Exploring Formal Methods for Cryptographic Hash Function Implementations

Nicky Mouha

Strativia, Largo, MD, USA
nicky@mouha.be

Abstract. Cryptographic hash functions are used inside many applications that critically rely on their resistance against cryptanalysis attacks and the correctness of their implementations. Nevertheless, vulnerabilities in cryptographic hash function implementations can remain unnoticed for more than a decade, as shown by the recent discovery of a buffer overflow in the implementation of SHA-3 in the eXtended Keccak Code Package (XKCP), impacting Python, PHP, and several other software projects. This paper explains how this buffer overflow vulnerability in XKCP was found. More generally, we explore the application of formal methods to the five finalist submission packages to the NIST SHA-3 competition, allowing us to (re-)discover vulnerabilities in the implementations of Keccak and BLAKE, and also discover a previously undisclosed vulnerability in the implementation of Grøstl. We also show how the same approach rediscovers a vulnerability affecting 11 out of the 12 implemented cryptographic hash functions in Apple's CoreCrypto library. Our approach consists of removing certain lines of code and then using KLEE as a tool to prove functional equivalence. We discuss the advantages and limitations of our approach and hope that our attempt to consolidate some earlier approaches can lead to new insights.

Keywords: SHA-3 · Hash Function · Keccak · BLAKE · Grøstl · CoreCrypto

1 Introduction

A (cryptographic) hash function takes a message of a variable length and turns it into a fixed-length output, known as a "hash value" or "hash." For a hash function to be secure, it should be computationally infeasible to invert the function for a given hash (preimage resistance) or to find two distinct messages that result in the same hash (collision resistance). These properties allow the use of the hash value in place of the message itself in a digital signature scheme, so that successful verification of the signature confirms that the message has not been altered.

In response to the cryptanalysis attack on the SHA-1 hash function presented at CRYPTO 2005 by Wang et al. [36], NIST decided to launch the SHA-3 competition for a new hash function standard [29]. The competition was announced in November 2007. By October 2008, 64 entries were received, and 51 were selected as first-round candidates in December 2008. Fourteen of these advanced as second-round candidates in July 2009, and the five finalists (BLAKE, Grøstl, JH, Keccak, and Skein) were announced in December 2010. The SHA-3 competition ended in October 2012, when Keccak was declared to be the winner.

Submission packages to the SHA-3 competition were required to include reference and optimized implementations in the C programming language [27]. NIST specified the Application Programming Interface (API) to be used (see Section 3) as well as the test vectors that were required in every submission package.

As the submission packages to the SHA-3 competition were subjected to public scrutiny, bugs were reported for several submissions in 2008 and 2009. A systematic analysis by Forsythe and Held of Fortify Software [18] found many bugs that commonly appear in C code, such as out-of-bounds reads, memory leaks, and null pointer dereferences. No bugs were reported during the remainder of the competition, and eventually, the resulting SHA-3 standard became widely implemented in many cryptographic libraries.

However, in September 2015, the implementation on the BLAKE website was updated with the comment: “fixed a bug that gave incorrect hashes in specific use cases” [4]. In 2018, Mouha et al. [24] rediscovered the bug using a new testing methodology that was eventually integrated into Google’s Project Wycheproof [10] and showed that the bug allows the collision resistance of the hash function to be violated.

At CT-RSA 2020, Mouha and Celi [22] showed a vulnerability affecting 11 out of the 12 implemented hash functions in Apple’s CoreCrypto library. The vulnerability required invoking the implementation on inputs of at least 4 GiB, which led to an infinite loop. This vulnerability showed a limitation in NIST’s Cryptographic Algorithm Validation Program (CAVP), which did not perform tests on hash functions for inputs larger than 65 535 bits. To overcome this limitation, NIST introduced the Large Data Test (LDT).

In October 2022, a vulnerability was disclosed by Mouha [31] that impacted both the final-round Keccak submission package and the resulting SHA-3 implementation by its designers. Depending on the specific inputs that were provided, the vulnerability resulted in either an infinite loop or a buffer overflow where attacker-provided values are XORed into memory [23].

Our Contributions. A shortcoming of previous work on finding vulnerabilities in hash function implementations is that they lack generality and clearly fall short as new vulnerabilities keep being found that remained unnoticed for over a decade (in spite of extensive public scrutiny). The novel contribution of this paper is to try to overcome this problem by introducing a new approach that can be used to find vulnerabilities in the hash function implementations of Apple’s CoreCrypto library, as well as in three out of the five SHA-3 finalist submissions: BLAKE, Keccak, and Grøstl. This paper explains how the vulnerability in Keccak was found. The vulnerability in Grøstl is a new contribution in this paper that has not been reported before. Our approach involves symbolic execution to find bugs within seconds, whereas test vectors may take much longer to execute.

2 Background and Related Work

The NIST approach to testing cryptographic implementations dates back to 1977, with the introduction of two sets of test vectors for the Data Encryption Standard (DES) in SP 500-20 [25]. Now known as Known Answer Tests (KATs) or static Algorithmic Functional Tests (AFTs), the first set of test vectors were intended to "fully exercise the non-linear substitution tables" (S-boxes) of the DES. The second set of test vectors, called Monte Carlo Tests (MCTs), contained "pseudorandom data to verify that the device has not been designed just to pass the test set." Although originally intended to test hardware implementations, this approach can be applied to both hardware and software and forms the basis of NIST's Cryptographic Algorithm Validation Program (CAVP).

Submissions to the NIST SHA-3 competition were required to include implementations in the C programming language. NIST specified an API [27] and provided source code to generate KATs and MCTs [28]. These KATs and MCTs helped to ensure that various implementations of the same algorithm were consistent. Moreover, an interesting innovation was the inclusion of an Extremely Long Message KAT, which provided a 1 GiB message with the goal of ensuring that large inputs are processed correctly.

For (authenticated) encryption algorithms, the SUPERCOP [7] and BRUTUS [34] frameworks perform some additional tests, such as checking whether overlapping inputs are handled correctly or whether encryption followed by decryption returns the original plaintext.

Aumasson and Romailler introduced crypto differential fuzzing [3] which uses a fuzzer to compare the outputs of different cryptographic libraries and find discrepancies. This approach turned out to be very effective, as shown by the many bugs found by Vranken's Cryptofuzz project [35].

Formal methods and program verification can also be applied to hash function implementations. Chudnov et al. [15] demonstrated that the Keyed-Hash Message Authentication Code (HMAC) implementation (using the SHA-256 hash function) of Amazon's s2n library conforms to a formal specification by using Galois's Software Analysis Workbench (SAW). The HACL* cryptographic library [32, 38] is formally verified using the F* verification framework. We refer to Protzenko and Ho [33] for an explanation of how its hash function implementations have recently been completely overhauled. Lastly, we mention Chapman et al.'s SPARKSkein [13] as an implementation of the SHA-3 finalist Skein that was written and verified using the SPARK [1] language and toolset.

Chapman et al. [13] pointed out a bug in the Skein submission package to NIST. The bug involves messages of more than $2^{64}-8$ bits. Although impractical, this violates a requirement in the SHA-3 call for submissions that a candidate algorithm (and therefore logically also a correct implementation of the algorithm) "shall support a maximum message length of at least $2^{64}-1$ bits" [26].

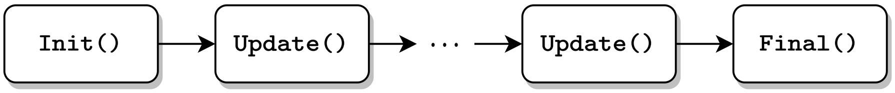

Fig. 1. An evaluation of a hash value on a message that is provided "on the fly" using any number of calls to `Update()` of arbitrary lengths.

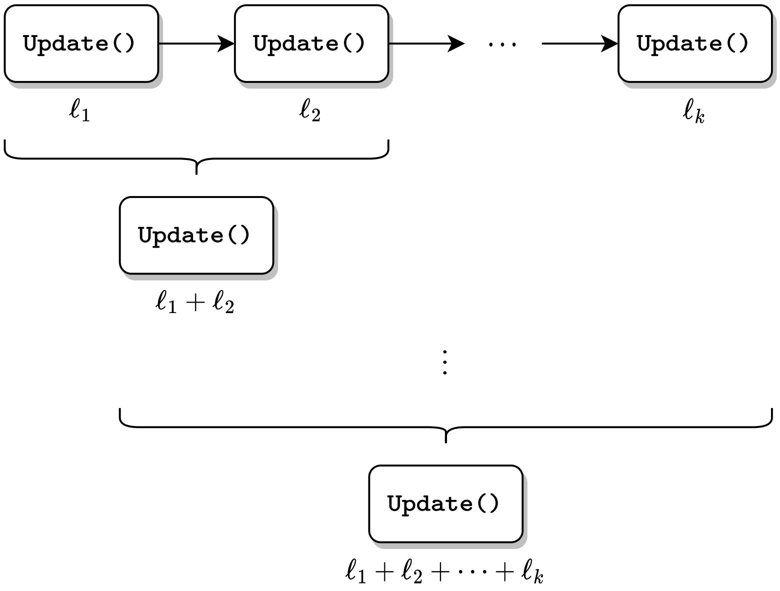

Fig. 2. To prove that any number of calls to `Update()` with arbitrary lengths result in a correct computation, it is sufficient to prove that two calls to `Update()` are equivalent to one larger call to `Update()` on the concatenation of both inputs.

3 Cryptographic Hash Function Interfaces

An API for the SHA-3 competition was specified by NIST [27], requiring the `hashState` data structure and four function calls: `Init()`, `Update()`, `Final()`, and `Hash()`. The API was designed for 64-bit operating systems, which were already common at the start of the SHA-3 competition.

The purpose of the `hashState` data structure is to contain "all information necessary to describe the current state of the SHA-3 candidate algorithm." It must contain the `hashbitlen` variable to indicate the output size of the particular instantiation of the hash function.

The four function calls show how `hashState` is intended to be used:

- `Init()` initializes the `hashState` data structure.
- Once initialized, any number of calls to `Update()` can be made to process parts of the message by updating `hashState` correspondingly. In practice,

this "incremental hashing" API offers a major efficiency improvement if the message is not available all at once or split over two or more non-contiguous arrays [33, p. 9].

- `Final()` performs any final processing needed on `hashState` to output the hash value.
- `Hash()` processes the message all-at-once by calling `Init()`, `Update()`, and `Final()`.

Let us assume that the all-at-once computation using `Init()`, `Update()`, and `Final()` is correct. Then, a sufficient (but not necessary) condition for the correctness of a computation using *any* number of calls to `Update()` (as shown in Figure 1) is:

Condition 1 *Two consecutive calls to `Update()` change `hashState` in the same way as one call to `Update()` on the concatenation of both inputs.*

It can be seen that Condition 1 is sufficient by means of a proof by contradiction where the condition is applied recursively as illustrated in Figure 2. However, there are several cases where Condition 1 is not necessary:

- Let us denote a `hashState` as *valid* if and only if is reachable from `Init()` followed by any number of calls to `Update()`. Then, it is not necessary that Condition 1 holds if `hashState` is invalid.
- If the `Final()` function can return the same hash value on two distinct but valid `hashState` data structures, Condition 1 is not necessary either. However, this does not seem to occur in practice, as we have only encountered implementations where `hashState` uniquely represents the message processed so far.
- In the NIST SHA-3 API, all lengths are provided in bits using a 64-bit unsigned integer [27], and a candidate algorithm may impose a maximum message length of $2^{64} - 1$ bits [26]. If this maximum message length is imposed, Condition 1 is not necessary for a sequence of two `Update()` calls that exceed the maximum message length (as the hash function may not be defined in this case).
- The NIST SHA-3 API document specifies that all calls to `Update()` contain data lengths (in bits) that are divisible by eight, except possibly the last call. Therefore, for two consecutive calls to `Update()`, we may restrict the first call to a data length that is a multiple of eight bits.

In the next section, our goal will be to verify Condition 1 for a given hash function implementation, possibly along with some preconditions to exclude the aforementioned cases where Condition 1 is not necessary. We will remove some lines of code: as in many previous works we are aiming for the "less ambitious but still important goal of stating partial specifications of program behavior and providing methodologies and tools to check their correctness" [5].

Before concluding this section, note that we will assume throughout this paper that `Init()`, `Update()`, and `Final()` are called in the "correct" order. In practice it can be desirable to call `Update()` after `Final()`. However, as shown by Benmocha et al. [6], this can be highly insecure for cryptographic libraries that do not expect such usage.

4 Program Verification Using KLEE

In this paper, we will use KLEE [11], which is a symbolic execution tool built on top of the LLVM (Low-Level Virtual Machine) [21] compiler architecture.

Although a typical use of KLEE is to automatically generate test vectors that achieve high code coverage, it can also be used to prove the full functional equivalence of two implementations [11, § 5.5]. Whenever KLEE encounters an execution branch based on a symbolic value, it will (conceptually) follow both branches, maintaining a set of constraints called path conditions for each branch. A downside of this approach is that the number of paths can grow very quickly. To overcome this path explosion problem, KLEE employs a variety of strategies to reduce the number of queries that are sent to the underlying constraint solver.

Unlike CBMC [16], which is a Bounded Model Checker for C and C++ programs, KLEE does not require a bound on the number of iterations for every loop. The NIST SHA-3 competition required a maximum message length of at least $2^{64} - 1$ bits, and it is common to see hash functions that process the message iteratively using a compression function that takes 512 or 1024 bits of input. Computing the hash for such large messages is not possible in practice, however, we will see that our approach using KLEE handles such inputs quite quickly (and without the need for loop unwinding nor manual efforts to rewrite loops).

In the following sections, we will show how to apply KLEE to the reference implementations of the five SHA-3 finalists, as well as to the SHA-3 implementation of XKCP and the hash functions implemented in Apple's CoreCrypto library. In this paper, we only provide the full source code for our KLEE experiments on the JH algorithm. However. the code for all our experiments will be made available as a software artifact.

Table 1 summarizes the runtimes for a 256-bit hash output value, except for Keccak and SHA-3 where we will choose a rate of 1024 bits. We found that the execution time typically does not depend on the length of the hash value. Our experiments were performed on an Intel Core i7-1165G7 processor using KLEE 2.3 with the default STP (Simple Theorem Prover) solver [12, 19].

4.1 JH

JH [37] is a hash function designed by Hongjun Wu that advanced to the final round of the NIST SHA-3 competition. As required for all submissions [26], JH supports hash lengths of 224, 256, 384, and 512 bits. The message is padded to a multiple of 512 bits and then split into 512-bit blocks which are processed by the same compression function `F8()`.

Table 1. KLEE runtimes (in minutes and seconds) for a rate of 1024 bits (for Keccak and SHA-3) or a 256-bit hash output length (for all other hash functions). The second column is the runtime to find a test vector that reveals a bug (if the code is incorrect), and the third column is the runtime to prove correctness (after patching the implementation if there is a bug).

Implementation	Buggy	Correct
BLAKE	5 s	11 m 59 s
Grøstl	6 s	10 s
JH	—	50 s
Keccak	1 s	39 s
Skein	—	34 s
XKCP (SHA-3)	1 s	20 s
CoreCrypto	1 s	2 m 41 s

We provide the entire `jh_klee.c` that we used in our experiment in Listing 1. It contains the `Update()` function that is specified in `jh_ref.h` of the JH submission package. For readability, we adjusted the indentation, and for compactness, we removed the source code comments. The only other change that we made to `Update()` is to comment out the lines involving `memcpy()` and `F8()`, as our (partial) equivalence checking does not involve the contents of the message (only its length), nor does it involve the implementation of the compression function `F8()`. Moreover, throughout this paper we do not make any statements about the correctness of `Init()`, `Final()`, nor `Hash()`.

In Listing 2, we provide the Makefile that uses Docker as an easy and portable way to run KLEE on `jh_klee.c`. It will either return a test vector that violates Condition 1, or prove that no such test vector exists. We find that after 50 seconds, KLEE proves the (partial) consistency of the `Update()` function. An overview of the execution times for all our experiments is given in Table 1.

Five lines in Listing 1 are marked with the comment `// optional` following the reasoning in Section 3. They can be safely omitted with the only downside that they roughly double the execution time of KLEE. However, these five lines can be helpful to adapt the approach to other SHA-3 candidate implementations where they may be needed.

Listing 1. Application to JH (`jh_klee.c`).

```
#include <assert.h>
#include <klee/klee.h>

typedef unsigned char BitSequence;
typedef unsigned long long DataLength;
typedef enum { SUCCESS = 0, FAIL = 1,
               BAD_HASHLEN = 2 } HashReturn;
typedef struct {
  int hashbitlen;
  unsigned long long databitlen;
  unsigned long long datasize_in_buffer;
} hashState;

HashReturn Update(hashState *state, const BitSequence
                          *data, DataLength databitlen)
{
  DataLength index;
  state->databitlen += databitlen;
  index = 0;

  if ( (state->datasize_in_buffer > 0 ) &&
       ((state->datasize_in_buffer+databitlen)<512) ) {
    if ( (databitlen & 7) == 0 ) {
      //memcpy(state->buffer +
      //(state->datasize_in_buffer >> 3), data,
      //64-(state->datasize_in_buffer >> 3));
    }
    //else memcpy(state->buffer +
    //          (state->datasize_in_buffer >> 3), data,
    //          64-(state->datasize_in_buffer >> 3)+1);
    state->datasize_in_buffer += databitlen;
    databitlen = 0;
  }

  if ( (state->datasize_in_buffer > 0 ) &&
       ((state->datasize_in_buffer+databitlen)>=512)) {
    //memcpy(state->buffer +
    //        (state->datasize_in_buffer >> 3), data,
    //       64-(state->datasize_in_buffer >> 3) );
    index = 64-(state->datasize_in_buffer >> 3);
    databitlen = databitlen -
                     (512 - state->datasize_in_buffer);
    //F8(state);
    state->datasize_in_buffer = 0;
  }

  for ( ; databitlen >= 512; index = index+64,
                       databitlen = databitlen - 512) {
    //memcpy(state->buffer, data+index, 64);
    //F8(state);
  }

  if (databitlen > 0) {
    //if ((databitlen & 7) == 0)
```

```
      //memcpy(state->buffer, data+index,
      //       (databitlen & 0x1ff) >> 3);
    //else
      //memcpy(state->buffer, data+index,
      //       ((databitlen & 0x1ff) >> 3)+1);
    state->datasize_in_buffer = databitlen;
  }

  return(SUCCESS);
}

void test(int hashbitlen) {
  hashState s, s2;
  DataLength databitlen, databitlen1, databitlen2;

  klee_make_symbolic(&s, sizeof(s), "s");
  klee_make_symbolic(&s2, sizeof(s2), "s2");
  klee_make_symbolic(&databitlen, sizeof(databitlen),
                     "databitlen");
  klee_make_symbolic(&databitlen1, sizeof(databitlen1),
                     "databitlen1");
  klee_make_symbolic(&databitlen2, sizeof(databitlen2),
                     "databitlen2");

  s.hashbitlen = hashbitlen;
  s2.hashbitlen = hashbitlen;

  klee_assume(s.databitlen == s2.databitlen);
  klee_assume(s.datasize_in_buffer ==
              s2.datasize_in_buffer);
  klee_assume(s.datasize_in_buffer < 512); // optional
  klee_assume(s2.datasize_in_buffer < 512); // optional

  klee_assume(databitlen == databitlen1 + databitlen2);
  klee_assume(databitlen >= databitlen1); // optional
  klee_assume(databitlen >= databitlen2); // optional
  klee_assume(databitlen1 % 8 == 0); // optional

  Update(&s, NULL, databitlen);

  Update(&s2, NULL, databitlen1);
  Update(&s2, NULL, databitlen2);

  if (s.databitlen != s2.databitlen)
    klee_assert(0);
  if (s.datasize_in_buffer != s2.datasize_in_buffer)
    klee_assert(0);
}

int main() {
  //test(224);
  test(256);
  //test(384);
  //test(512);

  return 0;
}
```

Listing 2. Application to JH (`Makefile` with visible tabs for readability).

```
TARGET = jh_klee

all: $(TARGET)

$(TARGET): $(TARGET).c
␣docker run --rm -v $(CURDIR):/home/klee/host \
␣--ulimit='stack=-1:-1' klee/klee:2.3 \
␣/tmp/llvm-110-install_O_D_A/bin/clang -I \
␣klee_src/include -emit-llvm -c -g3 -O3 \
␣host/$(TARGET).c -o host/$(TARGET).bc
␣time docker run --rm -v $(CURDIR):/home/klee/host \
␣--ulimit='stack=-1:-1' klee/klee:2.3 \
␣klee_build/bin/klee -exit-on-error-type=Assert \
␣host/${TARGET}.bc
␣docker run --rm -v $(CURDIR):/home/klee/host \
␣--ulimit='stack=-1:-1' klee/klee:2.3 \
␣klee_build/bin/ktest-tool $$(docker run --rm -v \
␣$(CURDIR):/home/klee/host --ulimit='stack=-1:-1' \
␣klee/klee:2.3 sh -c "ls \
␣host/klee-last/*.assert.err" | head -n 1 | sed \
␣'s/.assert.err/.ktest/')

clean:
␣\rm -rf *.bc klee-last klee-out-*
```

4.2 Skein

Skein is a final-round SHA-3 submission designed by Ferguson et al. [17]. Like JH, its primary proposal processes the message in 512-bit blocks regardless of the hash value length.

Although the algorithm used by Skein's implementation to process the message in blocks follows a completely different approach compared to JH, the KLEE proving harness looks quite similar with the main difference that the assertion on `datasize_in_buffer` is replaced by an assertion on `u.h.bCnt`.

The execution time is even less than for JH, as KLEE only needs 34 seconds to prove that there are no test vectors that violate the assertions.

4.3 BLAKE

Another final-round SHA-3 submission is the hash function BLAKE by Aumasson et al. [2]. Depending on the hash length, BLAKE uses either a 512-bit or a 1024-bit compression function. It has a `datalen` variable to keep track of the number of bits in the buffer, similar to `datasize_in_buffer` for JH.

However, it also keeps track of a counter for the total number of message bits processed so far. Depending on the hash length, this counter is stored either in an array with two 32-bit unsigned integers, or an array with two 64-bit unsigned integers.

In September 2015, the BLAKE website [4] was updated to correct a bug in all implementations submitted during the SHA-3 competition. Using our approach, KLEE easily rediscovers this bug in just five seconds.

More specifically, for the 256-bit hash value, it provides a test vector showing that `Update()` on 384 bits followed by 512 bits results in a different state than a single update of 384+512=896 bits.

This is consistent with the bug conditions described by Mouha et al. [24]: the bug occurs when an incomplete block (less than 512 bits) is followed by a complete block (512 bits).

Using the updated code on the BLAKE website [4], we run into an obstacle when running KLEE. It does not terminate within a reasonable amount of time, as it suffers from the path explosion problem mentioned in Section 2.

Further analysis shows that an if-branch inside the while-loop is the culprit of the path explosion. For the 512-bit block size, the BLAKE code is as follows:

```
while( databitlen >= 512 ) {
  state->t32[0] += 512;
  if (state->t32[0] == 0)
    state->t32[1]++;

  //compress32( state, data );
  data += 64;
  databitlen  -= 512;
}
```

We found that this path explosion can be avoided if the counter of the message bits hashed so far is not stored as an array of two unsigned 32-bit variables, but as one unsigned 64-bit variable. More specifically, we change the BLAKE code as follows:

```
while( databitlen >= 512 ) {
  state->t64 += 512;

  //compress32( state, data );
  data += 64;
  databitlen  -= 512;
}
```

With this replacement, KLEE proves that the assertions are unreachable in less than 12 minutes. Clearly, the execution time is an order of magnitude higher than in the previous examples. It appears that this is due to the additional counter variable used by the BLAKE hash function. If this counter variable is removed, the execution time of KLEE is reduced to only nine seconds.

4.4 Grøstl

We now move on to another SHA-3 finalist: Grøstl by Gauravaram et al. [20]. The message is split into either 512-bit or 1024-bit blocks, depending on the length of the hash value. To the best of our knowledge, no bugs have been reported for this implementation.

In the reference implementation of Grøstl, not all loops are inside `Update()` but also inside the function `Transform()` that does not just process one block, but any number of complete blocks. We already notice a first problem here: all variables representing the message length are 64-bit integers, but the function `Transform()` is declared with a parameter of the (user-defined) 32-bit type `u32`, resulting in an incorrect implicit cast.

As we apply our approach using KLEE, it takes six seconds to find a second bug. The bug is again due to the use of incorrect types: the variable `index` is declared as `int`, which is a 32-bit datatype on 64-bit processors. However, for sufficiently large message inputs, the value of `index` overflows, which results in undefined behavior in the C programming language.

In Listing 3, we provide a program to demonstrate the bug. When compiled using gcc, the program writes a large amount of data into memory, almost certainly resulting in a crash. It gets more interesting when we compile this program using clang. The undefined behavior causes clang to perform an optimization that avoids a buffer overflow but instead outputs the same hash for two messages of a different length. Thereby, the implementation violates the collision resistance property (see Section 1).

We searched for implementations that may be vulnerable due to this bug but did not identify any projects or products that might be impacted. For this reason, we did not submit a vulnerability report. The most notable use of Grøstl that we found was as a part of the proof-of-work algorithm of the initial version of the Monero cryptocurrency. However, the use of Grøstl there has long been discontinued.

With the two type errors fixed, proving the correctness using KLEE turned out to be much more difficult than expected. We again face a path explosion problem, which we addressed by hard-coding the block size and rewriting a loop that copied data byte-by-byte into a buffer. With these modifications, KLEE terminated in ten seconds with a proof that the assertions are unreachable.

Listing 3. Due to undefined behavior, the Grøstl bug results in a segmentation fault when compiled using gcc, or a collision when compiled using clang (`groestl_bug.c`).

```c
#include <stdio.h>
#include <sys/mman.h>
#include "Groestl-ref.h"

int main() {
  int hashbitlen = 256;
  DataLength len1 = (1uLL<<35) + 8;
  DataLength len2 = 8;

  BitSequence* Msg = (BitSequence*) mmap(NULL, len1/8,
        PROT_READ, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);

  BitSequence digest[64];

  printf("Hashing %llu-bit message... \nHash: ", len1);

  Hash(hashbitlen, Msg, len1, digest);

  for (int i = 0; i < hashbitlen/8; i++) {
    printf("%02x", digest[i]);
  }
  printf("\n");

  printf("Hashing %llu-bit message... \nHash: ", len2);

  Hash(hashbitlen, Msg, len2, digest);

  for (int i = 0; i < hashbitlen/8; i++) {
    printf("%02x", digest[i]);
  }
  printf("\n");

  return 0;
}
```

4.5 Keccak

The only SHA-3 finalist that we did not yet study in this paper, is the submission that won the competition: Keccak by Bertoni et al. [9]. Keccak pads the message and splits it into blocks, which are then processed by a cryptographic permutation. The block size, also known as the rate, has a default value of 1024 bits [9]. We will focus on this default value for now, and discuss the impact of the block size later.

A vulnerability was reported by Mouha and assigned CVE-2022-37454 [31]. As the winner of the SHA-3 competition, the Keccak reference code is quite widespread, and the vulnerability impacted various projects such as Python and PHP. For the details of the vulnerability, we refer to Mouha and Celi [23]. In this paper, we explain how the vulnerability was discovered using KLEE.

A straightforward approach using KLEE does not terminate in a reasonable amount of time. Therefore, it can be good to rule out an infinite loop in the Keccak implementation, as this would lead KLEE to enter into an infinite loop as well.

For `while(i < databitlen)` to terminate, a sufficient but not necessary condition is that `i` advances in every loop iteration. This is easy to check by introducing an `old_i` variable that is initialized to `i`. When the variable `i` is modified, we ensure that it is different from the previous loop iteration:

```
if (i == old_i) klee_assert(0);
```

At the end of the loop, we set `old_i = i`. With this additional code to detect infinite loops, it takes KLEE less than a second to output an assertion error. By analyzing the test vector provided by KLEE, we can confirm that we have indeed found an infinite loop. Note that this additional code is only used to allow us to easily detect an infinite loop in KLEE, the additional code is not necessary for a correct implementation (which does not contain an infinite loop).

It turns out that there is not only an input that leads to an infinite loop, but another input that causes a large amount of data to be written into memory, leading to a segmentation fault. The details of this buffer overflow are given by Mouha and Celi [23] and outside the scope of this paper.

The bug presents itself when there are already x bits of data in the buffer, and then an `Update()` of $2^{32} - x$ bits or more is made. Two problems can occur in Keccak's code below: the higher bits may be discarded due to an incorrect cast to a 32-bit integer, and the addition may overflow:

```
partialBlock = (unsigned int) (databitlen - i);
if (partialBlock + state->bitsInQueue > state->rate)
  partialBlock = state->rate - state->bitsInQueue;
```

We can correct this bug by rearranging the code a bit, so that `partialBlock` is at most equal to the block size. In that case, a 32-bit integer suffices for `partialBlock`:

```
if (databitlen - i > state->rate - state->bitsInQueue)
  partialBlock = state->rate - state->bitsInQueue;
else
  partialBlock = (unsigned int) (databitlen - i);
```

For the corrected code, KLEE requires only 39 seconds to prove that the assertions cannot be violated.

Finally, we note that KLEE did not seem to terminate within a reasonable amount of time for block sizes that are not a multiple of two. Such block sizes were proposed during the SHA-3 competition to handle different levels of security. Unfortunately, it appears to be a common issue that solvers have difficulties handling divisions by a constant that is not a power of two. KLEE has a `-solver-optimize-divides` flag that tries to optimize such divisions before passing them to the solver. However, even with this flag, we could not find a way to make KLEE terminate for block sizes that are not a multiple of two.

4.6 XKCP (SHA-3)

The eXtended Keccak Code Package (XKCP) [8] is maintained by the Keccak team. It contains the NIST-standardized variant of the Keccak hash function. Between the final-round Keccak submission and the SHA-3 standard, only the message padding is different.

The implementation of SHA-3 in XKCP is based on the implementation of the final-round Keccak submission. However, it has gone through quite a bit of refactoring. For this reason, we list XKCP's SHA-3 as a separate implementation in Table 1.

The same bug that impacts the final-round Keccak submission is present in XKCP as well, although two calls to `Update()` with a combined length of 2^{32} bytes (4 GiB) rather than 2^{32} bits (512 MiB) are required to trigger it. Moreover, it has another bug (not present in the Keccak submission) where messages slightly below 2^{64} bytes result in an infinite loop. This is due to an overflow in the comparison operation (`dataByteLen >= (i + rateInBytes)`), which has been replaced by `dataByteLen-i >= rateInBytes` in the corrected version of the code.

Using KLEE, it takes less than one second to provide a test vector that triggers the bug, assuming we again augment the code to detect infinite loops. For a 1024-bit block size, KLEE requires 20 seconds to prove that the assertions are unreachable.

4.7 CoreCrypto

Lastly, we would like to revisit the infinite loop in Apple's CoreCrypto library. The vulnerability impacted 11 out of the 12 implemented hash functions and was assigned CVE-2019-8741 [30]. For details of the bug, we refer to Mouha and Celi [22].

The approach using KLEE is quite straightforward to implement. KLEE finds a vulnerable test vector in less than a second for the original implementation and can prove that the assertions are unreachable in less than three minutes for the updated implementation.

We want to point out an interesting coincidence here. If we start from Apple CoreCrypto's corrected implementation of `Update()`, with just a little bit of refactoring (such as replacing `len` by `dataByteLen-i`, renaming variables and functions, and removing unneeded code), we end up with the corrected implementation of `Update()` that is used by XKCP. It seems that `Update()` is simple enough that two teams can independently arrive at the same implementation (up to simple refactoring), but complex enough to contain vulnerabilities that remained undiscovered for over a decade.

5 Limitations and Discussion

After the SHA-3 competition ended in 2012, vulnerabilities were found in the implementations of the SHA-3 finalist BLAKE in 2015 [4], in 11 out of the 12 implemented hash functions of Apple's CoreCrypto library in 2019 [30], and recently in the reference implementation of the SHA-3 winner Keccak [31].

These vulnerabilities were all related to the `Update()` function that is used to process the message in blocks. Nevertheless, the impact of the vulnerabilities is quite different. Whereas XKCP's SHA-3 implementation contained a buffer overflow vulnerability with the possibility of arbitrary code execution, the impact of the vulnerability in Apple's CoreCrypto library is limited to an infinite loop. The BLAKE vulnerability cannot be used to trigger any runtime error, however, it can be used to violate the collision resistance property of the hash function.

This allows us to make a first observation that approaches to avoid memory safety problems (such as enforcing coding standards, sandboxing, or moving away from C/C++ to safer programming languages) would be helpful, but not sufficient to avoid the vulnerabilities described in this paper. We are also reaching the limits of approaches using test vectors: the Large Data Test (LDT) can take quite a long time to execute on a 4 GiB message to detect bugs in Apple's CoreCrypto library, and for the XKCP bug, a single large call to `Update()` does not trigger the vulnerability (as it requires that some data is already present in the buffer).

Therefore, it can be interesting to consider approaches using symbolic execution. The approach we describe in this paper using KLEE turns out to be quite easy to deploy. We performed (partial) equivalence checking on the `Update()` function with lines involving the message contents and the compression function commented out. In a production environment, the proving harness would override these functions rather than commenting them out, so that the proofs can be part of a continuous integration process similar to how Amazon Web Services currently deploys CBMC [14].

Whereas approaches using large test vectors can take quite some time to execute (especially on slow hardware), symbolic execution can find bugs in a few seconds or prove correctness in less than 12 minutes, as shown by our timings in Table 1. This makes our approach a low-cost entry towards formal methods and program verification, and perhaps even a stepstone towards more rigorous approaches used in projects such as HACL* [32, 38] or SPARKSkein [13].

Indeed, the litmus test here would be to see how easily our approach extends to other submissions in the SHA-3 competition. We studied the implementations of all five finalists of the competition and found that we can either prove correctness or unearth new bugs (as in the case of Grøstl, where we show how undefined behavior can violate the collision resistance of the hash function implementation). And although our approach intends to check whether two calls to `Update()` are consistent with one call on the concatenation of both inputs, it also finds bugs that can be triggered by one call to `Update()`, as shown by the bugs in Grøstl and Apple's CoreCrypto library (as they cause KLEE to enter into an infinite loop).

Lastly, although our approach was helpful to discover bugs, we stress again that it is insufficient to claim that the implementations are correct. For example, we make no statements about potential bugs in `Init()`, `Final()`, or `Hash()`, nor potential bugs in the lines of `Update()` that were commented out. We also do not look into bugs related to the use of the API, such as those found by Benmocha et al. [6].

6 Conclusion and Future Work

We revisited the implementations of the five finalists of the NIST SHA-3 competition. These had been subject to a rigorous public review process from 2008 to 2012. However, it was not until 2015 that a vulnerability was discovered in the implementation of BLAKE, and very recently in the winning Kecak submission using the technique that is first described in this paper.

We showed how these bugs can be (re-)discovered in only a matter of seconds, requiring only minimal effort to construct a proving harness for the original code. Moreover, we also found a vulnerability in the Grøstl submission, allowing the construction of two messages that result in the same hash value when compiled using clang. Our approach would also have discovered a bug in Apple's CoreCrypto library that was reported in 2019.

Our approach requires symbolic execution to check whether two `Update()` calls (with some lines of code removed) are equivalent to one `Update()` call on the concatenation of both inputs. To check this property, we used the KLEE symbolic execution framework.

Unfortunately, our approach involves a bit of trial and error. In particular, to prove that none of the assertions fail, we sometimes needed to rewrite the code a bit to avoid that KLEE fails to terminate due to path explosion. This is a limitation as we would ideally like to make no changes at all to the source code, but perhaps this is an acceptable compromise to achieve the goal of (partial) program verification.

An interesting direction for future work is to find a way to extend our approach to Keccak and SHA-3 when the block size is not a power of two.

References

1. AdaCore, Thales: Implementation Guidance for the Adoption of SPARK (July 2020), https://www.adacore.com/uploads/books/pdf/Spark-Guidance-1.2-web.pdf

2. 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), https://www.aumasson.jp/blake/blake.pdf
3. Aumasson, J.P., Romailler, Y.: Automated Testing of Crypto Software Using Differential Fuzzing. Black Hat USA 2017 (2017), https://yolan.romailler.ch/ddl/talks/CDF-wp_BHUSA2017.pdf
4. Aumasson, J.P.: SHA-3 proposal BLAKE (September 2015), https://web.archive.org/web/20150921185010/https://131002.net/blake/
5. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In: Boiten, E.A., Derrick, J., Smith, G. (eds.) IFM 2004. LNCS, vol. 2999, pp. 1–20. Springer (2004). https://doi.org/10.1007/978-3-540-24756-2_1
6. Benmocha, G., Biham, E., Perle, S.: Unintended Features of APIs: Cryptanalysis of Incremental HMAC. In: Dunkelman, O., Jr., M.J.J., O'Flynn, C. (eds.) SAC 2020. LNCS, vol. 12804, pp. 301–325. Springer (2020). https://doi.org/10.1007/978-3-030-81652-0_12
7. Bernstein, D.J., Lange, T.: eBACS: ECRYPT Benchmarking of Cryptographic Systems (December 2022), https://bench.cr.yp.to
8. Bertoni, G., Daemen, J., Hoffert, S., Peeters, M., Assche, G.V., Keer, R.V.: eXtended Keccak Code Package (October 2022), https://github.com/XKCP/XKCP
9. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: The Keccak SHA-3 submission. Submission to the NIST SHA-3 Competition (Round 3) (2011), https://keccak.team/files/Keccak-submission-3.pdf
10. Bleichenbacher, D., Duong, T., Kasper, E., Nguyen, Q.: Project Wycheproof (December 2019), https://github.com/google/wycheproof
11. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In: Draves, R., van Renesse, R. (eds.) OSDI 2008. pp. 209–224. USENIX Association (2008)
12. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: automatically generating inputs of death. In: Juels, A., Wright, R.N., di Vimercati, S.D.C. (eds.) CCS 2006. pp. 322–335. ACM (2006). https://doi.org/10.1145/1180405.1180445
13. Chapman, R., Botcazou, E., Wallenburg, A.: SPARKSkein: A Formal and Fast Reference Implementation of Skein. In: da Silva Simão, A., Morgan, C. (eds.) SBMF 2011. LNCS, vol. 7021, pp. 16–27. Springer (2011). https://doi.org/10.1007/978-3-642-25032-3_2
14. Chong, N., Cook, B., Eidelman, J., Kallas, K., Khazem, K., Monteiro, F.R., Schwartz-Narbonne, D., Tasiran, S., Tautschnig, M., Tuttle, M.R.: Code-level model checking in the software development workflow at Amazon Web Services. Softw. Pract. Exp. **51**(4), 772–797 (2021). https://doi.org/10.1002/spe.2949
15. Chudnov, A., Collins, N., Cook, B., Dodds, J., Huffman, B., MacCárthaigh, C., Magill, S., Mertens, E., Mullen, E., Tasiran, S., Tomb, A., Westbrook, E.: Continuous Formal Verification of Amazon s2n. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10982, pp. 430–446. Springer (2018). https://doi.org/10.1007/978-3-319-96142-2_26
16. Clarke, E.M., Kroening, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer (2004). https://doi.org/10.1007/978-3-540-24730-2_15
17. Ferguson, N., Lucks, S., Schneier, B., Whiting, D., Bellare, M., Kohno, T., Callas, J., Walker, J.: The Skein Hash Function Family. Submission to the NIST

SHA-3 Competition (Round 3) (2010), https://www.schneier.com/wp-content/uploads/2015/01/skein.pdf

18. Forsythe, J., Held, D.: NIST SHA-3 Competition Security Audit Results (February 2009), https://web.archive.org/web/20120222155656if_/http://blog.fortify.com/repo/Fortify-SHA-3-Report.pdf
19. Ganesh, V., Dill, D.L.: A Decision Procedure for Bit-Vectors and Arrays. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 519–531. Springer (2007). https://doi.org/10.1007/978-3-540-73368-3_52
20. Gauravaram, P., Knudsen, L.R., Matusiewicz, K., Mendel, F., Rechberger, C., Schläffer, M., Thomsen, S.S.: Grøstl – a SHA-3 candidate. Submission to the NIST SHA-3 Competition (Round 3) (2011), https://www.groestl.info/Groestl.pdf
21. Lattner, C., Adve, V.S.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: CGO 2004. pp. 75–88. IEEE Computer Society (2004). https://doi.org/10.1109/CGO.2004.1281665
22. Mouha, N., Celi, C.: Extending NIST's CAVP Testing of Cryptographic Hash Function Implementations. In: Jarecki, S. (ed.) CT-RSA 2020. LNCS, vol. 12006, pp. 129–145. Springer (2020). https://doi.org/10.1007/978-3-030-40186-3_7
23. Mouha, N., Celi, C.: A Vulnerability in Implementations of SHA-3, SHAKE, EdDSA, and Other NIST-Approved Algorithms. In: Rosulek, M. (ed.) CT-RSA 2023. LNCS, vol. 13871, pp. 3–28. Springer (2023). https://doi.org/10.1007/978-3-031-30872-7_1
24. Mouha, N., Raunak, M.S., Kuhn, D.R., Kacker, R.: Finding Bugs in Cryptographic Hash Function Implementations. IEEE Trans. Reliab. **67**(3), 870–884 (2018). https://doi.org/10.1109/TR.2018.2847247
25. National Bureau of Standards: Validating the Correctness of Hardware Implementations of the NBS Data Encryption Standard. NBS Special Publication 500-20 (November 1977), https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nbsspecialpublication500-20e1977.pdf
26. National Institute of Standards and Technology: Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA-3) Family. 72 Fed. Reg. (November 2007), https://www.federalregister.gov/d/E7-21581
27. National Institute of Standards and Technology: ANSI C Cryptographic API Profile for SHA-3 Candidate Algorithm Submissions (February 2008), https://csrc.nist.gov/CSRC/media/Projects/Hash-Functions/documents/SHA3-C-API.pdf
28. 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
29. National Institute of Standards and Technology: Hash Functions: SHA-3 Project (June 2020), https://csrc.nist.gov/projects/hash-functions/sha-3-project
30. National Vulnerability Database: CVE-2019-8741 (April 2020), https://nvd.nist.gov/vuln/detail/CVE-2019-8741
31. National Vulnerability Database: CVE-2022-37454 (November 2022), https://nvd.nist.gov/vuln/detail/CVE-2022-37454
32. Polubelova, M., Bhargavan, K., Protzenko, J., Beurdouche, B., Fromherz, A., Kulatova, N., Béguelin, S.Z.: HACLxN: Verified Generic SIMD Crypto (for all your favourite platforms). In: Ligatti, J., Ou, X., Katz, J., Vigna, G. (eds.) CCS 2020. pp. 899–918. ACM (2020). https://doi.org/10.1145/3372297.3423352

33. Protzenko, J., Ho, S.: Functional Pearl: Zero-Cost, Meta-Programmed, Dependently-Typed Stateful Functors in F*. CoRR **abs/2102.01644** (2021), https://arxiv.org/abs/2102.01644
34. Saarinen, M.J.O.: BRUTUS (August 2016), https://github.com/mjosaarinen/brutus
35. Vranken, G.: Cryptofuzz - Differential cryptography fuzzing (December 2022), https://github.com/guidovranken/cryptofuzz
36. Wang, X., Yin, Y.L., Yu, H.: Finding Collisions in the Full SHA-1. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 17–36. Springer (2005). https://doi.org/10.1007/11535218_2
37. Wu, H.: The Hash Function JH. Submission to the NIST SHA-3 Competition (Round 3) (2011), https://www3.ntu.edu.sg/home/wuhj/research/jh/jh_round3.pdf
38. Zinzindohoué, J.K., Bhargavan, K., Protzenko, J., Beurdouche, B.: HACL*: A Verified Modern Cryptographic Library. In: Thuraisingham, B., Evans, D., Malkin, T., Xu, D. (eds.) CCS 2017. pp. 1789–1806. ACM (2017). https://doi.org/10.1145/3133956.3134043