Extending NIST's CAVP Testing of Cryptographic Hash Function Implementations

Nicky Mouha and Christopher Celi

National Institute of Standards and Technology, Gaithersburg, MD, USA
nicky@mouha.be, christopher.celi@nist.gov

Abstract. This paper describes a vulnerability in Apple's CoreCrypto library, which affects 11 out of the 12 implemented hash functions: every implemented hash function except MD2 (Message Digest 2), as well as several higher-level operations such as the Hash-based Message Authentication Code (HMAC) and the Ed25519 signature scheme. The vulnerability is present in each of Apple's CoreCrypto libraries that are currently validated under FIPS 140-2 (Federal Information Processing Standard). For inputs of about 2^{32} bytes (4 GiB) or more, the implementations do not produce the correct output, but instead enter into an infinite loop. The vulnerability shows a limitation in the Cryptographic Algorithm Validation Program (CAVP) of the National Institute of Standards and Technology (NIST), which currently does not perform tests on hash functions for inputs larger than 65 535 bits. To overcome this limitation of NIST's CAVP, we introduce a new test type called the Large Data Test (LDT). The LDT detects vulnerabilities similar to that in CoreCrypto in implementations submitted for validation under FIPS 140-2.

Keywords: CVE-2019-8741, FIPS, CAVP, ACVP, Apple, CoreCrypto, hash function, vulnerability.

1 Introduction

The security of cryptography in practice relies not only on the resistance of the algorithms against cryptanalytical attacks, but also on the correctness and robustness of their implementations. Software implementations are vulnerable to software faults, also known as bugs.

A (cryptographic) hash function turns a message of a variable length into an output of a fixed length, often called a message digest, or digest. This fixed-length output can then serve as a "fingerprint" for the message, in the sense that it should be computationally infeasible to construct two messages that result in the same digest. Hash functions are crucial to the security of many higher-level cryptographic algorithms and protocols.

In the context of digital signature schemes, hash functions are used to ensure that only the given message and the corresponding signature (along with the public key) passes the signature verification process. Digital signatures provide authentication in a similar manner to signatures in the real world. For example, a web browser can verify a package that is downloaded comes from a specific

website by verifying the signature that was provided with the download using the known, trusted public key of the website. As a part of this verification process, the browser hashes the downloaded data so that the fixed-length digest can stand in place of the large variable-length data in the digital signature scheme.

A recent study by Mouha et al. [12] of the National Institute of Standards and Technology (NIST) SHA-3 (Secure Hash Algorithm) competition found that about half of the implementations submitted to the SHA-3 competition contained bugs, including two out of the five finalists. It appears that cryptographic algorithms can be difficult to implement, given that even the designers of the algorithm can have trouble to develop a correct implementation. Furthermore, even for a secure and well-designed cryptographic algorithm, bugs can be particularly severe with respect to the cryptographic properties of the algorithm's implementation.

For example, in the case of all submitted implementations of the BLAKE [4] algorithm to the SHA-3 competition, given one message and its corresponding hash function output, it is easy to construct another message that produces the same hash value. This "second preimage attack" is not due to a weakness in the BLAKE algorithm specification, but due to an implementation bug that remained undiscovered for seven years.

In [12], Mouha et al. did not find any bugs in the submission packages of Keccak [6], the hash function algorithm that won the SHA-3 competition and that is now standardized in Federal Information Processing Standard (FIPS) 202 [17]. In this paper, we explore whether implementations of hash functions that are standardized by NIST and currently used in commercial products may also contain bugs. Furthermore, we investigate how these bugs can impact more complex cryptographic operations such as digital signature schemes.

2 Testing within NIST's CAVP

NIST maintains the Cryptographic Algorithm Validation Program (CAVP), which provides validation testing for the NIST-recommended cryptographic algorithms. The CAVP is a prerequisite for validating cryptographic implementations according to FIPS 140-2 under the Cryptographic Module Validation Program (CMVP). Since the Federal Information Security Management Act (FISMA) of 2002, U.S. Federal Agencies no longer have a statutory provision to waive FIPS 140-2. This means that commercial vendors must validate their cryptographic implementations, also known as modules, according to CAVP/CMVP before they can be deployed by U.S. Federal Agencies.

The CAVP testing methodology is derived directly from the algorithm specification, independent of the actual code that a vendor's implementation uses. Therefore, it is realistic to expect three main limitations of the CAVP:

1. The CAVP does not require that the internals of an implementation are known in order to generate tests, and is therefore restricted to black-box testing. For many widely-used cryptographic libraries, however, the software

is either open source or available on the vendor's website, which may be used to reveal additional bugs through static analysis (including checking software coding standards), or white-box testing.

2. The CAVP tests only the capabilities of the implementation that are declared by the vendor. For example, a hash function implementation may declare that it can only process messages up to 65 535 bits, corresponding to the largest test vectors currently in the CAVP, even though it may encounter much larger inputs under typical use. When NIST introduces tests for larger inputs, it is therefore the vendor's responsibility to declare whether or not their implementation supports such inputs. However, it is in the vendor's interest to avoid bugs and therefore declare the capabilities of the implementations as broadly as possible.
3. The CAVP focuses mostly on the correct processing of valid inputs (positive testing), rather than the rejection of invalid inputs (negative testing). Because of the nature of black-box testing, the CAVP provides test vector data to the implementation. A developer of the module must program a test harness to submit this data to the interfaces of the cryptographic library itself and collect the output to send back to the CAVP. As the test harness is outside the bounds of the CAVP, it is difficult to know from a validation perspective whether invalid inputs are handled by the module, or by the test harness. There are a few notable exceptions to this, such as the CAVP tests for digital signature schemes that test whether the implementation can recognize valid versus invalid signatures.[1]

Furthermore, the focus of most cryptographic algorithm testing is on correctness towards common cases within the specification. This may leave cryptographic algorithms vulnerable to malicious inputs that manifest themselves very rarely under random testing. Notable examples exploit bugs in modular arithmetic [7], incorrect group order validation [21], or improper primality testing [1] to result in full or partial key recovery attacks on OpenSSL and other implementations. These examples show the importance to consider not just random but also "rare" and "unusual" inputs for cryptographic implementations, as they may lead to catastrophic security failures.

In spite of these limitations, the CAVP can be highly effective at detecting many types of bugs. This is because the CAVP test design is aware of the internals of "typical" implementations of cryptographic algorithms. The focus of the CAVP is not just conformance testing but also regression testing, as the CAVP test design is also aware of how changes to the implementations may lead to certain bugs. To see this, we now explain how the CAVP tests are generated.

The two test types in the CAVP are the Algorithm Functional Test (AFT), and the Monte Carlo Test (MCT). They were introduced in 1977 by the National Bureau of Standards (NBS), the former name of NIST, in the (now-withdrawn) Special Publication (SP) 500-20 [13] to test the Data Encryption

[1] For the signature verification operation, the CAVP also includes some invalid padding tests.

Standard (DES). In this standard, static AFTs known as Known Answer Tests (KATs) were provided in order to “fully exercise the non-linear substitution tables” (S-boxes), whereas MCTs contained “pseudorandom data to verify that the device has not been designed just to pass the [fixed] test set.” Additionally, the large amount of data of the MCT was intended to detect whether it can “cause the device to hang or otherwise malfunction,” for example due to a memory leak [8] in present-day implementations. The spirit and design of these tests was carried over to other algorithms such as the Advanced Encryption Standard (AES) in FIPS 197 [14] and hash functions.

This paper focuses on testing for hash functions within the CAVP at NIST. FIPS 180-4 [16] standardizes the hash functions SHA-1, SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, and SHA-512/256. As these hash functions closely resemble each other, they are considered functionally equivalent for the purpose of this document. Testing for SHA-3 was added after the publication of FIPS 202 [16], and with the exception of the SHAKE extendable-output functions (XOFs), mimics the testing done for the FIPS 180-4 hash functions. As with the other CAVP tests, the Secure Hash Algorithm Validation System (SHAVS) [5] specifies both AFTs and MCTs.

Testing by the CAVP was done for many years using the Cryptographic Algorithm Validation System (CAVS) tool. An implementation under test (IUT) is accompanied with a declaration to the CAVS tool of which digest sizes it supports along with a couple of other properties such as whether or not it can hash an empty message, whether or not it can hash incomplete bytes (i.e. a 7-bit message), and the maximum message size. The maximum message size allowed by the tool is 65 535 bits.

As of 2019, the CAVP is undergoing a transition to use the Automated Cryptographic Validation Protocol (ACVP) to enable the generation and validation of standardized algorithm test vectors. This involves a shift of generating and validating tests at remote, approved laboratories, to performing these actions on NIST-hosted servers. The concept of first-party testing is introduced to allow vendors to test and validate their implementations without laboratories as intermediaries. This combined with hosting a demo server (a sandbox environment for algorithm testing), allows vendors to incorporate continuous testing of crypto implementations in their development process. The ACVP thereby significantly speeds up testing and validation.

The ACVP uses a JSON (JavaScript Object Notation) format to specify the test cases. The client to the NIST ACVP servers would then correspond to the test harness in the previous CAVS model, and is responsible for communicating with the server and exercising the proper interfaces on the module. In the JSON examples below, some of the original content has been trimmed for readability. For more information on the protocol itself, as well as the complete examples, we refer to the GitHub repository of the ACVP [11].

2.1 Algorithm Functional Test (AFT)

AFTs take a single message as input, and verify the correctness of the corresponding output. A JSON file is sent from the server to the client, which usually provides inputs to a cryptographic algorithm, and is very simple for an individual test case:


```
{
 "msg": "BCE7",
 "len": 16
}
```


where "msg" corresponds to the message represented as hexadecimal, and "len" corresponds to the length in bits of the message. The messages have fixed values that have been drawn uniformly at random from the space of messages of a certain bit length, ranging from the client's specified minimum to their specified maximum or 65 535, whichever comes first. The expected response to this test case is another simple JSON object:


```
{
 "md": "1FA29E9B23060562F9370453EF817E18C56AE844E5B85F2ED34B4B38"
}
```


where "md" corresponds to the message digest. The hash function in this example is SHA-224.

AFTs can vary in length from byte-oriented messages (i.e., "len" is a multiple of 8) or bit-oriented messages (with any bit lengths). This allows implementations to specify their properties to the CAVP to receive appropriate test cases.

These tests are intended to provide assurance that an implementation can handle messages of various sizes. However, the assurance that the AFTs currently offer may be limited, as they may not test more than one message of any specific bit length.

2.2 Monte Carlo Test (MCT)

MCTs, on the other hand, construct a chain of hash outputs by combining the previous three hash outputs into a single message, and use it to produce the next hash output. Each chain consists of 1000 iterations, and returns the hash output that is obtained at the end. This whole process is repeated 100 times with the original message replaced by the latest hash output.

The initial condition for an MCT is as follows:


```
{
 "msg": "B4FCB616B3A4A7C9E6AF1D836CF1576709A67F16141217B827E52611",
 "len": 224
}
```

where `"msg"` becomes the `seed` in the pseudocode of the MCT, which is given in Alg. 1. The `seed` is not fixed, but is drawn uniformly at random for every invocation of the test.

Algorithm 1 The Monte Carlo Test (MCT) for hash functions

Require: seed (random string of same length as hash output)
  for $i = 1$ to 100 **do**
    MD[0] = MD[1] = MD[2] = seed;
    for $j = 3$ to 1002 **do**
      Msg[j] = MD[$j-3$] $\|$ MD[$j-2$] $\|$ MD[$j-1$];
      MD[j] = Hash(Msg[j]);
    end for
    seed = MD[1002];
    Output seed;
  end for

The response is an array of 100 hash outputs as follows:


```
{
"resultsArray": [
 {
  "md": "7B893BC7322AA6578A2EC565593B86776FB8376AC16B0A354E6DA016"
 },
 {
  "md": "4BCB655F36D976ADAAE620B485DA7FD8ED321E0BF060E0FE2B5F9AFE"
 },
 {
  "md": "57AA388954B3D52645BFAC69E87F48B3D57A86CF385F38A2549FE957"
 }
]
}
```


shortened to only three outputs for brevity, and again using the SHA-224 hash function in the example. The CAVP makes an implicit assumption here that the client's implementation can handle a message that is three times the size of the hash output.

These tests are intended to provide assurance that an implementation is correct for valid inputs over thousands of iterations. However, the assurance that the MCTs currently offer may be limited, as the bit lengths of the messages do not vary between test cases. Furthermore, as this bit length is three times the digest size, the MCTs only cover a negligibly small percentage of the total input space of the given bit length.

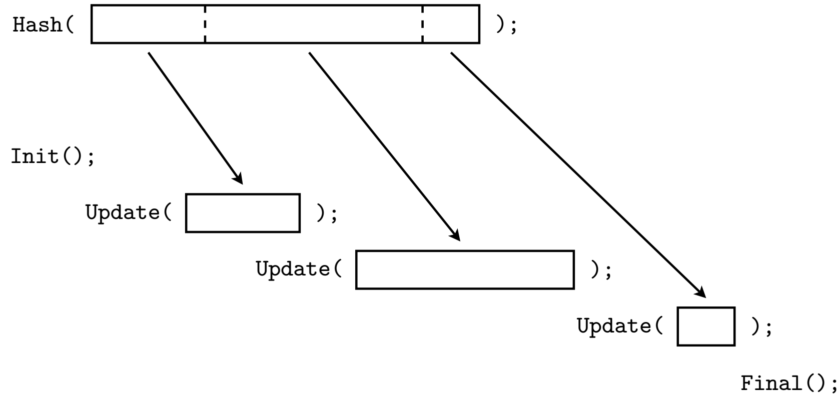

Fig. 1. Hash functions are commonly implemented using a `Hash` interface that takes a variable-length message, and returns a fixed-length output. It is common to also have an `Init-Update-Final` interface, which can be convenient to process large messages on the fly.

3 Common Hashing Interfaces

Although not mentioned in the NIST hash function standards [16, 17], many cryptographic implementations have at least two distinct functional interfaces for hash operations, as shown in Fig. 1. One of the two interfaces, or both interfaces, may be available to a consumer of the module or to higher-level algorithms within the module. The first is an `Init-Update-Final` interface. This structure allows implementations to constantly stream smaller chunks of data into `Update()` repeatedly, rather than keep the message as a single large chunk. Perhaps the entire message is not available at once, or perhaps there is a limit to the capacity of a single `Update()` call.

The other interface is a more intuitive `Hash()` call that expects the whole message up front. This is different from the previous interface and the same module could potentially behave differently under these two interfaces [12].

In practice, the `Init-Update-Final` interface can be convenient to hash the concatenation of various elements. For example, the American National Standards Institute (ANSI) X9.63 Key Derivation Function (KDF) [2] computes the hash of a secret value Z, a counter, and an optional `SharedInfo` string that is shared between two entities. This hash can be computed using one `Init()` call, followed by an `Update()` call to process Z, another `Update()` call for the counter, and then an optional third `Update()` call for `SharedInfo`. The `Final()` call can then be used to compute the hash function output.

To hash the contents of a file, there are two approaches that are commonly encountered in practice. One approach is to loop through the contents of the file (e.g., using `fread()` in C), and process each chunk using a call to `Update()`. Another common approach is to map the file to the virtual address space (e.g.,

using mmap() in C), and then compute the hash by calling Hash(). This second approach must be used when the interface requires the data to be located in memory. For example, the interface of the Ed25519 signature scheme in Apple's CoreCrypto requires a pointer for the data to be hashed, therefore if an application wants to compute (or verify) a digital signature on a file (e.g., containing a large software update), it must first use mmap() to map this file into memory.

4 Vulnerability in Apple's CoreCrypto Library

We show how adding test cases beyond the current coverage of the CAVP can reveal previously undiscovered bugs in cryptographic implementations.

First, we look the SHAVS document [5], which states that:

> *"While the specification for SHA specifies that messages up to at least $2^{64} - 1$ bits are possible, these tests only test messages up to a limited size of approximately 100,000 bits. This is adequate for detecting algorithmic and implementation errors."*

In contrast, the SHA-3 Competition Test Suite [15] also contains an "Extremely Long Message Test," which contains a message of 2^{33} bits (1 GiB), with the intention of checking whether messages of more than 2^{32} bits were processed correctly. This test from the SHA-3 competition is not adopted by the CAVP however. We now explain how adding a similar test for large messages reveals a bug in the widely-used Apple CoreCrypto library.

Apple makes the source code of its CoreCrypto library publicly available [3] to allow for "verification of its security characteristics and correct functioning."[2] The CoreCrypto library provides low-level cryptographic primitives that are fundamental to the security of Apple's products, and is currently deployed in iPhone, iPad, and Mac devices worldwide. The library has also undergone rigorous testing, and is currently present in 20 FIPS 140-2-validated modules.

In the latest CoreCrypto library, the bug is present in the ccdigest_update.c file, which is located in the ccdigest/src subdirectory. This code is shared by all implemented hash functions except for MD2. The full code of the function is given in App. A. All the implemented hash functions are iterated hash functions, which means that an underlying compression function processes the message in multiples of a block size that is specific to the algorithm. Part of the code to process message in multiples of the block size is as follows:

```
1  //low-end processors are slow on division
2  if (di->block_size == 1<<6 ){ //sha256
3    nblocks = len >> 6;
4    nbytes = len & 0xFFFFffC0;
5  } else if(di->block_size == 1<<7 ){ //sha512
```

[2] We refer to the latest CoreCrypto that is available online at the time of writing (November 25, 2019). It does not appear to have a version number, but can be identified by the year 2018 in the copyright notice.

```
6    nblocks = len >> 7;
7    nbytes = len & 0xFFFFff80;
8  } else {
9    nblocks = len / di->block_size;
10   nbytes = nblocks * di->block_size;
11 }
```

In this code, the variables `len`, `nblocks`, and `nbytes` are declared as `size_t`, which corresponds to a 64-bit unsigned integer on a 64-bit architecture. The `len` variable is the length of the message in bytes. In case `len` is less than 2^{32}, the value of `nblocks` is the number of complete blocks to be hashed: `len` divided by the block size (in bytes), whereas `nbytes` is the number of bytes of these complete blocks.

However, for block sizes of 64 or 128 bytes (i.e., when `di->block_size` is `1<<6` or `1<<7`), the calculation of `nbytes` contains a bug: the four highest bytes of `size_t` are incorrectly set to zero by the bitwise AND (`&`) operation. Consequently, when `len` is at least 2^{32} (corresponding to messages of at least 4 GiB), the value of `nbytes` does not contain the correct number of complete blocks. Therefore, later in the code, the statement `len -= nbytes` does not decrement `len` by the correct amount; instead `len` remains 2^{32} or larger. Given that all these statements are contained in a while-loop with condition `len > 0`, the program enters into an infinite loop.

A list of affected hash function implementations is given in Table 1.

Table 1. Hash function implementations in Apple's CoreCrypto library.

Algorithm	Block size (in bytes)	vulnerable
MD2	16	✗
MD4	64	✓
MD5	64	✓
RIPEMD-128	64	✓
RIPEMD-160	64	✓
RIPEMD-256	64	✓
RIPEMD-320	64	✓
SHA-1	64	✓
SHA-224	64	✓
SHA-256	64	✓
SHA-384	128	✓
SHA-512	128	✓

When this code was written, perhaps the assumption was made that `size_t` corresponds to a 32-bit value, in which case the code would have been correct. When `size_t` is 64 bits, however, the integer constant used to perform the AND operation is incorrect.

One way to avoid this type of bug, could be to follow software coding standards, such as the Computer Emergency Response Team (CERT) C Coding

Standard. This standard states in INT17-C: "Define integer constants in an implementation-independent manner" [19], and gives an example that is very similar to the bug in Apple's CoreCrypto library. Note that it is possible to avoid masks altogether, by using `nbytes = nblocks << 6` or `nbytes = nblocks << 7` for 64-byte and 128-byte blocks respectively.

4.1 Experimental Verification

We downloaded the latest CoreCrypto library from Apple's website [3], and compiled it using the Xcode IDE (Integrated Development Environment) on macOS 10.14 (Mojave) on a mid 2015 MacBook Pro, as well as using Clang 8 under Ubuntu 14.04 on an Intel Skylake processor. For Linux, the README.md file warns that the Linux Makefile is not up-to-date, therefore we needed to make some minor changes to the Makefile to allow compilation.

Because the bug is due to incorrect C code, we expect that the bug will manifest itself on any 64-bit platform for which the code is compiled. To confirm that the executable is stuck in an infinite loop, we added some source code instrumentation.

In our proof of concept code, we generated an input with a length of 2^{32} bytes. Because the actual value of the input is not relevant for the bug, we arbitrarily set all bits to zero in our experiments. When this input is provided to MD4, MD5, RIPEMD-128, RIPEMD-160, RIPEMD-256, RIPEMD-320, SHA-1, SHA-224, SHA-256, SHA-384, or SHA-512, we verified that the implementation enters into an infinite loop. We mentioned earlier that the MD2 implementation does not share the code of `ccdigest_update.c`, and we also confirmed that the same input does not cause an infinite loop for MD2. This provides experimental confirmation for the results of Table 1.

Then, we looked into higher-level cryptographic operations. We found that the implementation of the ANSI X9.63 KDF is not vulnerable when provided with a secret value Z of length 2^{32} bytes. This is due to a range check in the input length, which is documented by the following source code comment in CoreCrypto: "`ccdigest_update only supports 32bit length`."

However, such a range check is not applied to every hash function calculation, and most other cryptographic algorithms inside Apple's CoreCrypto library that use hash functions are vulnerable. We verified that HMAC enters into an infinite loop for all the vulnerable algorithms in Table 1 when provided with a message of 2^{32} bytes.

For the Ed25519 signature scheme, we found that a message of at least $2^{32} +$ 64 bytes is needed to trigger the bug. To explain this, note that the Ed22519 algorithm always prepends some data to the message before computing the hash value using SHA-512. This is implemented in Apple's CoreCrypto using the `Init-Update-Final` interface. When there are 64 bytes already in the buffer, the first 64 bytes of the message are used to complete a 128-byte block, which we recall is the block size for the SHA-512 algorithm. After processing the first 64 bytes of the message, if there are at least 2^{32} bytes or more left, then the

bug is triggered. For details, we refer to the full code of the `ccdigest_update()` function in App. A.

We verified that the Ed25519 implementation indeed enters into an infinite loop when a message of $2^{32} + 64$ bytes is digitally signed or verified. Note that in order to trigger the bug in the verification operation, it is not necessary to provide a valid signature. Therefore, even if the private key is stored properly and never used to sign long messages, the verification operation still enters into an infinite loop for an incorrectly-signed message of $2^{32} + 64$ bytes or more. Note that digitally signed messages typically come from untrusted sources, because the concern that a message can be modified by an adversary is typically the reason to apply a digital signature in the first place.

Another cryptographic operation in Apple's CoreCrypto that uses hash functions, is the Secure Remote Password (SRP) protocol. This protocol is run between a client and a server, which can create additional security concerns when communication is done over a network and the adversary controls either the client or the server, and may therefore send malicious inputs. In CoreCrypto's SRP implementation, the username is provided as a null-terminated string.

We verified that when this string contains 2^{32} repetitions of the `'a'` character followed by a null character, then the SRP implementation of both the client and the server enter into an infinite loop. Note that in contrast to the previous examples, the length in this case is not provided by the adversary as a separate parameter, but it is derived inside CoreCrypto using C's `strlen()` function. Therefore, range checking all input length values to CoreCrypto would not have been effective to avoid this attack using a long null-terminated string.

In Sect. 2, we recalled that an input that would "cause the device to hang" was already a concern when the MCT test was introduced for DES in 1977. But an infinite loop is also a security vulnerability, categorized under Common Weakness Enumeration (CWE) 835 [20], where it is also known as a "Loop with Unreachable Exit Condition." More specifically, an adversarially-crafted input that causes an implementation to enter an infinite loop, can lead to a "denial of service" (DoS) attack when it consumes excessive CPU resources.

5 Proposing the Large Data Test (LDT)

In the current CAVP tests, the length of the largest message is 65 535 bits. Such small testing sizes are not realistic towards normal usage. We propose a new Large Data Test (LDT) for the CAVP to provide a greater assurance for the implementations that undergo validation.

The LDT would be a type of AFT, and could be specified similarly to the example in Sect. 2.1. Implementations could specify the size of the largest message size that they can handle, for example on the order of 2 GiB to 8 GiB. The ACVP server can select one of many large supported arbitrary sizes to craft messages. However, a test for such messages may be impractical to communicate natively within the normal JSON structures. To work around this limitation, the LDT employs a simple function to generate the test input, as defined in Alg. 2.

Algorithm 2 The Large Data Test (LDT)

Require: Msg (a non-zero number of bytes), fullLength (in bits)
  FullMsg = "";
  for i = 0 to ceil(fullLength / bitlength(Msg)) **do**
    FullMsg = FullMsg || Msg;
  end for
  FullMsg = truncate(FullMsg, fullLength);
  Output FullMsg;

Due to the truncation at the end, it is possible for the LDT to output messages of any number of bits, instead of only multiples of the size of the repeating `Msg` pattern. The `Msg` pattern itself needs to be an integer number of bytes, in order to greatly simplify implementations in C-like programming languages. This is, however, not an actual restriction to the messages that can be output. The reason is that any 7-bit repeating pattern (for example) can also be written as a 56-bit (= 7-byte) repeating pattern, where 56 is the least common multiple of 7 and 8 (the number of bits in one byte).

With a generator function defined to expand a short message of a few bytes, into a large message of any arbitrary size, we can define the JSON structure for the LDT as the following:

```
{
  "largeMsg": {
    "content": "D6F7",
    "contentLength": 16,
    "fullLength": 34359738368,
    "expansionTechnique": "repeating"
  }
}
```

We define an `"expansionTechnique"` to allow extensibility in the future for other methods of producing a message of the proper size. In this example `"repeating"` corresponds to the repeating nature of Alg. 2.

After the test generates a message of a specific number of bits, this message would then be hashed on the server to produce a single hash output similar to the AFTs. Once the test is sent to the client, this could flush out implementations for faults from long messages that produce incorrect outputs. As hashing is a core operation to many other cryptographic operations, it is important to consider scenarios where an adversary may maliciously generate large inputs.

Note that to unearth the bug in the Apple CoreCrypto library, it is necessary to use either the `Hash()` interface on a message of 4 GiB or more, or the `Init-Update-Final` interface where at least one of the `Update()` calls contains 4 GiB or more. In the latter case, it may be necessary to make the message a few bytes longer, as explained in Sect. 4.1.

Given that the LDT is designed to work with large data, we need to take into consideration that the implementation may run out of memory. When allocating

dynamic memory (e.g., using `malloc()` in C) or mapping files to the virtual address space (e.g., using `mmap()` in C) are unsuccessful on the target platform, it may be an option to consider increasing the memory available to the platform or even simulating the environment for the purposes of testing.

6 Discussion

As hash functions are a core primitive within many other cryptographic algorithms, it is critically important to ensure correctness under all valid inputs. Yet the methods with which these algorithms are tested are still based on techniques from 1977. While the original tests are still valid, an automated system allows the CAVP to continually add test types and boost the assurances gained from the program. With a publicly standardized JSON protocol, and open-source test harnesses such as libacvp [9], the CAVP is in a good position to move forward with improved testing techniques. We suggest the LDT as a way to directly improve the assurances gained from the CAVP. Of course, one needs to design, specify, publicly review the tests, etc. before they can be used in a program such as CAVP. Openness and transparency are important for acceptance in this highly sensitive domain.

To test the limits of common variable types such as 32-bit unsigned integers, the LDT would need to be on the order of 2^{32} bytes or 4 GiB. This would be sufficient to detect the CoreCrypto bug, and potentially similar bugs in other cryptographic implementations.

However, an inherent limitation of the CAVP and of software testing in general, is that it is a selection process, where a very small subset is selected from the total number of possible test cases. Therefore, testing is not a method to prove the correctness over all types of inputs for an implementation. As stated by Dijkstra, "Program testing can be used to show the presence of bugs, but never to show their absence!" Indeed, the entire goal of software testing is to determine how to perform this selection process, in order to try to quantify the assurance that we get from testing.

Furthermore, the CAVP only tests the capabilities that are declared by the vendor, and would therefore not detect the bug if it only declares support for short messages. While this is reflected in the final validation certificate the vendor receives, this shows the potential need for a wider amount of negative testing. Negative tests are those that test not only well-defined inputs that may be beyond the advertised capabilities, but also invalid inputs.

We note the potential hazards of exposing multiple entry points to a single set of functionality. As mentioned, hash functions often provide at least two interfaces: an `Init-Update-Final` interface and a `Hash()` interface. Often both are exposed such as within CoreCrypto.

Lastly, it can be interesting to explore the parallels between different levels at which vulnerabilities can be handled, as we now explain.

A security vulnerability report to the vendor can allow for a rapid response to address a vulnerability. The FIPS 140-2 Implementation Guidance (IG) [18]

encourages this process by providing the vendors with a "means to quickly fix, test and revalidate a module that is subject to a security-relevant CVE." A CVE (Common Vulnerability and Exposure) is security-relevant if it affects how the module meets the requirements of the FIPS 140-2 standard.

For FIPS 140-2 validated cryptographic modules, publishing a vulnerability with a CVE can accelerate the time for end users to obtain crucial security updates. Yet the very nature of the CVE system is an ad hoc procedure, and there is no mechanism in place to ensure that a vendor has learned from such a vulnerability. A vendor may implement test cases within their own development process to detect similar issues in the future, but this holds a very limited scope. The implementations of other vendors could be susceptible to similar issues, but there may be no incentive to react.

If the CAVP implements tests based on CVEs (e.g., as done by Project Wycheproof [10]), then lessons learned from a CVE are not restricted to a single implementation. The requirement of FIPS validation would then also provide stronger assurances to government and private entities that rely on the program. If a CVE can be detected via existing test types, a static test could be seamlessly included from the NIST server. By using an existing test type, no additional code is needed from a test harness to understand how to process the test. In addition, with the speed of testing under ACVP, it is mutually beneficial to constantly test while developing cryptographic implementations.

7 Conclusion

Apple's CoreCrypto library contains a bug due to the implementation-dependent manner in which integer constants are specified. Due to this bug, the MD4, MD5, and the RIPEMD and SHA family hash function implementations enter into an infinite loop for messages of 4 GiB or larger. The bug affects all implemented hash functions (except MD2), and higher-level operations such as HMAC, Ed25519, and SRP. To detect the bug in NIST's CAVP, we proposed a new Large Data Test (LDT) to calculate the hash value for large messages. We also pointed out that stricter coding standards might be helpful to avoid this type of bug.

Responsible Disclosure. The Apple Product Security team was notified of the vulnerability described in this paper on May 30, 2019, and has since taken steps to address the issue. In a conference call on July 17, 2019, Apple Product Security clarified that they do not object to the publication of the research results presented in this paper. On July 23, 2019, Apple Product Security informed us that they are planning to assign a CVE to this issue. On October 29, 2019, Apple publicly announced CVE-2019-8741 to address the vulnerability described in this paper for macOS Catalina 10.15, tvOS 13, watchOS 6, iOS 13, iTunes 12.10.1 for Windows, and iCloud for Windows 7.14.

Acknowledgments. The authors would like to thank the anonymous reviewers and their NIST colleagues for providing useful comments and suggestions. Spe-

cial thanks go to Patrick Kamongi, Andrew Regenscheid, Apostol Vassilev, and Jeffrey Marron for their detailed feedback. Certain algorithms and commercial products are identified in this paper to foster understanding. Such identification does not imply recommendation or endorsement by NIST, nor does it imply that the algorithms or products identified are necessarily the best available for the purpose.

References

1. Albrecht, M.R., Massimo, J., Paterson, K.G., Somorovsky, J.: Prime and Prejudice: Primality Testing Under Adversarial Conditions. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018. pp. 281–298. ACM (2018), https://doi.org/10.1145/3243734.3243787
2. American National Standards Institute: Public Key Cryptography for the Financial Services Industry - Key Agreement and Key Transport Using Elliptic Curve Cryptography. ANSI X9.63 (2017), https://webstore.ansi.org/standards/ascx9/ansix9632011r2017
3. Apple: Security - Apple Developer (September 2019), https://developer.apple.com/security/
4. Aumasson, J.P., Henzen, L., Meier, W., Phan, R.C.W.: SHA-3 proposal BLAKE. Submission to the NIST SHA-3 Competition (Round 3) (2010), http://131002.net/blake/blake.pdf
5. Bassham III, L.E., Hall, T.A.: The Secure Hash Algorithm Validation System (SHAVS) (May 2014), https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Algorithm-Validation-Program/documents/shs/SHAVS.pdf
6. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: The Keccak SHA-3 submission. Submission to the NIST SHA-3 Competition (Round 3) (2011), http://keccak.noekeon.org/Keccak-submission-3.pdf
7. Brumley, B.B., Barbosa, M., Page, D., Vercauteren, F.: Practical Realisation and Elimination of an ECC-Related Software Bug Attack. In: Dunkelman, O. (ed.) Topics in Cryptology - CT-RSA 2012 - The Cryptographers' Track at the RSA Conference 2012, San Francisco, CA, USA, February 27 - March 2, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7178, pp. 171–186. Springer (2012), https://doi.org/10.1007/978-3-642-27954-6_11
8. Celi, C.: ACVP Secure Hash Algorithm (SHA) JSON Specification. IETF Internet-Draft (2018), https://usnistgov.github.io/ACVP/artifacts/draft-celi-acvp-sha-00.html
9. Cisco: The libacvp library (September 2019), https://github.com/cisco/libacvp
10. Google: Project Wycheproof tests crypto libraries against known attacks. (September 2019), https://github.com/google/wycheproof
11. Industry Working Group on Automated Cryptographic Algorithm Validation: ACVP (September 2019), https://usnistgov.github.io/ACVP/
12. Mouha, N., Raunak, M.S., Kuhn, D.R., Kacker, R.: Finding Bugs in Cryptographic Hash Function Implementations. IEEE Trans. Reliability **67**(3), 870–884 (2018), https://doi.org/10.1109/TR.2018.2847247
13. National Bureau of Standards: Validating the Correctness of Hardware Implementations of the NBS Data Encryption Standard. NBS Special Publication 500-20 (November 1977), https://doi.org/10.6028/NBS.SP.500-20e1977

14. National Institute of Standards and Technology: Advanced Encryption Standard (AES). NIST Federal Information Processing Standards Publication 197 (November 2001), https://doi.org/10.6028/NIST.FIPS.197
15. National Institute of Standards and Technology: Description of Known Answer Test (KAT) and Monte Carlo Test (MCT) for SHA-3 Candidate Algorithm Submissions (February 2008), https://csrc.nist.gov/CSRC/media/Projects/Hash-Functions/documents/SHA3-KATMCT1.pdf
16. National Institute of Standards and Technology: Secure Hash Standard (SHS). NIST Federal Information Processing Standards Publication 180-4 (August 2015), http://dx.doi.org/10.6028/NIST.FIPS.180-4
17. National Institute of Standards and Technology: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. NIST Federal Information Processing Standards Publication 202 (August 2015), https://doi.org/10.6028/NIST.FIPS.202
18. National Institute of Standards and Technology and Canadian Centre for Cyber Security: Implementation Guidance for FIPS 140-2 and the Cryptographic Module Validation Program (August 2019), https://csrc.nist.gov/CSRC/media/Projects/cryptographic-module-validation-program/documents/fips140-2/FIPS1402IG.pdf
19. SEI CERT C Coding Standard: INT17-C. Define integer constants in an implementation-independent manner (September 2019), https://wiki.sei.cmu.edu/confluence/display/c/INT17-C.+Define+integer+constants+in+an+implementation-independent+manner
20. The MITRE Corporation: CWE-835: Loop with Unreachable Exit Condition ('Infinite Loop') (2019), https://cwe.mitre.org/data/definitions/835.html
21. Valenta, L., Adrian, D., Sanso, A., Cohney, S., Fried, J., Hastings, M., Halderman, J.A., Heninger, N.: Measuring small subgroup attacks against Diffie-Hellman. In: 24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017. The Internet Society (2017), https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/measuring-small-subgroup-attacks-against-diffie-hellman/

A The `ccdigest_update()` function of Apple's CoreCrypto

Here, we provide the implementation of the `ccdigest_update()` in Apple CoreCrypto, which is made available to the public on Apple's website [3]. For readability, we made minor changes to the indentation, corrected the spelling of the word "division" and expanded the `CC_MEMCPY` macro to `memcpy`.

```
1  void ccdigest_update(const struct ccdigest_info *di, ccdigest_ctx_t ctx,
2                       size_t len, const void *data) {
3    const char * data_ptr = data;
4    size_t nblocks, nbytes;
5
6    while (len > 0) {
7      if (ccdigest_num(di, ctx) == 0 && len > di->block_size) {
8        //low-end processors are slow on division
9        if (di->block_size == 1<<6 ){ //sha256
10         nblocks = len >> 6;
```

```
11          nbytes = len & 0xFFFFffC0;
12        } else if(di->block_size == 1<<7 ){ //sha512
13          nblocks = len >> 7;
14          nbytes = len & 0xFFFFff80;
15        } else {
16          nblocks = len / di->block_size;
17          nbytes = nblocks * di->block_size;
18        }
19
20        di->compress(ccdigest_state(di, ctx), nblocks, data_ptr);
21        len -= nbytes;
22        data_ptr += nbytes;
23        ccdigest_nbits(di, ctx) += nbytes * 8;
24      } else {
25        size_t n = di->block_size - ccdigest_num(di, ctx);
26        if (len < n)
27          n = len;
28        memcpy(ccdigest_data(di, ctx) + ccdigest_num(di, ctx), data_ptr, n);
29        /* typecast: less than block size, will always fit into an int */
30        ccdigest_num(di, ctx) += (unsigned int)n;
31        len -= n;
32        data_ptr += n;
33        if (ccdigest_num(di, ctx) == di->block_size) {
34          di->compress(ccdigest_state(di, ctx), 1, ccdigest_data(di, ctx));
35          ccdigest_nbits(di, ctx) += ccdigest_num(di, ctx) * 8;
36          ccdigest_num(di, ctx) = 0;
37        }
38      }
39    }
40  }
```