes as trivial as getting one more temporary variable from the shared `BN_CTX`—or even just changing the order by which temporary variables are retrieved—can influence the execution flow of seemingly unrelated functions, eluding manual analysis and defying developer expectations;

- a `BIGNUM` variable with `BN_FLG_CONSTTIME` unexpectedly set could reach function implementations that execute in variable time and should never be called with confidential inputs marked with `BN_FLG_CONSTTIME`. Such functions diligently check for API abuse and raise exceptions at run time: this defect can then result in unexpected application crashes or potentially expose to bug attacks;
- automated testing is made fragile, in part for the false negatives already mentioned, but additionally because the test suite becomes not representative of external application usage of the library, as different usage patterns of a shared `BN_CTX` in unrelated functions lead to different execution paths. Finally, the generated failure reports could be misleading as changes in unrelated functions might end up triggering errors in other modules.

The fix itself was relatively straightforward, and consisted in unconditionally clearing `BN_FLG_CONSTTIME` every time a `BIGNUM` variable is retrieved from a `BN_CTX`⁹.

What is remarkable is how Triggerflow assisted in the discovery of a defect that had been unnoticed for over a decade, automating the interaction with the debugger to pinpoint which revisions triggered the anomalous behavior.

6 Validation

In order to validate our work, we present next a study of the known flaws briefly discussed in Section 2.2 that led to several SCA attacks, security advisories, and significant manpower downstream to address these issues. We present these flaws as case studies, briefly discussing the root cause, security implications, and the results of running our tooling against an annotated OpenSSL. We separate the cases by cryptosystem and at the same we (mostly) follow the chronological discovery of these flaws.

As part of the validation, we used the same OpenSSL versions as in the original attacks. To that end, we forked OpenSSL branches on the respective versions and then, we applied the set of annotations previously discussed in Section 3.1. This approach allowed us to quickly back test and validate the effectiveness of our tooling to detect potential leakage in OpenSSL.

The list of cases presented here is not exhaustive but serves three purposes:

1. it gives insight to the types of flaws that our Triggerflow is able to find;
2. it shows it is not a trivial task to do, let alone automate; and

⁹ <https://github.com/openssl/openssl/pull/8253>

3. it demonstrates the fragility of the `BN_FLG_CONSTTIME` countermeasure introduced 14 years ago and the need of a secure-by-default approach in cryptography libraries such as OpenSSL.

Moreover, the flaws and vulnerabilities presented in this section and in Section 5 demonstrate the effectiveness and efficiency of integrating Triggerflow to the development pipeline. Maintaining annotations, either as separate patches or integrated in the code base, might be seen as tedious or error-prone but the automation benefits outweigh the disadvantages. On the one hand, maintaining annotations does not require deep and specialized understanding of the code, compared to manually finding and triggering all the possible vulnerable code paths across several platforms, CPUs, and versions. On the other hand, a misplaced annotation does not introduce flaws nor vulnerabilities, since they are used only for testing and reporting purposes.

6.1 DSA

The DSA signature generation implementation in OpenSSL has arguably the longest and most troubled history of SCA issues. In 2016, a decade after `BN_FLG_CONSTTIME` and the constant-time exponentiation function countermeasures were introduced, Pereida García et al. [27] discovered that the constant-time path was not taken due to a flag propagation issue. The authors noticed that `BN_copy` effectively copies the content from a `BIGNUM` variable to another but it fails to copy the existing flags, thus flags are not propagated and the constant-time flag must be set again. This issue left the DSA signature generation vulnerable to cache-timing attacks for more than a decade. To test this issue, we pointed Triggerflow at our annotated `OpenSSL_1_0_2k` branch, resulting in Figure 6 and therefore correctly reporting the flaw.

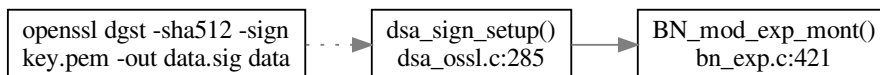


Fig. 6. Triggerflow detecting CVE-2016-2178, the flawed CVE-2005-0109 fix (abridged).

The authors provided a fix for this issue in Commit 5, but at the same time they introduced a new flaw in the modular inversion operation during DSA signature generation. This new vulnerability was enabled due to a missing constant-time flag in one of the input values to the `BN_mod_inverse` function. At that time, the flaw was confined to the development branch, subsequently promptly fixed in Commit 6, thus it did not affect users. Figure 7 shows the result of pointing Triggerflow to OpenSSL in Commit 5, detecting the flawed fix.

Later in 2018, Weiser et al. [36] found additional SCA vulnerabilities in DSA. The authors exploited a timing variation due to the `BIGNUM` structure to re-

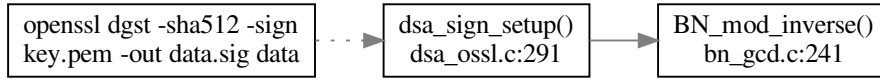


Fig. 7. Triggerflow detecting the flawed CVE-2016-2178 fix (abridged).

cover DSA private keys, an unrelated issue to the `BN_FLG_CONSTTIME` flag. However, the fix provided for this issue in Commit 8 was incomplete, and moreover it introduced a new SCA flaw, once again due to not setting a flag properly. Triggerflow detected this flaw (see Figure 8) in the `OpenSSL_1_1_1` branch, later fixed in Commit 9 but again only present briefly in development branches.

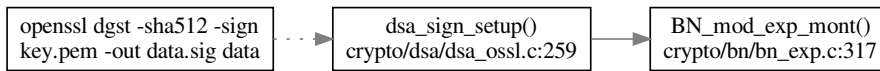


Fig. 8. Triggerflow detecting the flawed CVE-2018-0734 fix (abridged).

In the same work, the authors discovered that every time the library loads a DSA private key, it calculates the corresponding public key following a non constant-time code path due to a missing flag, and therefore is also vulnerable to SCA attacks. In fact, Triggerflow previously detected this vulnerability while back-testing Commit 5, suggesting that this issue was long present in the codebase and could have been detected earlier. This issue was recently fixed in Commit 7.

6.2 ECDSA

OpenSSL's ECDSA implementation has also been affected by SCA leakage. Pereira García and Brumley [26] discovered that the `BN_FLG_CONSTTIME` flag was not set at all during ECDSA P-256 signature generation. More specifically, the modular inversion operation was performed using the non constant-time path in the `BN_mod_inverse` function, thus leaving the scalar k vulnerable to SCA attacks.

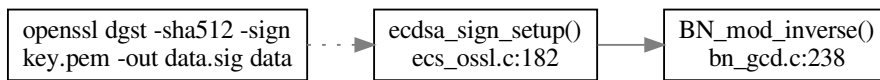


Fig. 9. Triggerflow detecting CVE-2016-7056 (abridged).

Similar to the previous case and in order to back-test this issue, we pointed Triggerflow to the annotated `OpenSSL_1_0_1u` branch and then we generated

ECDSA signatures, triggering the breakpoints. The tool reported back an insecure usage of the modular inversion function as shown in Figure 9. The flag was not set in the nonce k prior to the modular inversion operation. Surprisingly, this issue is still present in the OpenSSL 1.0.1 branch although the authors provided a patch for it, mainlined by the vast majority of vendors. It is worth mentioning the OpenSSL 1.0.1 branch reached EOL around the same time as the work—we assume that is the reason the OpenSSL team did not integrate it.

6.3 RSA

In 2018, two independent works [35, 3] discovered several SCA flaws during RSA key generation in OpenSSL. OpenSSL’s RSA key generation is a fairly complex implementation due to the use of several different algorithms during the process. It requires the generation of random integers; testing the values for primality; computing the greatest common divisor and the least common multiple, using secret values as input. For all of the previous reasons, it is not trivial to implement a constant-time RSA key generation algorithm. Both research works identified missing flags, flags set in the wrong variable, and a direct call to the non constant-time function `BN_gcd` as the culprits enabling the attacks.

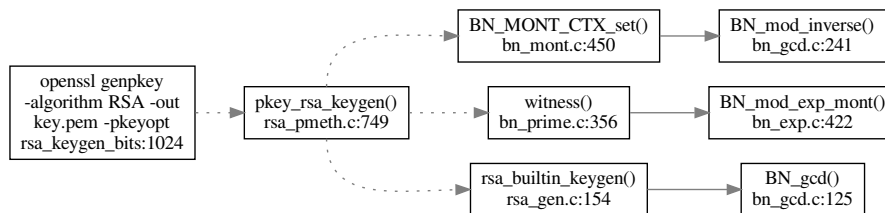


Fig. 10. Triggerflow detecting CVE-2018-0737 (abridged).

During back testing we used an annotated `OpenSSL_1_0_2k` branch, and we pointed the Triggerflow tool at it. It successfully reported all the vulnerabilities discovered by the authors. The authors submitted a total of four commits to OpenSSL codebase to fully mitigate this issue—see Commit 10, Commit 11, Commit 12, and Commit 13 for more details.

7 Related Work

The Triggerflow framework differs from other existing tools in being a tool to assist the development process rather than a system for automated detection and quantification of security vulnerabilities, and aims at being more general purpose and not restricted to the field of cryptographic applications. As such, it should be viewed as complementary rather than alternative to the approaches listed below.

Programming languages. Various works propose and analyze the option of using specialized programming languages to achieve constant-time code generation and verification [10, 14], while others analyze the challenges [7] or opportunities [31] of translating human-readable code into machine instructions through compilers when dealing with cryptographic software and the need for SCA resistant implementations. They differ from this work in the goal: our evaluation is not based on a lack of timing-resistant implementations, but rather in assisting the development process and making sure that insecure paths are not executed, by mistake, with confidential inputs.

Black box testing. These practices are based on statistical analysis to estimate the SCA leakage. *dudect* [29] applies this methodology measuring the timing of the system under test for different inputs.

Static program analysis. These techniques refers to the analysis of the source code [5, 38, 30] (building on the capabilities of the LLVM project to perform the analysis) or annotated machine code [9] of a program to quantify leakages. An alternative to this approach is represented by *CacheAudit* [17, 16] based on symbolic execution, which is usually applied to smaller software or individual algorithms as it requires more resources. *BLAZER* [6] and *THEMIS* [15] employ static analysis to detect side-channels in Java bytecode programs. *BLAZER* introduces a *decomposition* technique associated with *taint tracking* to discover timing channels (or prove their absence) in execution branches tainted by secret inputs. *THEMIS* combines lightweight static taint analysis with precise relational verification to verify the absence of timing or response size side-channels. Similar in spirit as it uses lightweight taint tracking, *Catalyze* [32] is a closed-source, commercial tool to detect potential leakage by filtering conditional branches and array accesses after marking sensitive inputs; the authors apply their tooling to the C-language MbedTLS library. All of these methods share with Triggerflow the requirement of access to the source code of the tested software (either direct or reasonably decompiled).

Dynamic program analysis. These techniques detect, measure, and accurately locate microarchitecture leakage during the execution of the code in the system. *ctgrind* [21], based on *Valgrind memcheck*, monitors control flow and memory accesses for dependencies on secret data. Previous work [37, 36] uses *Dynamic Binary Instrumentation*, adding instrumentation at run-time to collect metadata and measurements directly to the binary code without altering the execution flow of the program, independently providing extensible frameworks with high accuracy and supporting leakage models for the most relevant microarchitecture attacks. Relevant recent works employ symbolic execution to detect side-channel leaks. *CacheD* [34] is a hybrid approach that combines DBI, symbolic execution, taint tracking, and constraint solving, while the more recent *CaSym* [11] employs cache-aware IR symbolic execution; both works then combine different cache models to detect cache-based timing channels. *SPECTOR* [19] uses similar symbolic execution techniques in combination with *speculative non-interference* models to detect speculative execution leaks and

optimization opportunities in the strategies used by compilers to implement hardening measures.

Triggerflow is similar to Dynamic Program Analysis techniques with respect to performing the evaluation when the software is actively running on the target system. Although limited by requiring access to the source code, Triggerflow can leverage this property and avoid any instrumentation: the tested binary is exactly the one generated by the build process of the target, with the only requirement of not stripping the debug symbols, to aid GDB in mapping function names and the memory addresses of the routines included in the target software.

8 Limitations

Triggerflow requires access to the sources of the target software, and to annotate it with markup comments as described in Section 3. Preferably, Triggerflow annotations should be maintained directly in the codebase of the upstream target project, but Triggerflow includes support for versioning of annotation patches for the analysis of third-party projects. Additionally, it is worth stressing that Triggerflow does not automatically detect where to annotate the target code—this goes beyond the tool capabilities. Instead, it relies on developer expertise to annotate the execution paths of interest. As such, source code access is a limit only for the analysis of closed-source third-party projects, which fall out of the immediate scope of Triggerflow as an aid tool for the development process.

Triggerflow depends on the availability of GDB and Ruby on the target platform, and is limited to the executables that can be debugged through GDB. This is arguably a minor concern, with the only remarkable exception that debugging through GDB inside a virtualized container usually requires overriding the default set of system call restrictions that is meant to isolate the supervisor from the container, raising security concerns when running Triggerflow for third-party CI and partially limiting the selection of available CI platforms.

The tools developed during this work can also be applied to other software projects, not just OpenSSL. Triggerflow can work with any language GDB supports and is useful for analyzing and testing execution paths through any complex project that meets the minimal requirements.

A case study. To substantiate the above claims and demonstrate the flexibility of Triggerflow, we annotated the ECC portion of `golang`¹⁰. The documentation states the `P384` (pseudo-)class for NIST P-384 curve operations is not constant-time. Indeed, the `ScalarMult` method is textbook double-and-add scalar multiplication. We placed a `TRIGGERFLOW_POI` annotation inside this method, and used a `golang` ECDSA signing application as a trigger. Figure 11 shows the result, confirming Triggerflow is not restricted to OpenSSL or the C language.

¹⁰ <https://golang.org/pkg/crypto/elliptic/>

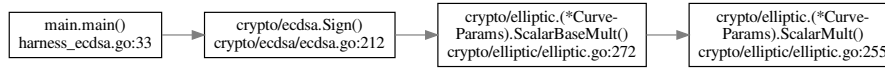


Fig. 11. Triggerflow detecting an insecure scalar multiplication path in golang.

9 Conclusion

Triggerflow complements the results offered by any of the analysis techniques described in Section 7: in large software projects like OpenSSL, pinpointing the location of a detected leak might not be sufficient. Similarly to other cryptographic libraries, OpenSSL often includes several implementations of the same primitive, many of which are designed for performance and safe to use only when all the inputs are public. When a leak is detected in one of these functions, developers are challenged with the task of discovering why and how secret data reached the insecure code path, rather than altering the location where the leakage is reported. As demonstrated in Sections 5 and 6, Triggerflow can be successfully and efficiently used to aid developers in these situations and, through CI, prevent regressions in the handling of secret data.

Considering the high number of valid combinations of supported platforms and build-time options for OpenSSL, and that the available implementations and control flow depend on these specific combinations, Triggerflow is a good solution to aid developers by exhaustively automating the `BN_FLG_CONSTTIME` tests and prevent future regressions similar to the ones described in this work.

In the context of using Triggerflow with OpenSSL to monitor `BN_FLG_CONSTTIME`, it should be mentioned that, security-wise, a secure-by-default approach would be desirable: i.e., all `BIGNUM` are considered *constant-time* unless the programmer explicitly marks them as public, so that when alternatives exist, the default implementation of each algorithm is the timing-resistant one, and insecure but more efficient ones need to be enabled explicitly and after careful examination. On the other hand, such change has the potential for being disruptive for existing applications, and is therefore likely to be rejected or implemented over a long period of time to meet the project release strategy.

Future work. On top of continued development of the tool as discussed, we plan to expand on this work in the future to widen the coverage of the OpenSSL library and of the project *apps* and their options, by setting more triggers and point of interest across multiple architectures and build-time options. In parallel, to further demonstrate the capabilities of the tool we plan to apply a similar methodology to other security libraries and cryptographic software, aiming at uncovering, fixing, and testing related timing leaks.

Responsible disclosure. All PRs submitted as a result of this work were coordinated with the OpenSSL security team. Following the GitHub PR URLs, readers will find more extensive discussions of the security implications of the identified leak and defect. To briefly summarize: (1) the leakage during projective to affine conversion does not appear to be exploitable with recent SCA

hardening to the EC module—we speculate it can only be utilized in combination with some other novel leak, by which time the larger additional leak would likely be enough independently; (2) while we were able to implement a straw man application to demonstrate the BN_CTX defect (reaching unintended code paths and inducing function failures), we were unable to locate a real-world OpenSSL-linking application matching our PoC characteristics, nor any technique to exploit the defect *within* the OpenSSL library itself. We also filed a report with CERT, summarizing our security findings.

Acknowledgments. This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No 804476).

References

1. Aciçmez, O., Gueron, S., Seifert, J.: New branch prediction vulnerabilities in OpenSSL and necessary software countermeasures. In: Cryptography and Coding, 11th IMA International Conference, Cirencester, UK, December 18-20, 2007, Proc. LNCS, vol. 4887, pp. 185–203. Springer (2007), https://doi.org/10.1007/978-3-540-77272-9_12
2. Aldaya, A.C., Brumley, B.B., ul Hassan, S., Pereida García, C., Tuveri, N.: Port contention for fun and profit. In: 2019 IEEE Symposium on Security and Privacy, SP 2019, Proc., 20-22 May 2019, San Francisco, California, USA. pp. 1037–1054. IEEE (2019), <https://doi.org/10.1109/SP.2019.00066>
3. Aldaya, A.C., Pereida García, C., Alvarez Tapia, L.M., Brumley, B.B.: Caching attacks on RSA key generation. IACR Cryptology ePrint Archive 2018(367) (2018), <https://eprint.iacr.org/2018/367>
4. Allan, T., Brumley, B.B., Falkner, K.E., van de Pol, J., Yarom, Y.: Amplifying side channels through performance degradation. In: Proc., 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016. pp. 422–435. ACM (2016), <http://doi.acm.org/10.1145/2991079.2991084>
5. Almeida, J.B., Barbosa, M., Barthe, G., Dupressoir, F., Emmi, M.: Verifying constant-time implementations. In: 25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016. pp. 53–70. USENIX Association (2016), <https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida>
6. Antonopoulos, T., Gazzillo, P., Hicks, M., Koskinen, E., Terauchi, T., Wei, S.: Decomposition instead of self-composition for proving the absence of timing channels. In: Proc., 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017. pp. 362–375. ACM (2017), <https://doi.org/10.1145/3062341.3062378>
7. Balakrishnan, G., Reps, T.W.: WYSINWYX: what you see is not what you execute. ACM Trans. Program. Lang. Syst. 32(6), 23:1–23:84 (2010), <https://doi.org/10.1145/1749608.1749612>
8. Benger, N., van de Pol, J., Smart, N.P., Yarom, Y.: “Ooh Aah... Just a Little Bit”: A small amount of side channel can go a long way. In: Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proc. LNCS, vol. 8731, pp. 75–92. Springer (2014), https://doi.org/10.1007/978-3-662-44709-3_5

9. Blazy, S., Pichardie, D., Trieu, A.: Verifying constant-time implementations by abstract interpretation. In: Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proc., Part I. LNCS, vol. 10492, pp. 260–277. Springer (2017), https://doi.org/10.1007/978-3-319-66402-6_16
10. Bond, B., Hawblitzel, C., Kapritsos, M., Leino, K.R.M., Lorch, J.R., Parno, B., Rane, A., Setty, S.T.V., Thompson, L.: Vale: Verifying high-performance cryptographic assembly code. In: 26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017. pp. 917–934. USENIX Association (2017), <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/bond>
11. Brotzman, R., Liu, S., Zhang, D., Tan, G., Kandemir, M.: CaSym: Cache aware symbolic execution for side channel detection and mitigation. In: 2019 IEEE Symposium on Security and Privacy, SP 2019, Proc., 20-22 May 2019, San Francisco, California, USA. pp. 364–380. IEEE (2019), <https://doi.org/10.1109/SP.2019.00022>
12. Brumley, B.B., Hakala, R.M.: Cache-timing template attacks. In: Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proc. LNCS, vol. 5912, pp. 667–684. Springer (2009), https://doi.org/10.1007/978-3-642-10366-7_39
13. Brumley, B.B., Tuveri, N.: Cache-timing attacks and shared contexts. In: Constructive Side-Channel Analysis and Secure Design - 2nd International Workshop, COSADE 2011, Darmstadt, Germany, February 24-25, 2011. Proc. pp. 233–242 (2011), <https://tutcris.tut.fi/portal/files/15671512/cosade2011.pdf>
14. Cauligi, S., Soeller, G., Brown, F., Johannesmeyer, B., Huang, Y., Jhala, R., Stefan, D.: Fact: A flexible, constant-time programming language. In: IEEE Cybersecurity Development, SecDev 2017, Cambridge, MA, USA, September 24-26, 2017. pp. 69–76. IEEE Computer Society (2017), <https://doi.org/10.1109/SecDev.2017.24>
15. Chen, J., Feng, Y., Dillig, I.: Precise detection of side-channel vulnerabilities using Quantitative Cartesian Hoare Logic. In: Proc., 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017. pp. 875–890. ACM (2017), <https://doi.org/10.1145/3133956.3134058>
16. Doychev, G., Köpf, B.: Rigorous analysis of software countermeasures against cache attacks. In: Proc., 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017. pp. 406–421. ACM (2017), <https://doi.org/10.1145/3062341.3062388>
17. Doychev, G., Köpf, B., Mauborgne, L., Reineke, J.: Cacheaudit: A tool for the static analysis of cache side channels. ACM Trans. Inf. Syst. Secur. 18(1), 4:1–4:32 (2015), <https://doi.org/10.1145/2756550>
18. Gridin, I., Pereida García, C., Tuveri, N., Brumley, B.B.: Triggerflow. Zenodo (Apr 2019), <https://doi.org/10.5281/zenodo.2645805>
19. Guarnieri, M., Köpf, B., Morales, J.F., Reineke, J., Sánchez, A.: SPECTECTOR: principled detection of speculative information flows. CoRR abs/1812.08639 (2018), <http://arxiv.org/abs/1812.08639>
20. Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre attacks: Exploiting speculative execution. In: 2019 IEEE Symposium on Security and Privacy, SP 2019, Proc., 20-22 May 2019, San Francisco, California, USA. pp. 19–37. IEEE (2019), <https://doi.org/10.1109/SP.2019.00002>

21. Langley, A.: ctgrind—checking that functions are constant time with Valgrind. <https://github.com/agl/ctgrind> (2010)
22. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M.: Meltdown: Reading kernel memory from user space. In: 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018. pp. 973–990. USENIX Association (2018), <https://www.usenix.org/conference/usenixsecurity18/presentation/lipp>
23. Maimut, D., Murdica, C., Naccache, D., Tibouchi, M.: Fault attacks on projective-to-affine coordinates conversion. In: Constructive Side-Channel Analysis and Secure Design - 4th International Workshop, COSADE 2013, Paris, France, March 6-8, 2013, Revised Selected Papers. LNCS, vol. 7864, pp. 46–61. Springer (2013), https://doi.org/10.1007/978-3-642-40026-1_4
24. Naccache, D., Smart, N.P., Stern, J.: Projective coordinates leak. In: Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004, Proc. LNCS, vol. 3027, pp. 257–267. Springer (2004), https://doi.org/10.1007/978-3-540-24676-3_16
25. Percival, C.: Cache missing for fun and profit. In: BSDCan 2005, Ottawa, Canada, May 13-14, 2005, Proc. (2005), <http://www.daemonology.net/papers/cachemissing.pdf>
26. Pereida García, C., Brumley, B.B.: Constant-time callees with variable-time callers. In: 26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017. pp. 83–98. USENIX Association (2017), <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/garcia>
27. Pereida García, C., Brumley, B.B., Yarom, Y.: “Make sure DSA signing exponentiations really are constant-time”. In: Proc., 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016. pp. 1639–1650. ACM (2016), <http://doi.acm.org/10.1145/2976749.2978420>
28. van de Pol, J., Smart, N.P., Yarom, Y.: Just a little bit more. In: Topics in Cryptology - CT-RSA 2015, The Cryptographer’s Track at the RSA Conference 2015, San Francisco, CA, USA, April 20-24, 2015. Proc. LNCS, vol. 9048, pp. 3–21. Springer (2015), https://doi.org/10.1007/978-3-319-16715-2_1
29. Reparaz, O., Balasch, J., Verbauwhede, I.: Dude, is my code constant time? In: Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017. pp. 1697–1702. IEEE (2017), <https://doi.org/10.23919/DATE.2017.7927267>
30. Rodrigues, B., Pereira, F.M.Q., Aranha, D.F.: Sparse representation of implicit flows with applications to side-channel detection. In: Proc., 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016. pp. 110–120. ACM (2016), <http://doi.acm.org/10.1145/2892208.2892230>
31. Simon, L., Chisnall, D., Anderson, R.J.: What you get is what you C: controlling side effects in mainstream C compilers. In: 2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018. pp. 1–15. IEEE (2018), <https://doi.org/10.1109/EuroSP.2018.00009>
32. Takarabt, S., Schaub, A., Facon, A., Guilley, S., Sauvage, L., Souissi, Y., Mathieu, Y.: Cache-timing attacks still threaten IoT devices. In: Codes, Cryptology and Information Security - Third International Conference, C2SI 2019, Rabat, Morocco, April 22-24, 2019, Proc. - In Honor of Said El Hajji. LNCS, vol. 11445, pp. 13–30. Springer (2019), https://doi.org/10.1007/978-3-030-16458-4_2

33. Tuveri, N., ul Hassan, S., Pereida García, C., Brumley, B.B.: Side-channel analysis of SM2: A late-stage featurization case study. In: Proc., 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018. pp. 147–160. ACM (2018), <https://doi.org/10.1145/3274694.3274725>
34. Wang, S., Wang, P., Liu, X., Zhang, D., Wu, D.: Cached: Identifying cache-based timing channels in production software. In: 26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017. pp. 235–252. USENIX Association (2017), <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/wang-shuai>
35. Weiser, S., Spreitzer, R., Bodner, L.: Single trace attack against RSA key generation in Intel SGX SSL. In: Proc., 2018 on Asia Conference on Computer and Communications Security, AsiaCCS 2018, Incheon, Republic of Korea, June 04-08, 2018. pp. 575–586. ACM (2018), <http://doi.acm.org/10.1145/3196494.3196524>
36. Weiser, S., Zankl, A., Spreitzer, R., Miller, K., Mangard, S., Sigl, G.: DATA - differential address trace analysis: Finding address-based side-channels in binaries. In: 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018. pp. 603–620. USENIX Association (2018), <https://www.usenix.org/conference/usenixsecurity18/presentation/weiser>
37. Wichelmann, J., Moghimi, A., Eisenbarth, T., Sunar, B.: MicroWalk: A framework for finding side channels in binaries. In: Proc., 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018. pp. 161–173. ACM (2018), <https://doi.org/10.1145/3274694.3274741>
38. Wu, M., Guo, S., Schaumont, P., Wang, C.: Eliminating timing side-channel leaks using program repair. In: Proc., 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018. pp. 15–26. ACM (2018), <https://doi.org/10.1145/3213846.3213851>
39. Yarom, Y., Falkner, K.: FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In: Proc., 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014. pp. 719–732. USENIX Association (2014), <https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom>

A OpenSSL Commits

- | | |
|--|--|
| 1. fe2d3975880e6a89702f18ec58881307bf862542 | 8. a9cfb8c2aa7254a4aa6a1716909e3f8cb78049b6 |
| 2. a766aab93a282774e63ba918d0bb1c6680a5f292 | 9. 00496b6423605391864fbbd1693f23631a1c5239 |
| 3. 46a643763de6d8e39ecf6f76fa79b4d04885aa59 | 10. e913d11f444e0b46ec1ebbf3340813693f4d869d |
| 4. 0ebfcc8f92736c900bae4066040b67f6e5db8edb | 11. 8db7946ee879ce483f4c81141926e1357aa6b941 |
| 5. 621eaf49a289bfac26d4cbcdcb7396e796784c534 | 12. 54f007af94b8924a46786b34665223c127c19081 |
| 6. b7d0f2834e139a20560d64c73e2565e93715ce2b | 13. 6939eab03a6e23d2bd2c3f5e34fe1d48e542e787 |
| 7. 6364475a990449ef33fc270ac00472f7210220f2 | 14. bd31fb21454609b125ade1ad569ebcc2a2b9b73c |