Automated testing of crypto software using differential fuzzing

Jean-Philippe Aumasson, Yolan Romailler
Kudelski Security
Cheseaux-sur-Lausanne, Switzerland
{jp.aumasson,yolan.romailler}@kudelskisecurity.com

Abstract

This note describes a new tool to automatically test software implementations of cryptographic algorithms. Our tool, called CDF, can be used to perform functionality and security testing, and in particular to find logic bugs, interoperabilities issues, or non-compliance with standard specifications. CDF uses a technique that we call differential fuzzing, in order to look for inconsistencies between two implementations of the same primitive (for example, between two implementations of SHA-3, or between a first library's ECDSA signing and a second library's ECDSA verification).

The goal of CDF is to provide more efficient testing, by providing greater code coverage than test vectors, yet without the cost and complexity of formal verification. CDF also provides limited detectiong of timing leaks.

CDF is coded in Go and is therefore trivially portable to most platforms, and can efficiently run concurrent tests on multi-core architectures. CDF tests software in a black-box and stateless way, and can test executable programs in any form, be it a compiled binary, a Python script, or a shell script running some Java application.

We used CDF to test libraries including Go's crypto package, OpenSSL, Cryptography.io, Crypto++, and mbed TLS. CDF is available at:
`https://github.com/kudelskisecurity/cdf`

1 Introduction

Most basic issues in cryptographic software are spotted thanks to *test vectors*, or sets of input–output values provided by the designers of the algorithm in order to test an implementation's correctness. But a handful of test vectors won't necessarily cover all the code paths of an implementation, nor does their validation ensure that nothing will go wrong. For example, *invalid* test values are rarely provided, and therefore implementations will often tolerate invalid or malformed input, in a way that no two different implementations will behave identically. The upshot is that test vectors conformance isn't enough—an implementation can pass test vectors and still have a completely wrong logic as well as software bugs.

Formal verification will provide strong guarantees that a new implementation behaves similarly than a baseline, trusted implementation. Verification tools can also provide proofs that a given implementation's logic satisfies certain security properties, such as forward secrecy.

Verification has been studied for over three decades [8, 9, 11] and is used by hardware companies [7, 14] to deal with the increasing complexity of their designs. Verification is less common in software engineering processes, although tools are becoming more powerful and usable, such as Galois' SAW [4] as used to verify the `s2n` library [2], or Tamarin [13] as used to verify TLS 1.3 [3] and Wireguard's protocol [5]. Even if formal verification won't prove total correctness nor security, it is orders of magnitude more powerful than test vectors. Why aren't all programs formally verified then? Reasons are that 1) not all programs or programming languages easily lend themselves to verification, and 2) specific skills are required to create and run an appropriate prover.

Our goal is then to provide a tool that sit between test vectors and formal verification, in order to catch the non-trivial issues that test vectors would miss at a much lower cost than formal verification methods. The tool we developed is called CDF, and was first presented at WarCon 2016's lightning talks session when it was only in preliminary stage. Meanwhile CDF was developed during a master's thesis project and used to test crypto implementations from li-

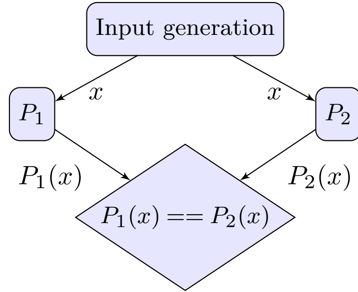

Figure 1: Differential fuzzing of two implementations on the same input, expecting the same output.

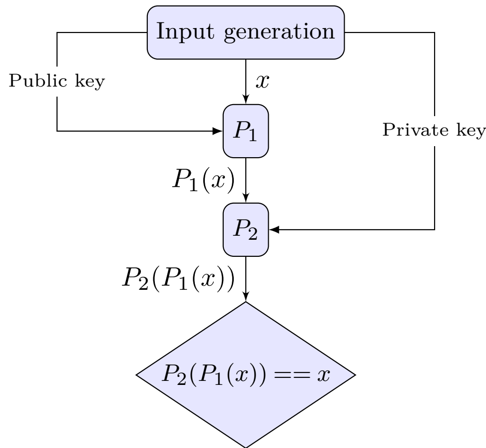

Figure 2: Differential fuzzing of two implementation performing encryption.

braries including Go's crypto package, OpenSSL, Cryptography.io, Crypto++, and mbed TLS. Some of our findings are presented in §**??**.

1.1 Our Approach

CDF combines two methods to detect issues in cryptographic software: comparative testing of implementations in a black-box and stateless fashion, and dedicated tests based on known vulnerabilities and edge cases of the tested algorithms. CDF will detect not only vulnerabilities caused by logic bugs or misimplementations, but also problems of interoperability between implementations. CDF also includes checks for compliance with NIST standards, which a minority of implementations adhere to.

Examples of comparative testing for different cryptographic functionalities are given in Figure 1, Figure 2, and Figure 3, where P_1 and P_2 are the two programs tested.

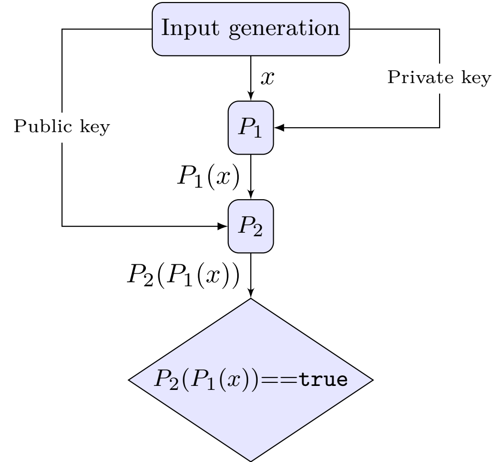

Figure 3: Differential fuzzing of two implementation performing digital signature and verification.

1.2 Related Work

The idea of comparative testing is not new and has been used by developers of cryptographic software to test their libraries. But there exists no generic framework to black-box test implementations against each other.

However, a framework for dedicated unit tests for vulnerabilities and edge cases was recently developed, with *Wycheproof* [6], a project by Google's Bleichenbacher, Duong, Käsper, and Nguyen. Wycheproof includes an extensive set of unit tests for various cryptographic primitives, and discovered many vulnerabilities in Java crypto implementations. At the time of writing, Wycheproof only supports Java's common crypto interface, and it does not aim at comparing implementations between each other.

2 CDF Architecture

CDF is built in Go, following a modular structure. It communicates with the tested executables through command-line arguments via *interface programs*, which convert the tested programs' API into CDF's generic interface for the tested functionality. Support of files instead of command-line arguments, *à la* afl[1], is a planned feature.

CDF is configured by editing the file `config.json` located in its execution directory. Examples of parameters are the minimal and maximal length of key material, or the number of maximum concurrent executions are to be specified.

CDF consists of two main parts:

[1] See `http://lcamtuf.coredump.cx/afl`

- The `cdf-lib` package, which contains all of the interfaces and their tests, helper methods and unit tests.

- The `main.go` file, which produces the CDF executable using the `cdf-lib` package and allows users to test other executables .

The `cdf-lib` package is modular: each interface is implemented in its own file and can be added or removed without consequence, since no other part of the library relies on it.

3 Interfaces

CDF interfaces are abstractions of cryptographic functionalities created in order to simply test specific implementations through *interface programs*. These allow CDF and the tested implementations to communicate without requiring any binary instrumentation or special bindings. In an interface program, the tested implementation must be called with arguments provided to the interface program, encoded in the proper format. For certain languages such as Java, a wrapper is needed in order to allow direct execution of the program by CDF. The wrapper's task is then simply to run the Java byte code on the Java machine and to forward the inputs and outputs, including the errors.

Note however that this approach restricts the maximum input and output size of arguments, since the `exec()` system call sets a limit on both the maximum size of an argument and the maximum number of arguments.

The following sections briefly describe some CDF interfaces.

3.1 The `prf` Interface

The `prf` interface aims to support symmetric-key primitives that take as input a secret key and a public input, such as pseudorandom functions (PRFs), messages authentication codes (MACs), or stream ciphers:

Operation	Input	Output
Computation	`k m`	`h`

Here `k` is a key, `m` is a message (or nonce in case of a stream cipher), and `h` is the result of the PRF computation. Our interface assumes fixed key size and variable input lengths.

3.2 The `enc` Interface

The `enc` interface tests symmetric encryption and decryption operations, typically when performed with a block cipher. It must support encryption and decryption:

Operation	Input	Output
Encryption	`k m`	`c`
Decryption	`k c`	`r`

Here `k` is a symmetric key, `m` is a message, `c` is the ciphertext produced and `r` is the plaintext recovered after decryption.

The `enc` interface does not test IVs and nonces values. This may be implemented using an otional flags.

3.3 The `ecdsa` Interface

The `ecdsa` interface supports *signature* and *verification* operations, as per the following interface:

Operation	Input	Output
Signature	`x y d m`	`r s`
Verification	`x y r s m`	`truth`

Here `x` and `y` are public key coordinates, `d` is a private key, `m` is a message, and `r` and `s` is the ECDSA signature. Finally the `truth` value, a string either "`true`" or "`false`", is the expected boolean result of the verification.

The optional flag `-h` means that the interface provides directly the hash value to be signed, as opposed to a message hashed prior to signaing. This flag is optional because not all APIs allow to bypass the hash computation.

CDF assumes a fixed curve and a fixed hash function, both defined in the tested program.

4 Examples of Tests

Besides comparative testing on various input values and lengths, CDF does a number of dedicated tests in order to identify security shortcomings. CDF can also attempt to detect timing leaks, using a reimplementation of dudect [12], which we used to verify a potential leak in Go's OAEP implementation—but which seems unexploitable using Manger's attack, because of the low signal level.

Below we focus on DSA and ECDSA implementations, presenting some test cases as well as the results observed with Go and OpenSSL. Indeed, many libraries do not provide any way for the user to check

the validity of the domain parameters (for the DSA and ECDSA algorithm) or even of the public or private keys. We focus on Go and OpenSSL because of their popularity and the carefulness of their implementations, but they are not the only affected libraries. It is noteworthy that we could not find any problem in the Crypto++ library, so there also are libraries featuring good defense in depth, not concerned by those issues.

4.1 DSA

The Digital Signature Algorithm (DSA) is sensitive to parameters tampering, as shown in [15, 16]. We can thus test a DSA implementation by observing an implementation's behavior upon invalid parameters. We notably discovered that even FIPS-certified libraries don't always satisfy the generic requirements of the digital signature standard (DSS) [FIPS186-4] regarding domain parameters and public key validation, which leads to generation and even validation of invalid signatures.

Our tests encompass generic invalid parameters, such as generator equal to 0 or 1, primes equal to 1, etc. You may find discussion of such issues in [15, 17].

An interesting pitfall of DSA is that, if the computation of either r or s fails and returns 0, then the algorithm is ran again with a new random value for k. This creates an infinite loop, and therefore unvalidated parameters may be used to DoS a signer.

In practice, domain parameters are usually hardcoded and rarely controllable by an attacker, yet FIPS-compliant implementations should implement that check to ensure protection even against the most unusual attack scenarios.

Note that malformed parameters already led to CVE-2016-3959[2], which impacted Go's DSA verification and allowed for remote DoS upon verification.

In CDF, we test the following DSA signature cases:

- $p = 0$ or $p = 1$, since it can cause $r = 0$, which can in turn trigger infinite loops in unsafe implementations
- $q = 0$ or $q = 1$, since it can cause either $r = 0$ or $s = 0$, which can trigger infinite loops on unsafe implementations
- $g = 0$, since it can cause $r = 0$, which can trigger infinite loops in unsafe implementations
- $g \equiv 1 \mod p$, since it causes $r = 1$, which may trigger always true signatures, if a key is generated with it and then used
- $x = 0$, since it should not be accepted as a valid private key and provides no security

And the following verification cases:

- $r = 1$ and $s = 0$, since if we (erroneously) compute $s^{-1} \mod q = 0$ as some implementations do, then $w = u_1 = u_2 = 0$ and $(g^{u_1} y^{u_2} \mod p) \mod q = 1$
- $r = 1$ and $s = q$, since the same holds
- $r = 0$ and $s = 1$, since the verification then only relies on the hash, however, notice that even if multiple messages might be crafted so that $H(m) \equiv 0 \mod q$, this remains difficult
- $g = 1$, since it may then validate wrong signatures, if the signature is crafted so that $r = 1$

There are other cases of degenerate behavior in unsafe implementations.

Go. The Go `crypto` package failed our tests in the following cases for DSA:

- If the generator g is congruent to 0 modulo p, then it led to infinite loops.
- If the generator g is congruent to 1 modulo p, then it leads to the generation of invalid keys and to the verification or creation of signatures whose value is entirely independent from the actual message.

After our report, the first issue was mitigated in Go 1.8.0, by limiting the number of attempts to 10.

OpenSSL. OpenSSL suffers from the same lack of checks regarding the domain parameters than Go's `crypto` did, except that none were patched. So it is possible to cause infinite loops upon signature operation and to make it generate always-valid signatures.

In addition, OpenSSL also accepts 0 as a valid private key, which is not a valid value as per [FIPS186-4], and proceed with it, generating signatures which are independent of the message signed for the public key of value 1.

[2] CVE-2016-3959 is described as "The Verify function in crypto/dsa/dsa.go in Go before 1.5.4 and 1.6.x before 1.6.1 does not properly check parameters passed to the big integer library, which might allow remote attackers to cause a denial of service (infinite loop) via a crafted public key to a program that uses HTTPS client certificates or SSH server libraries."

4.2 ECDSA

As with DSA, the standard [FIPS186-4] specifies validity checks for ECDSA implementations, in particular the check that points belong to the curve. Invalid curves attacks are mostly known for ECDH, however can also be a concern for ECDSA [17].

We specifically test the following signature cases:

- $x = 0$, $y = 0$ and $d = 0$, since it should not be accepted as a valid public key point and private integer
- $e = 0$, since it would then produce a signature whose value does not depend of the message

And the following verification cases:

- $(r, s) = (0, 0) = Q$, since if we (erroneously) compute $s^{-1} \mod n = 0$ as some implementations do, then $w = u1 = u2 = 0$ and the point $Q = (0, 0)$ would validate any message
- $H(m) = 0$, since then $z = 0$ and $u_1 = 0$, and carefully crafted (r, s) pairs could lead to wrong validation. This covers notably the cases where $z \equiv 0 \mod n$. Note that this test require the optional flag `-h` to be supported by the tested program
- $r = 1$ or $s = 1$ are also tested since those are invalid inputs and could lead to problems
- $Q = (0, 0)$, since it is not a valid public key point (see [1, 10])

Those tests allowed us to discover that both mbedTLS and Go were not performing proper checks on their ECDSA inputs.

Too, when the tested program supports the `-h` flag CDF does sign–verify checks with hashes of various length, from `minMsgLen` to `maxMsgLen`. This may result in interesting cases when the hash length is bigger than the group size. For instance in the deterministic ECDSA case, as described in [RFC6979], the hash length is to be truncated to the group size. Some implementations will then silently truncate the hash, without informing the caller that they provided an input of incorrect length (e.g. mbed TLS)

Go. Go's crypto doesn't verify that a point received belongs to the curve in ECDSA. While there exists currently no known invalid curve attacks against ECDSA, such checks should nonetheless be performed according to the standards, to ensure defense in depth, and to avoid (hardware) faults. Obviously, if the generator can be controlled by an attacker, in case of non-prime order curves or if the curve is also provided by the attacker, there are more dangerous subgroup attacks to worry about.

The private key integer can be set to 0, which is not within the possible values it should be allowed to take and signature operation will be carried on using that invalid value. This leads to infinite loops in the very improbable case where the hash is also equal to 0.

OpenSSL. Unlike Go, OpenSSL does verify that a point belongs tpo the curve. However, a private key set to 0 will be accepted, which will lead to infinite loops if the hash received is also equal to 0.

5 Conclusion

The automated testing tool CDF is a work in progress, which currently served to detect a number of issues in widely used crypto libraries. CDF is however limited in the number of primitives it supports—not even ECDH at this point—and in the number of tests it performs. Todos therefore include adding tests for existing interfaces, adding support for ECDH, as well as implementing a file-based interface.

We only tested popular, well-tested and robust libraries such as Go, OpenSSL, or Crypto++,so it would be interesting to test a larger set of libraries and APIs and compare their behavior with respect to CDF's tests.

References

[1] Jean-Philippe Aumasson. *Should Curve25519 keys be validated?* 2017.

[2] AWS Security Blog. *Automated Reasoning and Amazon s2n.* 2016. (Visited on 12/16/2016).

[3] Cas Cremers et al. "Automated Analysis and Verification of TLS 1.3: 0-RTT, Resumption and Delayed Authentication". In: *IEEE Symposium on Security and Privacy, SP 2016.* 2016.

[4] Robert Dockins et al. "Constructing Semantic Models of Programs with the Software Analysis Workbench". In: *Verified Software. Theories, Tools, and Experiments - 8th International Conference, VSTTE 2016, Toronto, ON, Canada, July 17-18, 2016, Revised Selected Papers.* 2016.

[5] Jason Donenfeld and Kevin Milner. *Formal Verification of the Wireguard protocol*. `https://www.wireguard.com/papers/wireguard-formal-verification.pdf`. 2017.

[6] Google Security Blog. *Project Wycheproof*. 2016. (Visited on 12/21/2016).

[7] John Harrison. "Formal Methods at Intel–An Overview". In: *Proceedings of the Second NASA Formal Methods Symposium (NFM 2010)*. Ed. by César Muñoz. 2010.

[8] Richard A. Kemmerer. "Analyzing Encryption Protocols Using Formal Verification Techniques". In: *EUROCRYPT*. 1986.

[FIPS186-4] Cameron F. Kerry. *Digital Signature Standard (DSS)*. National Institute of Standards and Technology (NIST), FIPS PUB 186-4, U.S. Department of Commerce. 2013.

[9] David Nowak. "On Formal Verification of Arithmetic-Based Cryptographic Primitives". In: *Information Security and Cryptology - ICISC 2008, 11th International Conference, Seoul, Korea, December 3-5, 2008, Revised Selected Papers*. Vol. 5461. LNCS. Springer, 2008.

[10] Trevor Perrin. *X25519 and zero outputs*. 2017.

[11] Anna Pogosyants and Roberto Segala. "Formal Verification of Timed Properties for Randomized Distributed Algorithms". In: *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, Ottawa, Ontario, Canada, August 20-23, 1995*. ACM, 1995.

[RFC6979] Thomas Pornin. *Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)*. RFC 6979 (Informational). Internet Engineering Task Force, Aug. 2013.

[12] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. *Dude, is my code constant time?* Cryptology ePrint Archive, Report 2016/1123. `http://eprint.iacr.org/2016/1123`. 2016.

[13] Benedikt Schmidt et al. "Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties". In: *25th IEEE Computer Security Foundations Symposium, CSF 2012*. 2012.

[14] Erik Seligman, Tom Schubert, and Madhunapantula V. Achutha Kiran Kumar. *Formal verification: an essential toolkit for modern VLSI design*. Morgan Kaufmann, 2015.

[15] Serge Vaudenay. "Evaluation report on DSA". In: *IPA work delivery* 1002 (2001).

[16] Serge Vaudenay. "Hidden collisions on DSS". In: *Annual International Cryptology Conference*. Springer. 1996.

[17] Serge Vaudenay. "The Security of DSA and ECDSA". In: *International Workshop on Public Key Cryptography*. Springer. 2003.