Finding Bugs in Cryptographic Hash Function Implementations

Nicky Mouha, Mohammad S Raunak, D. Richard Kuhn, and Raghu Kacker

***Abstract*—Cryptographic hash functions are security-critical algorithms with many practical applications, notably in digital signatures. Developing an approach to test them can be particularly difficult, and bugs can remain unnoticed for many years. We revisit the NIST hash function competition, which was used to develop the SHA-3 standard, and apply a new testing strategy to all available reference implementations. Motivated by the cryptographic properties that a hash function should satisfy, we develop four tests. The Bit-Contribution Test checks if changes in the message affect the hash value, and the Bit-Exclusion Test checks that changes beyond the last message bit leave the hash value unchanged. We develop the Update Test to verify that messages are processed correctly in chunks, and then use combinatorial testing methods to reduce the test set size by several orders of magnitude while retaining the same fault-detection capability. Our tests detect bugs in 41 of the 86 reference implementations submitted to the SHA-3 competition, including the rediscovery of a bug in all submitted implementations of the SHA-3 finalist BLAKE. This bug remained undiscovered for seven years, and is particularly serious because it provides a simple strategy to modify the message without changing the hash value returned by the implementation. We detect these bugs using a fully-automated testing approach.**

***Index Terms*—Cryptographic Algorithm, Cryptographic Hash Function, Combinatorial Testing, Metamorphic Testing, SHA-3 Competition.**

I. Introduction

A (cryptographic) hash function transforms a *message* of a variable length into a fixed-length *hash value*. Among other properties, it should be difficult to invert the function or to find two messages with the same hash value (a *collision*).

One of the earliest uses of hash functions was to create small, fixed-sized, collision-resistant message digests (MDs) that can stand in place of large variable-length messages in digital signature schemes. Later, these functions started to be routinely used in other applications including message authentication codes, pseudorandom bit generation, and cryptographic key derivation. Starting from the early 1990s, a series of related hash functions were developed including MD4, MD5, SHA-0, SHA-1, and the SHA-2 family. The SHA-2 family, defined in Federal Information Processing Standard (FIPS) publication 180-4 [1], consists of four algorithms that produce MDs of lengths 224, 256, 384, and 512 bits. Over the years, many weaknesses [2], [3] of these functions have been discovered and attacks have been found that violate expected hash function security properties. Although no attack is known on the SHA-2 family, these hash functions are in the same general family, and could potentially be attacked with similar techniques.

In 2007, the National Institute of Standards and Technology (NIST) released a Call for Submissions [4] to develop the new SHA-3 standard through a public competition. The intention was to specify an unclassified, publicly disclosed algorithm, to be available worldwide without royalties or other intellectual property restrictions. To allow the direct substitution of the SHA-2 family of algorithms, the SHA-3 submissions were required to provide the same four message digest lengths.

Chosen through a rigorous open process that spanned eight years, SHA-3 became the first hash function standard that resulted from a competition organized by NIST. Out of the 64 initial submissions to the competition, 51 were selected for the first round, and were made publicly available. All submissions had to provide hash values, also known as message digests, for a NIST-provided set of messages (known as 'test vectors'), which we will refer to as the "SHA-3 Competition Test Suite."

Over a period of five years, these submitted algorithms and implementations were analyzed in multiple rounds. Out of the 51 first-round candidates, 14 made it to the second round, and five were selected for the third and final round. At the end of the process, the algorithm Keccak was selected as the winner, and subsequently became the SHA-3 standard [5].

The implementations of hash functions are vulnerable to implementation faults, also known as bugs. Designing test cases with high fault-finding effectiveness for hash functions is particularly difficult, because these functions fall into the category commonly known as "non-testable" programs [6], i.e., those lacking a *test oracle* to check their correctness.

To alleviate the test oracle problem [7], we will develop a systematic testing methodology, which can be considered within the framework of metamorphic testing [8]–[11]. Metamorphic testing relies on identifying some relations in the underlying algorithm that allow two test cases to work as pseudo-oracles of each other. Given a particular test case t_1 for a program p, a metamorphic relation would allow selection of another test case t_2 such that the outputs produced by the two tests: $p(t_1)$ and $p(t_2)$ can validate each other. All the tests that we propose in this paper will make use of such a metamorphic relation: either equality (the hash values must be the same) or inequality (the hash values must be different).

Traditional test selection strategies may use source code coverage information such as branch coverage or condition coverage. These strategies may not be effective in discovering certain types of bugs related to bit manipulations, which are very common in hash functions. Randomly selected test

N. Mouha, R. Kuhn and R. Kacker are with the National Institute of Standards and Technology. E-mail: nicky.mouha@nist.gov

M. Raunak is with the Department of Computer Science, Loyola University Maryland, Baltimore, MD 21210. E-mail: raunak@loyola.edu

cases, on the other hand, are likely to be weak in their fault discovering ability, because random generation of test values makes it highly likely that some special combinations of inputs will be missed.

In this paper, we will only focus on hash function implementations that follow the SHA-3 Competition API. Other algorithms and APIs are outside the scope of the research presented here. Furthermore, this work focuses only on the reference implementations of the SHA-3 candidates. Reference implementations are often the basis of other implementations, and it is reasonable to expect that the reference implementation can be used without modifications. Bugs in reference implementations can have a serious impact for practical applications: for example, a bug in the RSA reference implementation [12] affected several commercial and non-commercial SSH implementations in 1999. Optimized implementations often contain sections that are specific to a particular compiler, platform, or processor. As we restrict ourselves to a single build environment in this paper, the systematic testing of optimized implementations is out of scope, but nevertheless a very interesting topic for future work.

Between the rounds of the SHA-3 competition, the submitters were able to update their submissions. We took all submissions and their updates from the NIST website [13], a total of 86 implementations, and ran our tests on them. Through the selection of test cases that specifically target potential vulnerabilities in the implementations, we discovered a large number of bugs, many of which have never been discovered before, or remained hidden for years.

We designed four sets of test cases, two of which are simple variations of each other, to investigate the effectiveness of better partitioning of the message space combined with a combinatorial approach in discovering bugs [14]. Our test that checks whether input differences affect the hash value, found errors in 19 out of 86 implementations. We also found 32 implementations that incorrectly handle the processing of a message in chunks. Lastly, 17 implementations were found to have bugs where the hash value is not uniquely determined by the message. We should note that none of these 68 bugs were discovered by the SHA-3 Competition Test Suite.

A. Outline

The rest of the paper is organized as follows. After discussing related work and the limitations of existing approaches in Section II, we explain the background of hash function standardization and implementation validation in Section III. We introduce our new testing methods in Section IV, and apply them to all reference implementations submitted to the SHA-3 competition in Section V. We explore some bugs in detail in Section VI, in order to give more intuition about the bugs in the implementations, and the difficulty in detecting them at an early stage using traditional methods. In Section VII, we perform a code coverage analysis of various test suites. We discuss our results in Section VIII, and conclude the paper in Section IX.

II. Related Work and Traditional Approaches

A. Related Work

Implementations of cryptographic algorithms have been known to be a significant source of vulnerabilities for some time. Anderson [15], writing on why cryptosystems fail, cites a "senior NSA scientist" in 1993 as saying that "the vast majority of security failures occur at the level of implementation detail." Despite recognition of problems with many implementations, relatively little research has been done on systematic test methods tailored to cryptographic implementations. Most testing in the field follows established practices for general software testing, but cryptographic software may have several bit operations expressed in a few lines of code, with relatively few conditionals. As we discuss in this paper, cryptographic algorithms typically have characteristics quite different from other types of software, such that traditional test methods do not apply well.

Braga and Dahab [16] note the prevalence of failures in cryptographic software, and the lack of widely accepted methods and tools to prevent such bugs. They also note that unique aspects of cryptographic code make ordinary static analysis tools less effective, and mention the need for functional testing. Nevertheless, their survey did not find systematic methods for deriving such tests.

A 2014 study [17] investigated the distribution of flaws in cryptographic software implementations, finding 269 reported vulnerabilities, of which 223 were found in applications, 39 in protocols, and 7 in cryptographic primitives. The same study reviewed open problems in testing cryptographic primitives and included an example found in a popular algorithm implementation, noting that conventional verification and testing methods do not address many low level operations used in cryptographic software and do not scale adequately.

Among the approaches described in this paper, combinatorial testing has been applied to cryptographic functions for detection of Trojan Horse malware [18] and for applications using cryptographic library functions [19]. Model-based testing (MBT) has also been shown to be effective for cryptographic components [20], including hash function usage. However, in [20] MBT is applied at the application level (e.g., sequence of hash function calls), rather than for cryptographic primitives. Metamorphic testing has been applied to various security-related applications [21], and some work has been done towards developing metamorphic relations for cryptographic modules [22].

Very recently, Aumasson and Romailler [23] applied fuzz testing to compare two different implementations of the same cryptographic algorithm, and find several bugs in widely-used cryptographic libraries such as OpenSSL. Their technique is only effective when several implementations of an algorithm are available that do not all contain the same bug. In this paper, we look into techniques that overcome this limitation.

B. Limitations of Traditional Approaches

A large part of software testing is inherently a selection process, where a very small subset from an often infinite set of test cases is selected to execute the Implementation

Under Test (IUT). Due to our inability to perform 'exhaustive testing' in most cases and also due to the fact that software systems usually lack continuity, one can never use testing to prove the correctness of a system. As pointed out by Dijkstra [24], testing thus is primarily an approach to discover bugs. Consequently, the most effective test sets are those that are likely to discover the largest number of bugs or faults in the implementation.

The test suite design process is driven by two overarching questions:

1) How do we go about selecting the test cases from a practically infinite set of possible tests?
2) How do we know when we have selected enough test cases to feel confident about the discovery of all, or nearly all, bugs in the system?

Over the last five decades, testing researchers have tried to answer these questions by taking various approaches. Testing approaches can generally be characterized as either 1) white box or implementation-based testing; and 2) black box or specification-based testing.

White box testing assumes access to the source code and selects test cases based on how well they exercise different structural elements such as statements, branches, and paths of the IUT [25]. Selecting test cases based on structural coverage of the source code is known as meeting a coverage criterion. Statement coverage, branch coverage, and covering some subsets of paths (e.g. all-def or all-use) are common coverage criteria used for selecting test cases.

Black box testing, on the other hand, relies on the specification of the IUT as opposed to its implementation. Test cases, in the case of black box testing, are selected based on the expected behavior as defined in the requirements specification of the system. A stopping-rule for selecting test cases here is guided by other criteria such as partitioning the input space and ensuring some type of coverage of those partitions.

Test case selection for both white-box-based and black-box-based coverage criteria often assume the presence of a *test oracle* – the mechanism through which one can definitively conclude if an execution of the IUT has resulted in the expected behavior or not. In other words, the test oracle specifies when a test case has revealed a failure scenario.

Unfortunately, not all systems have well-defined or easily implementable test oracles. Computational learning as well as many other artificial intelligence related algorithms, simulation models, and cryptographic algorithms are some examples of programs where it is difficult to develop test oracles, because the test oracle may need to be as complex as the IUT. This makes it harder for designing and improving test cases for these types of programs.

The criteria based on covering structural elements of a program such as statements or branches are based on the observation that test cases that do not exercise a statement or a branch are unlikely to discover a bug in that structural element. Hence a simple objective is to select test cases that exercise all statements or branches. For complex programs, interactions of branches (specific paths) that a specific program execution traverses is also a good place to look for potential anomalies or bugs. Traditional approaches thus focus on test case selection that based on covering these structural elements.

Special scenarios arise with cryptographic algorithms. First, these algorithm implementations are classic examples of programs without test oracles. It is usually not possible to ascertain if the output generated by a cryptographic function is correct or not. Second, cryptographic algorithms often include many bit manipulations as opposed to long, winding, or complex paths through the program, which is not necessarily a common scenario for the vast majority of software that are developed and tested by software engineers. Consequently, ensuring the exercise of every statement or every branch may result in a set of trivial test cases that are not very effective in discovering bugs, especially the ones hiding deep inside in the bit manipulations.

These two characteristics make it particularly difficult to design test cases for cryptographic algorithms. Consequently, it also becomes difficult to know when to stop adding test cases to a test suite and how to measure the effectiveness of a test suite.

III. Background of Hash Function Standardization and Validation

Before we elaborate our specific approach and corresponding results from testing cryptographic hash function algorithms, a brief background discussion is necessary about how test cases were determined at NIST for testing cryptographic algorithms and their implementations.

US federal agencies that use cryptography to protect sensitive information must follow the standards and guidelines of the National Institute of Standards and Technology (NIST), and must validate their implementations, referred to as cryptographic modules.

To validate cryptographic modules according to FIPS 140-1 [26], *Security Requirements for Cryptographic Modules*, the Cryptographic Module Validation Program (CMVP) was established in 1995. Since the Federal Information Security Management Act (FISMA) of 2002, US federal agencies no longer have a statutory provision to waive FIPS. The Cryptographic Algorithm Validation Program (CAVP) [27] involves the testing of the implementations of FIPS-approved and NIST-recommended cryptographic algorithms, and is a prerequisite for the CMVP. The CAVP sends the inputs and possibly some intermediate computational state from a NIST reference implementation to the Implementation Under Test (IUT) and then compares the obtained and precomputed results.

In 1997, NIST launched an open competition for an Advanced Encryption Standard (AES) block cipher, and made a formal call for algorithms [28]. By 1998, fifteen AES candidate algorithms were received. In 2000, NIST announced that it had selected Rijndael for the AES [29].

As clarified in the Call for Submissions of the AES competition [28], submitters were required to provide a single floppy disk with two types of test vectors: Known Answer Tests (KATs) and Monte Carlo Tests (MCTs). Testing using KATs and MCTs is the basis of the CAVP, and the underlying ideas date back at least to 1977, when the National Bureau of Standards (NBS), the former name of NIST, published SP 500-20 [30] to test the encryption and decryption operations of the Data Encryption Standard (DES). Originally, KATs were intended to "fully exercise the non-linear substitution tables" (S-boxes), whereas the MCTs contain "pseudorandom data to verify that the device has not been designed just to pass the test set."

In the context of the AES competition, a set of KAT inputs were provided by NIST, for which the submitters needed to generate the corresponding outputs. Clearly, NIST could not know at the time of submission whether these KAT inputs would fully excercise any tables used in an implementation of the submitted algorithm. Therefore, for algorithms that used tables, the submitters were additionally required to provide their own set of KATs that exercise every table entry. For the MCTs, the choice of the pseudorandom inputs was left to the submitters.

Inspired by SP 500-20 [30], the NIST-provided KATs for the AES competition consist of all vectors where one input (the plaintext or the secret key) is kept constant, and the other input is the set of all bit strings with exactly one bit set. The MCTs are identical to those of SP 500-20: the submitters needed to iterate the algorithm four million times on a given input, while providing the outputs after every multiple of ten thousand iterations.

During the NIST SHA-3 competition, the testing methodology was again borrowed from the CAVP, as the KATs and MCTs of the SHA-3 Competition Test Suite were based on the CAVP tests for SHA-2. In addition to this, the "Extremely Long Message Test," not present in the CAVP for SHA-2, required the submitters to generate the hash value corresponding to a message with a length of 1 GiB.

Of particular interest here is that the KATs provided by NIST to test hash functions never contain more than one test vector of any particular length. This is true for the SHA-2 standard in the CAVP, as well as for the SHA-3 Competition Test Suite. As we will show later, this property will limit the ability to discover implementation bugs, as some bugs can only be revealed by testing two messages of the same length.

The iteration used in the MCTs means that a large number of messages of the same length will be processed. But the MCTs required for the SHA-3 competition only process inputs of exactly 1024 bits, which will not reveal bugs that may be present for other message lengths.

Besides this testing effort by NIST, the most notable external testing effort of the SHA-3 candidate submissions came from Forsythe and Held of Fortify [31]. They downloaded all 43 submission packages of the SHA-3 candidates that were not withdrawn at the time of their analysis, and tested all reference implementations. Their testing effort detected bugs in only five submissions: Blender, CRUNCH, FSB, MD6 and Vortex. Four types of bugs were found: buffer overflows, out-of-bound reads, memory leaks and null dereferences.

IV. Experimentation on Testing SHA-3 Candidates

To experiment with our test suite design and test case selection approaches, we have considered the SHA-3 competition for cryptographic hash functions and all the candidates submitted to it. A cryptographic hash function is designed to take an input string m of any size, and return a fixed-sized bitstring as output, $H(m)$. The input string is commonly known as the 'message' and the output is called the 'message digest' or 'MD'. The message digest is also called the 'digest', 'hash value', or 'hash output'.

A. Properties of cryptographic hash functions

To design our test cases for this experiment, we need to understand and possibly utilize the required characteristics of a (cryptographic) hash function. Informally, a hash function is expected to have the following main properties [32]:

1) *Preimage resistance.* Hash functions should be one-way functions. That is, given a message digest y, it should be infeasible to find a preimage, i.e., a message m for which $H(m) = y$.
2) *Second-preimage resistance.* Given a message m and a message digest $H(m)$, it should be infeasible to find a second message m' ($m \neq m'$) with the same message digest, i.e., $H(m) = H(m')$. A one-way function is both preimage and second preimage resistant. However, it is possible to construct functions that are second-preimage resistant, but not preimage resistant [32, Note 9.20]. Due to this subtle difference, preimage resistance and second-preimage resistance are considered to be separate notions.
3) *Collision resistance.* It should be infeasible to find two different messages that produce the same message digest, i.e., finding $m \neq m'$ for which $H(m) = H(m')$. Note that collisions necessarily exist due to the pigeonhole principle, however for a hash function it should be computationally infeasible to find them. Collision resistance implies second-preimage resistance, but it is a strictly stronger notion: the attacker can now choose both m and m', instead of choosing only m' (for a fixed m).
4) *Function behavior.* There is always an implicit property that comes with any mathematical function. This function property would require that for any message m, the computed digest $H(m)$ is always deterministic. If m is not altered, it should always produce the same $H(m)$.

We have designed three different sets of test cases for our experiments, to detect violations of some of these properties in a particular *implementation* of a hash function. Our test suites may seem nominally large, i.e., a large number of hash values computed while testing each IUT. However, we should note three important factors:

1) Our test suites are comparable to the sizes of the SHA-3 Competition Test Suite that each IUT needed to provide hash values for.
2) The test suites could finish executing in reasonable time, in the order of minutes or hours at most.
3) The number of test cases in our test suite is still much smaller than any exhaustive testing even in a reasonably restricted domain of different sized messages.

TABLE I
SHA-3 COMPETITION TEST SUITE SIZE (FOR EACH MESSAGE DIGEST SIZE)[†]

Name of the Test	# Message Digests
Short Message KAT	2048
Long Message KAT	513
Extremely Long Message KAT	1
Monte Carlo Test	100 000
Total	102 562

[†]Not a good indicator for runtimes as messages vary in size.

TABLE II
OUR TEST SUITE SIZES (FOR EACH MESSAGE DIGEST SIZE)[†]

Name of the Test	# Message Digests
Bit-Contribution Test	2 100 225
Bit-Exclusion Test	131 072
Update Test	1 048 576
Combinatorial Update Test	1510
Total	3 281 383

[†]Not a good indicator for runtimes as messages vary in size.

In Tables I and II, we show the number of message digests that are computed for the SHA-3 Competition Test Suite, compared to our test suite. The numbers are given for one message digest size, and therefore need to be multiplied by four to find the total number of message digests per hash function implementation.

Our test suite computes about 32 times more digests. However, our tests are roughly four times faster than the SHA-3 Competition Test Suite. The SHA-3 Competition Test Suite is dominated by the Extremely Long Message KAT. On the other hand our test suite's most time-consuming part is the Bit-Contribution Test. It computes about 2 million digests with an average size of 1024 bits, whereas the Extremely Long Message KAT processes one message of 1 GiB, which corresponds to about 8 million blocks of 1024 bits.

Furthermore, our test vectors are relatively short and independent of each other and can thus be executed in parallel, which is not the case for the Extremely Long Message KAT and the Monte Carlo Test of the SHA-3 Competition Test Suite. The speed advantage in favor of our test suite compared to the SHA-3 Competition Test Suite therefore increases when more cores are available or when the testing is done simultaneously on multiple machines. In the following discussion, we explain our test suite for one digest size. These tests are performed for each of the four different digest sizes that are required for a valid SHA-3 submission: 224, 256, 384 and 512 bits.

B. Bit-Contribution Test

The first set of test cases is designed with respect to the second-preimage resistance mentioned in Section IV-A. We call this set of test cases Bit-Contribution Test. In this scenario, we take a fixed message m of size n and compute the message digest, $H(m)$. We then systematically alter a single bit of m and compute $H(m)$ again. We repeat the process for every single bit of m. As part of the test suite, we vary the size n from 0 to 2048. No bits are set in the degenerate case of $n = 0$, as this corresponds to the empty string. All these message digests are stored in a large table to then search for possible collisions. If no collisions are found, the Bit-Contribution Test is passed successfully.

The idea for this test suite is based on the fact that any good hash function should have some contribution from each of the bits of a message m. Thus, any single bit altered in m should be reflected in the digest $H(m)$. We have combined this insight for test case designs with the additional expectation that the digests of two distinct messages should not collide, even if one message is only a single bit longer (or shorter) than the other message. This reasoning explains why all digests are inserted into the same large table (regardless of the length of the message), and why the zero-length string is included as well.

A typical hash function will split a message into blocks of a particular size, after which processing is done on each block. Well-known hash functions such as MD5, SHA-1 and SHA-2 use block sizes of 512 or 1024 bits, and such block sizes are typical for many SHA-3 candidates as well. Our test suite thus stops at messages of 2048 bits, which is twice the largest commonly-used block size of 1024 bits.

For messages of size 1, 2, . . ., 2048 bits, there are $2048 \times (1 + 2048)/2$ messages with exactly one bit set (one possible message of length one bit with one bit set, two possible messages of length two bits with one bit set, and so on). For the same range of messages, there are 2048 messages with all bits set to zero. We also add the empty string. This leads to $2048 \times (1 + 2048)/2 + 2048 + 1 \approx 2^{21}$ different hash values in our Bit-Contribution Test for each of the digests sizes of the IUT.

C. Bit-Exclusion Test

One of the requirements for the SHA-3 competition was that the candidate functions should properly handle messages whose sizes are not multiples of a byte. In other words, not only should the functions handle messages of size $0, 8, 16, \ldots 128, \ldots, 512$ bits etc.; they should also be able to handle message lengths of fractional byte sizes such as $5, 43$ or 509 bits.

Even though a message size is specified to be a fixed number of bits, i.e., not multiple of a byte, the SHA-3 Competition API [33] requires the message to be provided in an array of bytes. Furthermore, the C programming language does not prevent the programmer from going past the end of the array, which could result in an implementation that incorrectly reads past the last bit of the message. As the API clarifies that the data to be processed should be in the first n bits, it is implied that the bit outside of the specified message size n should be ignored. This is consistent with the function behavior property of Section IV-A.

We therefore designed a set of test cases for various message lengths n, and then started altering bits outside of the specified size; i.e., n^{th} bit, $(n+1)^{th}$ bit, $(n+2)^{th}$ bit etc. (assuming bit indices start at 0), and computed the hash value for each of these messages. We then look for a case where the message digest changes due to a bit alteration beyond n, while the message m within the specified size of n bits remains the same.

The idea behind the above scenario is that any bit outside the specified size parameter should be excluded and should not contribute towards computing the digest. Consequently, perturbations beyond position n must not create a different digest for a particular message. As a concrete example, let us suppose that we provide a message of size 509. As the message is provided as a byte array, it is necessary for the calling application to store at least 512 bits of data. But the parameter to the `Hash()` function specifies that the digest $H(m)$ is to be created based on the first 509 bits. If we provide the same message with a bit flipped in the 509^{th} position (assuming again that bit indices start at 0), this should not alter the message digest $H(m)$.

Our Bit-Exclusion Test suite computes 2^{17} different message digests for each digest size of the IUT. This number is derived from multiplying the range of message sizes (2048, namely from 0 to 2047 bits) and the number of bits outside the message size (which is 32 bits in our test), and then doubling this value as we always perform two hashes: one with, and one without the specified bit set.

D. Update Test

For our third set of test cases, we took a close look at the SHA-3 Competition API specification [33] looking for potential metamorphic relations. The API specified that each implementation will have to provide four different functions: `Init()`, `Update()`, `Final()` and `Hash()`. The specification of the `Update()` API is important; it states:

> This API uses a function called `Update()` to process data using the algorithm's compression function. Whatever integral amount of data the `Update()` routine can process through the compression function is handled. Any remaining data must be stored for future processing. For example, SHA-1 has an internal structure of 512-bit data blocks. If the `Update()` function is called with 768-bits of data the first 512-bits will be processed through the compression function (with appropriate updating of the chaining values) and 256-bits will be retained for future processing. If 2048-bits of data were provided, all 2048-bits would be processed immediately. If incremental hashing is being performed, all calls to update will contain data lengths that are divisible by 8, except, possibly, the last call.

```
HashReturn Update (hashState
*state, const BitSequence *data,
DataLength databitlen)
```

Let us now look at the relevant sections of the `Hash()` function API specification, which states:

> This function shall utilize the previous three function calls, namely `Init()`, `Update()`, and `Final()`.

Let us suppose we invoke the `Hash()` function of a candidate IUT with a message m of size n; and it produces the message digest y. Assume that we break the message m into multiple fractions such as m_1, m_2, …, m_k where all sizes of any m_i are multiples of 8 except possibly for m_k. Let us suppose an invocation of `Init()`, followed by `Update(`m_1`)`, `Update(`m_2`)`, …, `Update(`m_k`)` and finally ending with the invocation of `Final()` produces the message digest y'. Following the API specification requirements, the two message digests created in the above described fashion, y and y' must be the same. Note that this is actually again a manifestation of the function behavior property of Section IV-A.

The API includes this incremental processing requirement as it is necessary for the processing of very large messages. Also, in the case of communication over a network, often a message may not be available as a whole as it is transferred through packets. A hash function implementation should not wait for the whole message to arrive completely before starting the computation of the hash value.

Taking the above property into account, we designed our third set of test cases named Update Test. In this set of test cases we divide the message m into multiple fragments and invoke `Init()`, multiple `Update(`m_i`)` and `Final()`. We compare the two message digests produced in this fashion. If they do not match, we have a scenario where the API requirements specification has not been met by the IUT.

The first and second fragments consist of message lengths ranging from 0 to 2047 bits, such that the first fragment contains all lengths that are divisible by eight, and the second fragment contains all message lengths in the range. Two hashes are computed for each combination: one where the message is processed in two fragments, and another where the same message is processed in one fragment. Our Update Test suite therefore computes $(2048/8) \times 2048 \times 2 = 2^{20}$ different hash values for each digest size of the IUT.

E. Combinatorial Update Test

Our last set of test cases is a simple variation of the Update Test described above. It was developed by utilizing insights and methods from combinatorial testing. This approach includes two primary components: (1) design of an input model for partitioning the inputs into equivalence classes; and (2) use of a covering array or sequence covering array to map combinations of values from the input model into executable tests. A variety of methods can be used in developing the input model, including classification tree methods and traditional boundary value analysis. In this case we used the Update Test to select test variable values. The ACTS (Automated Combinatorial Testing for Software) tool [34] was used to produce a two-way covering array of the input values, and then each row of the covering array was converted to an executable test. This made possible a roughly 700:1 reduction in test set size, while still finding all bugs discovered in the exhaustive testing discussed in Section IV-D.

Combinatorial testing's ability to reduce the test set size is based on the discovery that software faults "in the wild" involve only a few factors [14]. That is, most faults are triggered by one or two factors interacting with each other, with progressively fewer faults getting discovered by three or more interacting factors. This empirical observation is sometimes

TABLE III
INPUT MODEL FOR COVERING ARRAY TO DIVIDE THE MESSAGE INTO MULTIPLE FRAGMENTS

Fragments	Input values to be chosen for that fragment (in bits)
Fragment 1	0, 8, 16, 24, 32, 30, 48, 56, 64
Fragment 2	0, 8, 16, 24, 32, 30, 48, 56, 64
Fragment 3	0, 8, 16, 32, 64, 128, 256, 512, 1024, 2048
Fragment 4	0, 1, . . ., 65, 127, 128, 129, 255, 256, 257, 511, 512, 513

referred to as the *interaction rule*. So far no faults have been observed that involve more than six factors. Consequently, testing t-way combinations of variable values, for small t, can produce test sets that are as effective as exhaustive testing, but orders of magnitude smaller [35], [36].

To accomplish the judicious covering of the different message fragments, we designed the input model shown in Table III. In combinatorial testing parlance, the input model is a $9^2 10^1 75^1$ configuration (two variables of 9 values each, one variable of 10 values, and one variable of 75 values). From this model, we used the ACTS tool to generate a two-way covering array, which includes all two-way combinations of these values, containing 755 rows. Each row of the covering array is converted into a test, as described below.

Let us consider a typical row in the two-way covering array created out of the input model shown in Table III (with input sizes given in bits): *64, 48, 512, 257*. When we use this combination as a test case, we will compute `Hash(`m`)` on a message of size 881 bits. Then we will break that same message in four parts and invoke `Init()`, `Update(`m_{64}`)`, `Update(`m_{48}`)`, `Update(`m_{512}`)`, `Update(`m_{257}`)`, and `Final()`. Once we have the two message digests computed, we will compare them for equality.

As mentioned above, this test suite is designed to detect the exact same types of failures as the test suite in Section IV-D; except with a tiny fraction of the number of test cases. Our test suite, in this case, computed $2 \times 755 = 1510$ message digests for each digest size of the IUT. In our experiments, we found that these 1510 digests detected the same faults as the Update Test of Section IV-D containing $1\,048\,576$ digests.

V. RESULTS

Our four test suites discovered a number of failures for some of the test cases. We will list all IUTs that failed at least one test case in our test suites. Recall that there were 64 candidates that were submitted to the competition. After initial requirements check in the submission package, 51 of the submissions were selected for the first round. With multiple years of scrutiny, analyses, and feedback, 14 were chosen for the second round, out of which five became finalists.

Through each of the rounds, the submitters were allowed to submit minor updates to their implementations. We considered all the 51+14+5=70 versions of the submissions. Additionally, some intermediate updates were submitted to the SHA-3 competition, for example to provide an updated specification document or to fix a bug in the implementation. There were 16 such updates with changes in the source code of the reference implementation, which brings our total number of IUTs to 86.

Whenever possible, the compilation was performed on Windows 7 using GCC version 5.4.0 under Cygwin. Compilation failed for the candidate ARIRANG because the reference implementation was incompatible with the SHA-3 Competition API; however this problem was fixed in an intermediate update. For all other submissions, we succeeded at compiling the reference implementations in our build environment. We consider a candidate implementation to have failed if one of our tests detects an inconsistency, if compilation fails due to an incompatibility with the API, or if the program crashes (e.g., due to a memory leak).

Our results are shown in Tables IV and V. Out of the 86 different candidate reference implementations, 19 failed the Bit-Contribution Test, 17 failed the Bit-Exclusion Test, and 32 failed the Update Test. The Combinatorial Update Test discovered the same bugs as the Update Test, and is therefore not shown in the table. In Section VI, we will delve deeper into the nature of some of these bugs and why our approach was effective in revealing them.

VI. SOME BUG DETAILS

In this section we describe a few of the bugs found in some detail, in order to show why they escaped detection for many years. We focus mostly on bugs in the SHA-3 finalists, but also consider some implementations of prominent first-round and second-round submissions, in particular when they contained serious bugs that nevertheless remained undiscovered for a long time.

A. BLAKE

The hash function BLAKE [37] was submitted to the SHA-3 competition by Aumasson et al., and was eventually selected as one of the five finalists. As mentioned earlier, there is an implementation bug that is still present today in the source code of all BLAKE implementations on the official SHA-3 competition website [13], the first of which appeared in 2008. In 2015, the reference implementation on the BLAKE website [38] was updated with a rather cryptic statement saying that the update "fixed a bug that gave incorrect hashes in specific use cases."

In a talk at TROOPERS 2016, Aumasson [39] confirmed that the bug remained unnoticed for seven years, after which it was "found by a careful user." However, no information is provided on how the bug was found, and there is no information on the scenarios under which the bug could present itself, nor about the potential impact of the bug on real-world applications.

In his talk, Aumasson explained that "test vectors are less useful" for detecting faults when the specification is not followed. The SHA-3 Competition Test Suite has been ineffective in discovering this bug in BLAKE: it is present in all submitted implementations, and therefore all implementations (reference and optimized) produce the same results for those test vectors.

We not only independently rediscovered this bug, but we also show that it can easily be found using our testing approach using only a small set of test vectors. In particular, the Update Test detects an inconsistency between processing the message

TABLE IV
OUR TEST RESULTS FOR ALL SHA-3 COMPETITION REFERENCE IMPLEMENTATIONS[‡]

Name of the Submission	Bit-Contribution Test	Bit-Exclusion Test	Update Test	Summary
Abacus	passed	**failed**	passed	**failed**
ARIRANG	**failed**	**failed**	**failed**	**failed**
ARIRANG Update	passed	passed	passed	passed
AURORA	passed	passed	passed	passed
BLAKE	**failed**	passed	**failed**	**failed**
BLAKE Round 2	passed	passed	**failed**	**failed**
BLAKE Final Round	passed	passed	**failed**	**failed**
Blender	**failed**	passed	**failed**	**failed**
Blue Midnight Wish	passed	passed	**failed**	**failed**
Blue Midnight Wish Round 2	passed	passed	**failed**	**failed**
BOOLE	passed	passed	passed	passed
Cheetah	**failed**	passed	**failed**	**failed**
CHI	passed	passed	passed	passed
CHI Update	passed	passed	passed	passed
CRUNCH	**failed**	**failed**	**failed**	**failed**
CRUNCH Update	passed	passed	**failed**	**failed**
CubeHash	passed	passed	passed	passed
CubeHash Round 2	passed	passed	passed	passed
DCH	passed	passed	passed	passed
Dynamic SHA	passed	**failed**	passed	**failed**
Dynamic SHA2	passed	**failed**	passed	**failed**
ECHO	passed	passed	passed	passed
ECHO Round 2	passed	passed	passed	passed
ECOH	passed	passed	**failed**	**failed**
EDON-R	passed	passed	**failed**	**failed**
EDON-R Update	passed	passed	**failed**	**failed**
EnRUPT	passed	passed	**failed**	**failed**
ESSENCE	passed	passed	passed	passed
FSB	passed	passed	passed	passed
Fugue	**failed**	**failed**	passed	**failed**
Fugue Round 2	**failed**	**failed**	passed	**failed**
Fugue Round 2 Update	passed	**failed**	passed	**failed**
Grøstl	passed	passed	passed	passed
Grøstl Round 2	passed	passed	passed	passed
Grøstl Final Round	passed	passed	passed	passed
Hamsi	**failed**	passed	**failed**	**failed**
Hamsi Update	**failed**	passed	**failed**	**failed**
Hamsi Round 2	passed	passed	**failed**	**failed**
JH	**failed**	**failed**	**failed**	**failed**
JH Update	passed	**failed**	**failed**	**failed**
JH Round 2	passed	**failed**	**failed**	**failed**
JH Final Round	passed	**failed**	passed	**failed**
Keccak	passed	passed	passed	passed
Keccak Round 2	passed	passed	passed	passed
Keccak Final Round	passed	passed	passed	passed
Khichidi-1	**failed**	passed	passed	**failed**
LANE	**failed**	passed	**failed**	**failed**
Lesamnta	passed	passed	passed	passed
Luffa	passed	passed	passed	passed
Luffa Round 2	passed	passed	passed	passed
LUX	passed	passed	**failed**	**failed**
MCSSHA3	passed	passed	passed	passed
MD6	passed	passed	passed	passed
MD6 Update	passed	passed	passed	passed
MeshHash	passed	passed	passed	passed
NaSHA	**failed**	**failed**	**failed**	**failed**
NaSHA Update	**failed**	**failed**	**failed**	**failed**
SANDstorm	passed	passed	passed	passed
SANDstorm Update	passed	passed	passed	passed
Sarmal	passed	passed	passed	passed
Sgàil	passed	passed	passed	passed
Shabal	passed	passed	passed	passed
Shabal Round 2	passed	passed	passed	passed
SHAMATA	passed	passed	**failed**	**failed**
SHAvite-3	passed	passed	**failed**	**failed**
SHAvite-3 Update	passed	passed	**failed**	**failed**
SHAvite-3 Round 2	passed	passed	passed	passed
SHAvite-3 Round 2 Update	passed	passed	passed	passed
SIMD	passed	passed	passed	passed

[‡] The results of the Combinatorial Update Test are identical to those of the Update Test.

TABLE V
CONTINUATION OF TABLE IV: OUR TEST RESULTS FOR ALL SHA-3 COMPETITION REFERENCE IMPLEMENTATIONS[‡]

Name of the Submission	Bit-Contribution Test	Bit-Exclusion Test	Update Test	Summary
SIMD Update	passed	passed	passed	passed
SIMD Round 2	passed	passed	passed	passed
Skein	passed	passed	passed	passed
Skein Update	passed	passed	passed	passed
Skein Round 2	passed	passed	passed	passed
Skein Final Round	passed	passed	passed	passed
Spectral Hash	**failed**	passed	**failed**	**failed**
Spectral Hash Update	**failed**	passed	**failed**	**failed**
Stream Hash	**failed**	**failed**	passed	**failed**
SWIFFTX	passed	passed	passed	passed
Tangle	passed	passed	passed	passed
TIB3	passed	passed	passed	passed
Twister	**failed**	passed	**failed**	**failed**
Vortex	**failed**	**failed**	**failed**	**failed**
Vortex Update	passed	**failed**	**failed**	**failed**
WaMM	passed	passed	passed	passed
Waterfall	passed	passed	passed	passed
Total Failed	19	17	32	41

[‡] The results of the Combinatorial Update Test are identical to those of the Update Test.

at once using `Hash()`, and processing it in variable-sized blocks using `Init()`, several calls to `Update()`, and then `Final()` to produce the message digest.

This BLAKE bug affects all output sizes of the hash function. The `Update32()` function of the reference and optimized implementations contains the following incorrect statement:

```
if( left && ( ((databitlen >> 3) & 0x3F)
                              >= fill ) ) {
```

The BLAKE design team proposes to fix the bug by omitting the `& 0x3F`, thereby correcting the statement as follows:

```
if( left && ( ((databitlen >> 3) )
                              >= fill ) ) {
```

The 224-bit and 256-bit output sizes of BLAKE use the `Update32()` function to process the message in blocks of 64 bytes. As there are $2^3 = 8$ bits in one byte, the shift operation `(databitlen >> 3)` represents the number of complete bytes of the message data that is to be processed by the `Update32()` function. Followed by `& 0x3F`, we have the number of complete bytes modulo the block size of 64 bytes.

In `Update64()`, which is used to provide the 384-bit and 512-bit output sizes of BLAKE, the message is processed in blocks of 128 bytes. Here, we find the exact same bug in the reference and optimized implementations, except that the corresponding statement contains `0x7F` instead of `0x3F`.

It seems unlikely that the `& 0x3F` and `& 0x7F` in BLAKE would raise suspicion during a careful line-by-line inspection of the source code. In fact, we could even say that the bug is not in these particular lines of code. Rather, the bug is due to the interaction with other parts of the code, which create a specific corner case that is not handled correctly.

To see this, let us describe an example input scenario where the bug manifests itself. First, assume that `Update()` is called on one byte. For a 224-bit or 256-bit output value, the call to `Update()` is passed on to `Update32()`. As one byte is clearly less than one complete block of 64 bytes, it will be necessary to store this byte in a buffer. Then, assume that `Update32()` is called again, this time on a full block of 64 bytes.

The variable `left` in the BLAKE implementation keeps track of the number of bytes in this buffer, which in our case is one. The "unused" bytes in the buffer are stored in `fill`, which is set to `64 - left`, or 63 in our example. Given that `64 & 0x3F` equals zero, which is smaller than 63, the if-clause is not executed.

Right after the if-clause, there is a `while( databitlen >= 512 )` that will evaluate to true, as we process a block of 64 bytes, which is 512 bits. In this specific case, a 64-byte block is processed that contains the 64-byte input for the current call of `Update32()`, without taking the byte in the buffer into account. In fact, the byte in the buffer will be "forgotten," as the variable `state->datalen` that keeps track of the number of bytes in the buffer is set to 0 at the end of `Update32()`.

This creates an inconsistency with a single call to `Update32()` with a corresponding 65-byte message, and therefore results in an incorrect implementation. But the problem is actually worse, as we can now trivially create a second preimage: given one message and its corresponding hash value, it is easy to produce another message (processed using a specific sequence of `Update()` calls) that results in the same hash value.

To see this, recall from our example that the first call to `Update32()` using a one-byte message has no effect when it is followed by a second call to `Update32()` on a 64-byte block. Therefore, we can omit this first call `Update32()` without affecting the internal state of the hash function. When the hash value is calculated using `Final()`, it will be the same, regardless of whether the one-byte call to `Update32()` was performed.

B. LANE

The hash function LANE was designed by Indesteege [40]. It was selected for the first round of the SHA-3 competition, but did not advance to the second round. We independently rediscovered a bug in all implementations of LANE (reference and optimized) that are available on the official SHA-3 competition website [13].

In 2009, Cornel Reuber informed the LANE designer of this particular bug [41]. Afterwards, the implementation on the LANE website [42] was updated, without mentioning the bug on the LANE website, the updated source code, or the supporting documentation. No public announcement was made either, and the LANE implementation on the SHA-3 website currently still contains this bug. For this reason, we expect that no more than a handful of people are currently aware that the implementation on the official SHA-3 website contains this particular bug.

This bug in the LANE implementation is extremely damaging, because it will ensure that the same hash value will be returned for all messages of a particular length, regardless of the content of the message. This makes it trivial to construct second preimages for the implementation of the hash function: for the problematic message lengths, any message of a particular length can be replaced by another message of the same length, without affecting the hash value.

We easily rediscovered the bug using the Bit-Contribution Test, which flips the input bits of a message and checks if the output hash value is altered. We note that for LANE, all hash outputs are distinct in the SHA-3 Competition Test Suite. This is because the those KAT test vectors only contain one message of any particular length. The MCT contain several hash values for 1024-bit messages, however the bug that we will describe is not present for this message length.

For now, let us consider LANE with 224-bit and 256-bit hash output sizes. A similar bug will appear for the 384-bit and 512-bit hash outputs, as we will briefly describe later. The message in LANE is processed in blocks of `BLOCKSIZE` bytes. For 224-bit and 256-bit outputs, `BLOCKSIZE` equals 64. The bug that we will now describe, appears for messages whose length (in bits) is 505 to 511 modulo 512. Later, we will revisit the 384-bit and 512-bit output sizes, and describe a bug for messages of length (in bits) 1017 to 1023 modulo 1024.

The bug appears in the following line:

```
const DataLength n =(((state->databitcount
                       - 1) >> 3) + 1) & 0x3f;
```

In the updated implementation on the LANE website, the bug is fixed as follows:

```
const DataLength n =(((state->databitcount
                      & 0x1ff) - 1) >> 3) + 1;
```

As messages are processed in blocks of `BLOCKSIZE` bytes, the hash function implementation maintains a buffer to store incomplete blocks that still need to be processed. The variable `n` is intended to compute the number of bytes in this buffer.

When `state->databitcount` is 505 to 511 modulo 512, there are `BLOCKSIZE` bytes in the buffer. The incorrect implementation, however, returns 0 instead of `BLOCKSIZE`. For all other values of `state->databitcount`, the old and the updated implementation give the same value for `n`.

Now, let us see what happens in the padding routine, which zeroes out the last `BLOCKSIZE-n` bytes of the buffer:

```
memset(state->buffer + n, 0, BLOCKSIZE-n);
```

For the problematic message lengths, this padding routine should not overwrite any data in the buffer. However, the incorrect implementation zeroes out the entire buffer. As a result of this, the hash value will be independent of the content of the message for these particular message lengths.

When a 384-bit or 512-bit hash output is requested, the problematic code is also present, however with `& 0x7f` instead of `& 0x3f`. For these hash output sizes, every message of length 1017 to 1023 modulo 1024 will return the same hash value.

We note that fixing this particular bug is not sufficient to guarantee that the implementation is correct. In fact, our Update Test found that the latest reference implementation of LANE contains another bug, which is currently still present on the SHA-3 competition website as well as on the designer's website.

C. Fugue

Fugue [43] was designed by Halevi, Hall and Jutla of IBM T.J. Watson Research Center. It was submitted to the SHA-3 competition, where it became one of the fourteen candidates that advanced to the second round in July 2009.

In October 2009, Jutla made an announcement to the official SHA-3 competition mailing list, where he pointed out that Stefan Tillich discovered a bug in the Fugue implementation. In particular, for all messages for which the length (in bits) is not divisible by eight, the implementation erroneously zeroes out the last incomplete byte. Jutla clarifies that the bug is present in all implementations.

This bug makes it trivial to generate second preimages for all Fugue implementations. In particular, if two messages contain a difference in the last incomplete byte, the difference will be zeroed out, and the hash output will be identical. We rediscovered this bug using the Bit-Contribution Test, which detects whether a difference in the last incomplete byte affects the hash output. The KATs that were submitted by the Fugue team to NIST were wrong, but they do not contain any collisions. This is perhaps not so surprising, as the SHA-3 Competition KATs do not contain more than one message of a particular length.

The bug is present in the following line in `SHA3api_ref.c`:

```
memset((uint8*)state->Partial
+ ((state->TotalBits&31)/8),0,need/8);
```

When the number of bits in a message is not a multiple of eight, the number of bytes should be rounded up, instead of rounded down. This bug is corrected in the updated Fugue implementation as follows:

```
memset((uint8*)state->Partial
+ (((state->TotalBits&31)+7)/8),0,need/8);
```

Note that our Bit-Exclusion Test discovered another bug in the latest Fugue implementation, which will not be discussed in this paper.

D. Hamsi

Hamsi [44] is a hash function designed by Küçük that advanced to the second round of the SHA-3 competition.

All Hamsi implementations that were submitted to the first round of the SHA-3 competition contained a very serious bug for the 384-bit and 512-bit hash outputs, due to the fact that only half of the message bits were processed. Using the Update Test, this bug can easily be discovered. Note that this particular Hamsi bug was already pointed out by Mouha [45], however he performed no systematic analysis of unused message bits for the SHA-3 candidate implementations. Subsequently, the designer fixed this bug in the second-round implementation.

There are still bugs present in the latest second-round reference implementation on the SHA-3 competition website. To the best of our knowledge, this paper is the first to explain these bugs. The bugs are related to the implementation of `Update()`, and were caught using the Update Test.

In Hamsi, messages are processed in blocks of `s_blocksize` bits, which is 32 or 64 depending on size of the hash output. Incomplete blocks are stored in a buffer, and the Hamsi implementation uses `state->leftbits` to keep track of the number of bits in this buffer between subsequent calls to `Update()`. The `bits2hash` variable is initially the number of bits to be processed by the call to `Update()`, but this value is decreased when message bits are processed.

We found that there are at least four different bugs in the second-round implementation of `Update()`, that all need to be fixed in order to pass the Update Test:

- The `state->leftbits = bits2hash` statement at the end of the function is incorrect. For a zero-byte call to Update(), `bits2hash` would then be set to zero. But in that case, any data that might be in the buffer is discarded. The correct statement should be `state->leftbits += bits2hash`.
- A `state->leftbits += bits2hash` occurs earlier in the code of `Update()`, but the `bits2hash` value is not set to zero. This leads to an inconsistency: in this code, the `bits2hash` bits have already been processed (by moving them to the buffer), so `bits2hash` should be set to zero.
- After processing a full block of data, there is a specific case in which internal counter `state->counter` is not updated. This leads to an incorrect calculation of the length of the message (which is used in the padding), and therefore an incorrect hash output.
- When data is copied to the buffer, the size of a block is expressed in bits instead of bytes. Therefore, `s_blocksize` should be replaced by `s_blocksize/8`. This causes a write to an array index that is out of bounds. This bug could have been detected by memory error detection tools (such as those used in the Fortify Report [31]). However, the offending code is not accessed by the SHA-3 Competition Test Suite, in which case it would not be detected by dynamic program analysis tools.

E. JH

The hash function JH [46] was designed by Hongjun Wu, and was selected as one of the five finalists of the SHA-3 competition. We have found the presence of a bug in all submitted versions of JH to the competition.

For messages that are not a multiple of eight bits, all JH implementations read bits outside of the bounds of the message. We detected this bug using the Bit-Exclusion Test, which flips input bits that are beyond the boundary of the message, and detects if the corresponding hash value has changed.

For all JH implementations (reference and optimized) submitted to each round of the competition, the padding does not follow the JH specification. According to the specification, the bit ‘1’ should be appended to the end of the message, followed by a specific number of zero bits. In the implementation of `Final()`, the bit ‘1’ is added, but no zero bits are added in case of a message that is not a multiple of eight bits. Instead, the implementation will use the input bits beyond the boundary of the message for the remainder of this byte.

For the final-round submission of JH, a source code comment was added inside the `Update()` function, which indicates that the author is aware of this issue. More specifically, it states: “assume that – if part of the last byte is not part of the message, then that part consists of 0 bits.” This assumption violates the SHA-3 Competition API specification, which does not assume that bits outside of the message are set to zero.

No mention of this unexpected behavior is made in the change-log or the supporting documentation. Still, this behavior should clearly be considered to be a bug: input bits that are not part of the message have an effect on the final hash value, which means that the implementation is not consistent with the SHA-3 Competition API specification.

Interestingly, we found another bug in all first-round and second-round implementations of JH, that was only fixed in the final-round implementation. Before the final round, the implementation only supported calls to `Update()` that are multiples of 512 bits, except for the last call to `Update()`. The functionality to handle other combinations of calls to `Update()` was not implemented.

In the SHA-3 Competition KATs, only one call to `Update()` is performed. The MCTs perform many calls to `Update()`, but each call consists of a message of 1024 bits. For this reason, the bug was not revealed in the SHA-3 Competition Test Suite, although our Update Test detects it easily. Note that it is unlikely to detect this bug using test cases that are based only on code coverage criteria. Coverage analysis helps to detect source code that is unreachable with certain test vectors, but it cannot be used to detect missing code.

VII. Code Coverage Analysis

To the best of our knowledge, there have not been any efforts to analyze the SHA-3 candidate implementations using code coverage analysis. The goal of code coverage analysis is to find areas of a program that are not exercised by the test vectors, and the creation of additional test cases to increase coverage, and thus to strengthen the test suite. Code coverage analysis is essential in verification of most safety or security-critical software, but may not be effective for some aspects of cryptographic algorithms. As noted by Langley [47]: "cryptographic functions result in abnormally straight line code, it's common for a typical input to exercise every instruction."

Nevertheless, we have identified an area where the code coverage is meaningful and could be improved compared to the SHA-3 Competition Test Suite, at least for some SHA-3 candidate implementations. The SHA-3 Competition testing approach does not always access all instructions, in particular when the API is used to process a message "on-the-fly" in several chunks, instead of providing the entire message at once.

Using GCC in combination with its gcov tool, we computed the statement coverage and branch coverage of the SHA-3 Competition Test Suite for the SHA-3 finalists, and compared them to the coverage of our Update Test.

As shown in Table VI, for Grøstl, JH and Skein, the coverage of our Update Test is slightly higher than that of the SHA-3 Competition Test Suite. This is because our test reaches an additional corner case related to the processing of a message in chunks using multiple calls to `Update()`. For Keccak, the branch and statement coverage is the same for both test suites. Interestingly, for BLAKE, our tests cover one statement less, related to the handling of an overflow in a 32-bit addition. This line is only executed when a very large message is processed, which does not happen in our test suite.

NIST encouraged submissions that provide additional functionalities, e.g., having not only a message input, but also an additional salt or key input. The C functions that implement such features are not part of the API, and are therefore not reached by the SHA-3 Competition Test Suite. Furthermore, some submissions support additional digest sizes (besides the four required ones), or messages that are longer than $2^{64} - 1$ bits. To finish our tests in a reasonable time, we turned on compiler optimizations (using the command-line flag `-O3`), which resulted in unreachable statements when functions are inlined. These are the main reasons why some source code files in Table VI have a relatively low coverage. Nevertheless, we found that SHA-3 candidates typically achieve complete code coverage of all API-required functionality, as expected for cryptographic functions.

VIII. Discussion

We have gained some important insights from our test suite design, experiment results, and from the detailed investigations of some of the bugs in the implementations. Ensuring the correctness of cryptographic algorithm implementations is of crucial importance to the security of information systems. To accomplish this goal for NIST-recommended cryptographic algorithms, the NIST Cryptographic Algorithm Validation Program (CAVP) has been established [27]. Our objective was to determine if we can improve those tests by applying different methods, while ensuring that the new tests would be practical in terms of test development and execution resources. We found that we could produce significant improvements without requiring a prohibitive amount of time for test development, and the resulting test set size is practical for real-world use. Note that metamorphic testing has been shown to detect bugs, even in well-tested and mature software systems, because it is based on a "diverse" perspective that is not previously used by other testing techniques [48].

Another important goal of this effort was to provide a systematic approach to test cryptographic algorithm implementations, i.e., scientific test and analysis techniques, which can be defined as "efficient, rigorous test strategies that will yield defensible results [49]." In practice, this can be achieved through approaches such as statistical testing and design of experiments methods. Cryptographic software, particularly hash algorithms, tends to consist of many bit manipulations. As discussed earlier, measurable test criteria such as statement or branch coverage do not provide much assurance for the type of software we tested, as typical cryptographic implementations have full branch coverage of the functionality that is required by the API. Furthermore, due to the lack of a test oracle, the KATs and MCTs in the SHA-3 Competition Test Suite failed to detect the bugs that are pointed out in this paper.

Despite the comprehensive SHA-3 Competition Test Suite, it did not include certain input combinations that are necessary to reveal bugs using our testing methodology. In particular, the SHA-3 Competition Test Suite either only performs one call to `Hash()` to process an entire message in the KATs, or processes a message in several calls to `Update()`, each with 1024 bits of data, in the MCTs. In our Update Test, we cover a wide range of scenarios where a message is processed in chunks, combined with the metamorphic testing approach to check for consistency. Unlike the SHA-3 Competition KATs, we also test more than one message of a particular length, which allows additional bugs to be found.

In our Bit-Contribution Test, we consider all messages of a particular size that have exactly one bit set (as in SP 500-20 for DES, see Section III), but additionally we also include the all-zero vector. When the all-zero vector is present, it becomes possible to compare the hash values of two messages that differ in one bit, and confirm that the corresponding hash value is changed.

We also introduce differences in the message buffer *beyond* the bits that are reserved for the message in the Bit-Exclusion Test, in order to check that the corresponding hash values are *not* changed. The use of memory error detection tools can detect out-of-bound *byte* reads, but they require access to the source code which may not be available. Furthermore, the Bit-Exclusion Test also checks if the hash value is altered through out-of-bound *bit* reads inside the last byte, an API-specific issue that cannot be revealed by memory error detection tools.

The strategy of the SHA-3 Competition Test Suite was borrowed from the CAVP for SHA-2. As explained in Section III,

TABLE VI
STATEMENT AND BRANCH COVERAGE OF THE SHA-3 COMPETITION TEST SUITE, COMPARED TO OUR UPDATE TEST SUITE[§]

Name of the SHA-3 Finalist	Statement Coverage			Branch Coverage		
	Number of Statements	SHA-3 Test Coverage	Update Test Coverage	Number of Branches	SHA-3 Test Coverage	Update Test Coverage
BLAKE						
⊢ `blake_ref.c`	348	96.26%	95.98%	116	92.24%	91.38%
Grøstl						
⊢ `Groestl-ref.c`	152	94.08%	96.05%	157	89.17%	91.72%
JH						
⊢ `jh_ref.h`	144	93.75%	100.00%	81	87.65%	97.53%
Keccak						
⊢ `KeccakF-1600-reference.c`	127	72.44%	72.44%	60	73.33%	73.33%
⊢ `KeccakNISTInterface.c`	30	93.33%	93.33%	18	66.67%	66.67%
⊢ `KeccakSponge.c`	106	82.08%	82.08%	78	66.67%	66.67%
Skein						
⊢ `SHA3api_ref.c`	38	68.42%	68.42%	18	38.89%	47.06%
⊢ `skein.c`	270	20.74%	22.22%	102	19.61%	20.59%
⊢ `skein_block.c`	183	30.60%	30.60%	42	33.33%	33.33%

[§] Several implementations have functions that are not part of the API, and are therefore not reached by the test suites.

this test suite was designed to ensure structural coverage of any table that may be used in the implementation by exercising every table entry. We note that such structural coverage would still be ineffective for this type of programs due to the non-existence of a test oracle. Simply observing a message digest or hash value produced by a test vector does not provide any information about its correctness.

Our intention is not to criticize the existing approaches to testing cryptographic functions, but to point out their shortcomings and suggest several areas of improvement. Our approach provides insights into how to design highly effective test suites for testing cryptographic algorithms, and to significantly reduce the number of required test cases without sacrificing their fault-finding effectiveness.

An interesting topic for future work is to extend our testing approach to other cryptographic algorithms. In general, when certain input bits must have a high probability to affect certain output bits, the Bit-Contribution Test may be used. When certain input bits should not affect the output, the Bit-Exclusion Test may be considered. The Update Test can be useful to test the processing of a message in multiple fragments. Note that domain-specific knowledge about cryptographic algorithms is required to develop a testing approach, and that the specific customization and the effectiveness of the approach is dependent on the specification of that particular algorithm and the properties that may be used to develop the tests.

IX. CONCLUSIONS

Based on the framework of metamorphic testing, we have described a systematic method for developing tests for cryptographic hash function implementations, and applied this method to test all software implementations submitted to the NIST SHA-3 competition. The test development method begins with the main cryptographic properties that hash functions should satisfy: (second-)preimage resistance, collision resistance, and function behavior. We devise test cases that try to violate these properties for the *implementations* of these hash functions.

To assess the effectiveness of our tests in discovering bugs, we revisited the NIST SHA-3 competition. NIST determined that 51 submissions to the SHA-3 competition met the minimum submission requirements, and made them available on-line. In spite of the initial selection and testing by the submitters and by NIST, we have found bugs in 25 out of the 51 initial reference implementations. The percentage of bugs that we found is more or less consistent for the updated reference implementations, where we have found bugs in 16 out of 35 updated implementations.

One of the primary strengths of our approach is that it uses a practical number of test cases, yet detects complex flaws that were either previously unknown or only discovered years after release. Our tests are also roughly four times faster than the SHA-3 Competition Test Suite on a single core, even though we compute a larger number of hash values. This is primarily due to the fact that our test cases involve relatively short messages as input. Furthermore, unlike some traditional test suites developed for verifying cryptographic functions, our test cases are independent of each other and thus can easily be run in parallel. Finally, our tests can also be applied in the practically-relevant scenario where the source code or hardware description of the implementations are not available.

ACKNOWLEDGMENT

The authors would like to thank Lily Chen, Çağdaş Çalık, Meltem Sönmez Turan, Apostol Vassilev, and their NIST colleagues for their useful comments and suggestions. This work was partially supported by NIST ITL grant 70NANB17H035. Products may be identified in this document, but identification does not imply recommendation or endorsement by NIST, nor that the products identified are necessarily the best available for the purpose. Responsible disclosure: to the best of our knowledge, none of the SHA-3 candidate algorithms are used in commercial products, except for Keccak, which became the SHA-3 standard. However, we did not identify bugs in the reference implementations of Keccak.

References

[1] National Institute of Standards and Technology, "Secure Hash Standard (SHS)," NIST Federal Information Processing Standards Publication 180-4, p. 36, August 2015, http://dx.doi.org/10.6028/NIST.FIPS.180-4.

[2] X. Wang and H. Yu, "How to Break MD5 and Other Hash Functions," in *EUROCRYPT 2005*, ser. Lecture Notes in Computer Science, vol. 3494. Springer, 2005, pp. 19–35.

[3] X. Wang, Y. L. Yin, and H. Yu, "Finding Collisions in the Full SHA-1," in *CRYPTO 2005*, ser. Lecture Notes in Computer Science, vol. 3621. Springer, 2005, pp. 17–36.

[4] National Institute of Standards and Technology, "Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA-3) Family," 72 Fed. Reg., pp. 62 212–62 220, November 2007, https://www.federalregister.gov/d/E7-21581.

[5] ——, "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions," NIST Federal Information Processing Standards Publication 202, p. 37, August 2015, https://doi.org/10.6028/NIST.FIPS.202.

[6] E. J. Weyuker, "On Testing Non-Testable Programs," *Comput. J.*, vol. 25, no. 4, pp. 465–470, 1982.

[7] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The Oracle Problem in Software Testing: A Survey," *IEEE Trans. Softw. Eng.*, vol. 41, no. 5, pp. 507–525, May 2015.

[8] Z. Q. Zhou, D. H. Huang, T. H. Tse, Z. Yang, H. Huang, and T. Y. Chen, "Metamorphic Testing and its Applications," in *Proc. of the 8th International Symposium on Future Software Technology (ISFST 2004)*, 2004.

[9] T. Y. Chen, T. H. Tse, and Z. Q. Zhou, "Fault-based testing without the need of oracles," *Information and Software Technology*, vol. 45, no. 1, pp. 1–9, 2003.

[10] S. Segura, G. Fraser, A. Sanchez, and A. Ruiz-Cortés, "A Survey on Metamorphic Testing," *IEEE Trans. Softw. Eng.*, vol. 42, no. 9, pp. 805–824, Sept. 2016.

[11] T. Y. Chen, F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. H. Tse, and Z. Q. Zhou, "Metamorphic testing: A review of challenges and opportunities," *ACM Computing Surveys*, vol. 51, no. 1, pp. 4:1–4:27, 2018.

[12] CERT/CC, "CA-1999-15: Buffer Overflows in SSH daemon and RSAREF2 Library," http://www.cert.org/historical/advisories/CA-1999-15.cfm, December 1999.

[13] National Institute of Standards and Technology, "NIST Computer Security Division - The SHA-3 Cryptographic Hash Algorithm Competition, November 2007 - October 2012," February 2017. [Online]. Available: http://csrc.nist.gov/groups/ST/hash/sha-3/

[14] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, "Software Fault Interactions and Implications for Software Testing," *IEEE Trans. Softw. Eng.*, vol. 30, no. 6, pp. 418–421, 2004.

[15] R. Anderson, "Why Cryptosystems Fail," in *Proceedings of the 1st ACM Conference on Computer and Communications Security*. ACM, 1993, pp. 215–227.

[16] A. Braga and R. Dahab, "A Survey on Tools and Techniques for the Programming and Verification of Secure Cryptographic Software," *Proceedings of XV SBSeg*, pp. 30–43, 2015.

[17] D. Lazar, H. Chen, X. Wang, and N. Zeldovich, "Why does cryptographic software fail? A case study and open problems," in *Proceedings of 5th Asia-Pacific Workshop on Systems*. ACM, 2014, p. 7.

[18] P. Kitsos, D. E. Simos, J. Torres-Jimenez, and A. G. Voyiatzis, "Exciting FPGA cryptographic Trojans using Combinatorial Testing," in *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*. IEEE, 2015, pp. 69–76.

[19] D. E. Simos, R. Kuhn, A. G. Voyiatzis, and R. Kacker, "Combinatorial Methods in Security Testing," *IEEE Computer*, vol. 49, no. 10, pp. 80–83, 2016.

[20] J. Botella, F. Bouquet, J.-F. Capuron, F. Lebeau, B. Legeard, and F. Schadle, "Model-Based Testing of Cryptographic Components – Lessons Learned from Experience," in *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*. IEEE, 2013, pp. 192–201.

[21] T. Y. Chen, F.-C. Kuo, W. Ma, W. Susilo, D. Towey, J. Voas, and Z. Q. Zhou, "Metamorphic Testing for Cybersecurity," *Computer*, vol. 49, no. 6, pp. 48–55, 2016.

[22] C.-a. Sun, Z. Wang, and G. Wang, "A property-based testing framework for encryption programs," *Frontiers of Computer Science*, vol. 8, no. 3, pp. 478–489, 2014.

[23] J.-P. Aumasson and Y. Romailler, "Automated Testing of Crypto Software Using Differential Fuzzing," in *Black Hat USA 2017*, July 2017.

[24] E. W. Dijkstra, "Chapter I: Notes on Structured Programming," in *Structured Programming*, O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Eds. London, UK, UK: Academic Press Ltd., 1972, pp. 1–82.

[25] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Computing Surveys*, vol. 29, no. 4, pp. 366–427, 1997.

[26] National Institute of Standards and Technology, "Security Requirements for Cryptographic Modules," NIST Federal Information Processing Standards Publication 140-1, p. 56, January 1994, http://csrc.nist.gov/publications/fips/fips140-1/fips1401.pdf.

[27] ——, "NIST - Cryptographic Algorithm Validation Program (CAVP)," June 2017. [Online]. Available: http://csrc.nist.gov/groups/STM/cavp

[28] ——, "Announcing Request for Candidate Algorithm Nominations for the Advanced Encryption Standard," 62 Fed. Reg., pp. 48 051–48 058, September 1997, https://www.federalregister.gov/d/97-24214.

[29] ——, "Announcing the ADVANCED ENCRYPTION STANDARD (AES)," NIST Federal Information Processing Standards Publication 197, p. 51, November 2001, https://doi.org/10.6028/NIST.FIPS.197.

[30] National Bureau of Standards, "Validating the Correctness of Hardware Implementations of the NBS Data Encryption Standard," NBS Special Publication 500-20, p. 52, November 1977, http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nbsspecialpublication500-20e1977.pdf.

[31] J. Forsythe and D. Held, "NIST SHA-3 Competition Security Audit Results," Fortify Software Blog, 2009, archived at: http://web.archive.org/web/20120222155656if_/http://blog.fortify.com/repo/Fortify-SHA-3-Report.pdf.

[32] A. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 1997.

[33] National Institute of Standards and Technology, "ANSI C Cryptographic API Profile for SHA-3 Candidate Algorithm Submissions," February 2008. [Online]. Available: http://csrc.nist.gov/groups/ST/hash/documents/SHA3-C-API.pdf

[34] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "IPOG/IPOG-D: efficient test generation for multi-way combinatorial testing," *Software Testing, Verification and Reliability*, vol. 18, no. 3, pp. 125–148, 2008.

[35] D. R. Kuhn, R. N. Kacker, and Y. Lei, *Introduction to Combinatorial Testing*. CRC press, 2013.

[36] C. Montanez, D. R. Kuhn, M. Brady, R. M. Rivello, J. Reyes, and M. K. Powers, "Evaluation of Fault Detection Effectiveness for Combinatorial and Exhaustive Selection of Discretized Test Inputs," *Software Quality Professional Magazine*, vol. 14, no. 3, 2012.

[37] J.-P. Aumasson, L. Henzen, W. Meier, and R. C.-W. Phan, "SHA-3 proposal BLAKE," Submission to the NIST SHA-3 Competition (Round 3), 2010, http://131002.net/blake/blake.pdf.

[38] J.-P. Aumasson, "SHA-3 proposal BLAKE," September 2015. [Online]. Available: https://131002.net/blake/

[39] ——, "Crypto Code: The 9 circles of testing," TROOPERS 2016, March 2016. [Online]. Available: https://131002.net/data/talks/cryptocode_troopers16.pdf

[40] S. Indesteege, "The Lane hash function," Submission to the NIST SHA-3 Competition (Round 1), 2008, http://www.cosic.esat.kuleuven.be/publications/article-1181.pdf.

[41] ——, Personal Communication, July 2009.

[42] ——, "The Lane hash function," July 2009. [Online]. Available: https://www.cosic.esat.kuleuven.be/lane/

[43] S. Halevi, W. E. Hall, and C. S. Jutla, "The Hash Function Fugue," Submission to the NIST SHA-3 Competition (Round 2), 2009, archived at: http://www.webcitation.org/5nRO4qvXN.

[44] Ö. Küçük, "The Hash Function Hamsi," Submission to the NIST SHA-3 Competition (Round 2), 2009, http://www.cosic.esat.kuleuven.be/publications/article-1203.pdf.

[45] N. Mouha, "Automated Techniques for Hash Function and Block Cipher Cryptanalysis," Ph.D. dissertation, Katholieke Universiteit Leuven, 2012.

[46] H. Wu, "The Hash Function JH," Submission to the NIST SHA-3 Competition (Round 3), 2011, http://www3.ntu.edu.sg/home/wuhj/research/jh/jh_round3.pdf.

[47] A. Langley, "Checking that functions are constant time with Valgrind," April 2010. [Online]. Available: https://www.imperialviolet.org/2010/04/01/ctgrind.html

[48] T. Y. Chen, F.-C. Kuo, D. Towey, and Z. Q. Zhou, "A Revisit of Three Studies Related to Random Testing," *Science China Information Sciences*, vol. 58, no. 5, pp. 1–9, May 2015.

[49] Air Force Institute of Technology, "Scientific Test and Analysis Techniques Definition," 2017. [Online]. Available: https://www.afit.edu/STAT/page.cfm?page=358