IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal
SPECIAL SECTION ON SECURITY ANALYTICS AND INTELLIGENCE FOR CYBER PHYSICAL SYSTEMS

Received April 11, 2017, accepted April 22, 2017, date of publication May 10, 2017, date of current version June 7, 2017.

Digital Object Identifier 10.1109/ACCESS.2017.2697918

Verification of Implementations of Cryptographic Hash Functions

DEXI WANG[1], YU JIANG[1], HOUBING SONG[2], FEI HE[1], MING GU[1], AND JIAGUANG SUN[1]

[1]School of Software, Tsinghua University, Beijing 100084, China
[2]Department of Electrical and Computer Engineering, West Virginia University Institute of Technology, Montgomery, WV 25136 USA

Corresponding author: Dexi Wang (dx-wang12@mails.tsinghua.edu.cn)

ABSTRACT Cryptographic hash functions have become the basis of modern network computing for identity authorization and secure computing; protocol consistency of cryptographic hash functions is one of the most important properties that affect the security and correctness of cryptographic implementations, and protocol consistency should be well proven before being applied in practice. Software verification has seen substantial application in safety-critical areas and has shown the ability to deliver better quality assurance for modern software; thus, applying software verification to a protocol consistency proof for cryptographic hash functions is a reasonable approach to prove their correctness. Verification of protocol consistency of cryptographic hash functions includes modeling of the cryptographic protocol and program analysis of the cryptographic implementation; these require a dedicated cryptographic implementation model that preserves the semantics of the code, efficient analysis of cryptographic operations on arrays and bits, and the ability to verify large-scale implementations. In this paper, we propose a fully automatic software verification framework, VERIHASH, that brings software verification to protocol consistency proofs for cryptographic hash function implementations. It solves the above challenges by introducing a novel cryptographic model design for modeling the semantics of cryptographic hash function implementations, extended array theories for analysis of operations, and compositional verification for scalability. We evaluated our verification framework on two SHA-3 cryptographic hash function implementations: the winner of the NIST SHA-3 competition, Keccack; and an open-source hash program, RHash. We successfully verified the core parts of the two implementations and reproduced a bug in the published edition of RHash.

INDEX TERMS Model-predictive control, smoothening, gradient-based optimization, emission control, urban traffic control.

I. INTRODUCTION

Cryptographic hash function theory plays an important role in modern computing; it is the basis of a secure network, which includes identity verification, file integrity checking, secure key passing, and source code version control. A cryptographic hash function protocol is usually designed to ensure the correctness of its hash algorithms, and it will be well tested before being applied in industry, so the correctness of a cryptographic hash function protocol itself is not a problem. However, the correctness of implementations of cryptographic hash functions themselves is important; implementations are usually written manually by software developers with differing levels of experience and are the actual programs or libraries that are used in the cryptographic activities mentioned above. Thus, the protocol consistency of an implementation of a cryptographic hash function determines whether software applications will behave as expected, whether it is vulnerable to different types of attacks and whether it will simply fail because of an artificial implementation flaw. It would be ideal if one could prove that implementations in cryptographic hash functions are free of bugs and are consistent with the original protocols.

Software verification has seen heavy application in safety-critical software systems and has demonstrated the ability to ensure better implementation quality of programs by providing verification results with counter-examples generated by a software verifier. However, applying software verification to implementations of cryptographic hash functions need to address the following challenges. (i) A verification model is required to represent the semantics of the cryptographic hash function implementations, and the model should be well-defined to preserve important protocol information and easily

verified by the verification engine. (ii) Cryptographic hash function implementations usually operate on arrays of bits and attempt to implement the protocol in a compact style, making its code difficult to test or its array operations difficult to analyze, so traditional array theory in software verification must be extended to support operations on large arrays. (iii) Many cryptographic hash function protocols may introduce loops and recursions, together with different steps of permutation; if the implementation structure is too complex, the software verifiers may be faced with the problem of the explosion of the state space. Thus, a scalable verification method is needed for large-scale analysis.

Several verification on cryptographic protocols or cryptographic implementations address the above challenges. A formal verification of high-speed cryptographic software Curve25519 has been applied to an elliptic curve cryptography assembly [1]. An error in a previously published edition was found by the method and reproduced by the verifier. This work was based on modeling the elliptic curve cryptography structure and solved by SMT solvers; it also required manual interaction when constructing verification condition and constraint solving. An automatic verification tool [2] has been implemented in a compiler of cryptographic protocols. The verification was done by translating cryptographic protocols into rewrite rules that can be used by several types of automatic or semi-automatic tools to find protocol flaws. Cryptographic hash functions were not yet supported by the work, and the tool verifies only protocols, not implementations. We feel it would be imporved if the entire procedure were fully automatic. Additionally, there are ways to solve the verification problem from a modeling perspective [3]–[8], but a direct verification of the source code is still needed to prove protocol consistency.

To verify implementations in cryptographic hash functions, we propose a fully automatic software verification framework, VERIHASH, that integrates model checking techniques, extended array theory, and compositional verification to solve the challenges discussed above. An intermediate program model, ELTS, was designed to represent the semantics of the cryptographic implementations; it is also suitable for the state-space exploration and abstraction needed by the verification engine. Extended array theory makes it possible to perform verification on array manipulation and bit operations; thus, the framework can analyze array operations efficiently. By applying compositional verification, VERIHASH divides an implementation of a cryptographic hash function into different verification components according to its computation procedure and structure; it then combines the verification results of different components compositionally.

We experimented with VERIHASH on two open-source implementations of the SHA-3 hash function protocol, Keccack and RHash. Keccack won the NIST SHA-3 competition and is available in multiple implementations for different architectures, in both C and assembly language. RHash is a widely used cross-platform open-source console utility for computing and verifying hash sums of files; support for SHA-3 was recently added to its hash- function family. With a modified version of the open-source software verifier CPAchecker within our framework, we successfully verified the core parts of the two implementations and reproduced an error in the published edition of RHash. Verification results of the two implementations are discussed to demonstrate the verification ability of VERIHASH.

The paper is organized as follows: Section II provides the necessary background on software formal verification; Section III reviews work related to our approach; Section IV provides the problem definition; Section V presents details of our hybrid methodology; and Section VI presents the verification results. We conclude the paper and suggest future work in Section VII.

II. BACKGROUND

A. SOFTWARE VERIFICATION

Software verification [9] is a branch of formal verification. Compared to software testing, which is a type of dynamic analysis, formal verification can be seen as a type of static analysis.

Formal verification proves the correctness of a given protocol, model, structure, or algorithm by using formal methods of mathematics; the correctness property is also described in a formal specification. The main advantage of a formal verification is to explore the entire target state space and to produce a more reliable correctness result. Formal verification is heavily used in the verification of hardware circuits, protocol correctness, and many other areas in which the verification target can be easily modeled and extracted to a state space that can be explored.

Software verification, sharing the advantages of formal verification, attempts to migrate formal verification methods to software programs, libraries, and other types of protocol implementations. The most popular way to achieve this migration is to extract a model from a software implementation, then use techniques in model checking [10] to explore the state space of the implementation model; the verification result is generated from the analysis of the implementation model. Programs usually have an extensive number of states so that it is not possible to explore the mathematical model completely; memory will be exhausted, and the verifier will not produce result in a reasonable time. This is known as the state-space explosion problem [11].

To perform a successfull software verification, many researchers have proposed multiple techniques to solve the state-space explosion problem, such as bounded model checking [12], counter-example guided abstraction and refinement [13], abstract interpretation [14], and predicate abstraction [15].

B. THE SHA-3 CRYPTOGRAPHIC HASH FUNCTION

SHA-3 is a subset of the cryptographic primitive family KECCACK. One of its goals is to replace its flawed predecessors MD5 [16] and SHA-1 [17], which have been demonstrated to be vulnerable to collision attacks [18], [19].

The main design of SHA-3 is its sponge construction procedure and five functions of permutation. The sponge construction procedure allows SHA-3 to transform message blocks into a subset of the state in the "'absorb" phase, then output permutated message blocks in the "squeeze" phase. The five functions of permutation of SHA-3 are the functions θ, ρ, π, χ, and ι; each function performs a transformation of the SHA-3 "state", which is a three dimensional array of bits.

As a type of cryptographic hash function, SHA-3 performs operations on vast arrays of bits, together with loops to handle the continuous transformations of SHA-3 states. The SHA-3 state is a 5×5 array of 64-bit words, which totals of 1600 bits. Implementations of SHA-3 usually define multiple arrays to represent SHA-3 states and message blocks, and perform sets of bit-wise operations, variable type conversions, and array operations on them.

As a result, when applying software verification to SHA-3 implementations, and similar cryptographic hash functions, one should focus more attention on the operations mentioned above. Before verification, the model design, the extension of array theory, and the compositional verification method should be tuned according to the characteristics of cryptographic hash functions, so that the implementation model can preserve sufficient semantics for the verification engine, the array analysis can be accurate and bit-wise sound, and the verification method can generate the verification result with a low cost in time and memory.

III. RELATED WORK

Srivastava *et al.* [20] presented a tool called VS3 that automatically verifies complex properties of programs and infers maximally weak preconditions and maximally strong post-conditions by leveraging the power of SMT solvers. They have used VS3 to automatically verify $\forall$ and $\exists$ properties of programs, extending verification properties to qualified ones. The tool can discover program invariants but needs user interaction to supply a set of appropriate predicates and invariant templates, and requires further manual preparation for the verification procedure, which reduces the usability of the tool.

Srivastava *et al.* [21] described a novel technique for software verification by introducing the synthesis of imperative programs. Their technique synthesizes a program that meets the input-output specification and uses only the given resources for analysis. The insight behind the approach is to interpret program synthesis as generalized program verification, which allows bringing verification tools and techniques to program synthesis. They developed a tool to demonstrate the feasibility of the proposed approach by synthesizing programs in three different domains: arithmetic, sorting, and dynamic programming.

Das *et al.* [22] presented a new algorithm for partial program verification that runs in polynomial time and space, the algorithm can check that a program satisfies a given temporal safety property. Their insight is that by accurately modeling only those branches in a program for which the property-related behavior differs along the arms of the branch, they can design an algorithm that is sufficiently accurate to verify the program with respect to the given property. This is very helpful in solving the state-space explosion problem and can be applied to large programs. They have used the algorithm to provide the first verification of temporal safety properties for a program of the size of GCC.

Vardi and Wolper [9] proposed an automata-theoretic approach to automatic program verification. They focus their work on concurrent programs with temporal logic properties, and they are able to extend the approach to handle extensions of standard temporal logic and the model-checking approach to treat probabilistic programs.

Shaikhli *et al.* [23] addressed a comparative analysis study of the finalist SHA-3 candidates regarding the complexity of security, design and structure, as well as performance and cost, to measure the robustness of the algorithms in this area. They investigated the security of lightweight designs for the future security of hash functions. Analysis results in this paper are a good reference in construction of verification properties.

From our perspective, the software verification of implementations of cryptographic hash functions has not yet been conducted by researchers. Compared with our own verification methods, the above techniques do not fit the needs of array analysis and scalability, although some ideas about verification procedures in the above techniques influenced the design of VERIHASH.

IV. PROBLEM DEFINITION

Consider the software verification S on implementation I of a cryptographic hash function protocol A, the problem is defined in a formal perspective.

Firstly, an implementation I can be seen as a set of assignment of variables in it: $\mathcal{V}$ is a finite non-empty set of variables in the implementation I; an implementation setting ρ, denoted as $\rho := \mathcal{V} \to \mathbb{V}$, is a finite non-empty set of possible assignment behaviors of the implementation I, where $\mathbb{V}$ is a finite non-empty set of all possible assignment of variable values.

Secondly, a cryptographic hash function protocol A can be seen as a finite non-empty set of properties formula: $p(v)$ is a predicate of v, describing the constraint of values of v; a property formula $\phi \in A$ is the collection of predicates in A, and is denoted as $\phi := p(v)|(\phi_1 \wedge \phi_2)|(\neg\phi)$.

Thirdly, a software verification S on protocol consistency can be seen as a projection from cryptographic hash function protocol A and cryptographic hash function implementation I to a consistency value $\mathbb{B}$, where $\mathbb{B}$ has the value of true or false.

Finally, a software verification is defined as $S \in A \to I \to \mathbb{B}$, it takes a verification property ϕ and an implementation setting ρ as input, and outputs the verification result

$S(\phi, \rho)$ under the following rules:

$$S(true, \rho) := true$$
$$S(false, \rho) := false$$
$$S(\neg\phi, \rho) := \neg S(\phi, \rho)$$
$$S(\phi_1 \wedge \phi_2), \rho) := S(\phi_1, \rho) \wedge S(\phi_2, \rho)$$

A property formula $\phi \in A$ for I is verified to be true, denoted as $I \vDash \phi$, holds if and only if:

$$\forall \rho \in I : S(\neg\phi, \rho) = false$$

A cryptographic hash function implementation I is verified to be consistent with cryptographic hash function protocol A, denoted as $I \vDash A$, holds if and only if:

$$\forall \phi \in A : I \vDash \phi$$

The overall verification procedure is defined as follows: given a cryptographic hash function implementation I, a cryptographic hash function protocol A, the verification procedure is to find sufficient enough formula ϕ in A, and choose time and space efficient verification method S, to prove $I \vDash A$ and $I \vDash M$. That is to say, given a cryptographic hash function implementation I with implementation setting ρ and a cryptographic hash function protocol A with verification property ϕ, the software verification S framework should get the verification result $S(\phi, \rho)$ in a relatively low time and memory cost.

V. PROPOSED APPROACH

We present VERIHASH, a software-verification based solution for checking the protocol consistencies of cryptographic hash function implementations. In VERIHASH, an intermediate program model ELTS is designed to represent the semantics of the cryptographic implementations; then, an extended array theory is developed to support analysis of operations on vast arrays and bits; finally, the compositional-verification framework is presented to solve the large-scale analysis problem.

The overall architecture of VERIHASH is shown in Figure 1.

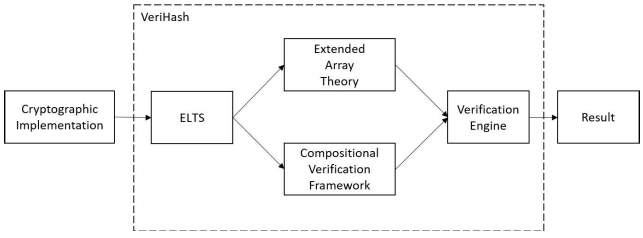


FIGURE 1. System Architecture of VERIHASH.

A. EXTENDED LABELED TRANSITION SYSTEMS

A traditional labeld transition system (LTS) [24] is designed to describe a system with states, labels, and transitions. As a benefit from its structure and automata semantics, the state space of an LTS is easily explored by software verification methods. However, an actual cryptographic hash function implementation I, in addition to states, labels, and location transitions, also contains variables, assignments, and operations; these make an LTS less expressive for modeling a software implementation with such elements. To enhance the expression ability of an LTS, we present an extended labeled transition system (ELTS); while preserving the advantage of being suitable for software verification, it is capable of modeling the semantics of an actual cryptographic hash function implementation.

1) EXTENSION COMPARED TO AN LTS

The ELTS extensions to an LTS are listed as follows:

- **support of program basic block:** transitions in ELTS are not only state changes, but also assignments of varaibles taking place in code blocks; thus, a state in ELTS is formed both by the location of a transition and values of variables;
- **guards for transitions:** a transition is allowed to take place in ELTS only when the guard expression is evaluated to true; guards of transitions help construct condition statements and loop semantics in ELTS;
- **built-in properties:** properties for ELTS are specified in first-order logic and are built-in to ELTS; this translates protocol consistency properties into ELTS reachability properties;
- **multiple erroneous states:** ELTS can have more than one erroneous state; this extends the ability of ELTS to express properties and allows ELTS to describe multiple reachability properties at once;
- **compostion rules:** ELTS is designed for source code verification and is highly capable to analyze function invocations, sequential code blocks, conditional statements, and loop semantics by defining ELTS composition rules. These rule will be dicussed later.

2) ELTS DEFINITION

Formally, an ELTS system S is a ten tuple $(L, l_0, V, \mathbb{V}_0, C, c_0, B, T, P, E)$ where:

- L is a finite non-empty set of locations
- l_0 is the initial location af a code block
- V is a finite non-empty set of variables in an implementation I
- $\mathbb{V}$ is a possibly infinite non-empty set of values of V
- C is a finite non-empty set of basic blocks in I
- c_0 is the first basic block, which initializes values of V
- B is a finite non-empty set of guards of transitions; a guard is constructed by variables from the program
- $T \subseteq L \times \mathbb{V} \times C \times B \times \mathbb{V} \times L$ is a finite non-empty set of transitions; a transition $t = (l_i, \mathbb{V}_i, c_i, b_i, \mathbb{V}_j, l_j) \in T$ can also be denoted as $(l_i, \mathbb{V}_i) \xrightarrow{c_i, b_i} (l_j, \mathbb{V}_j)$.
- P is a finite non-empty set of properties, a property p is a predicate of V
- E is a finite non-empty set of erroneous locations

3) ELTS MODEL EXTRACTION

Consider a cryptographic hash function implementation I of an cryptographic hash function protocol A. The ELTS model extraction is performed as follows:

- **code block extraction:** in the implementation I, let $C = \{c_0, c_1, c_2, ...\}$ be the finite non-empty set of code blocks, and let $L = \{l_0, l_1, l_2, ...\}$ be the finite non-empty set of locations. The initial code block is denoted $c \in C$, and the beginning location of the nth basic block c_n is denoted l_n.
- **control flow construction:** assignments in implementation I can be divided into code blocks in C according to program location, so I can be seen as set of $c \in C$, and a basic block can be seen as a set of assignments $\rho \in c$.
- **property modeling:** protocol consistency assertions $\phi \in A$ in source code are specified based on locations $l \in L$, and modeled as reachability properties $p \in P$ in ELTS. Let E be the set of erroneous locations in ELTS, a reachable erroneous location can be seen as a protocol consistency violation.

B. EXTENDED ARRAY THEORY

The extended array theory is designed to support analysis of operations on vast arrays and bits. In this section, read-write array theory is proposed to model basic array operations, set-copy array theory is proposed to model batch operations on a set of array elements, and the memset-memcpy theory is given to model manipulation of memory; this is a natural extension of previous array theories, because memory can be seen as a linear array of bits.

1) READ-WRITE ARRAY THEORY

The read-write array theory $T_{read-write}$ is shown in Table 1 to demonstrate the basic idea of array manipulation. It is a summary of [25]; the read operation returns the value of an element *ELE* at the given index *IDX* in array *ARR*, and the write operation updates the value of the element *ELE* at the given index *IDX* in array *ARR*. Each operation is performed on only one element of the array and can be used to model basic array semantics in C.

TABLE 1. Read-write array theory.

Sorts	ELE: elements IDX: indices ARR: arrays
Functions	read: ARR × IDX → ELE write: ARR × IDX × ELE → ARR

By definition of array, the semantics of read and write should also follow these read-over-write axioms:

$$p = r \Rightarrow read(write(a, p, v), r) = v$$
$$\neg(p = r) \Rightarrow read(write(a, p, v), r) = read(a, r)$$

The first axiom ensures that after updating the element at index p with the value v via a write operation, the read operation on the same element at index p will return the value v. The second axiom ensures that updating the element at index p with value v via a write operation will not affect the values of elements at other indices. These axioms state that storing the value v into an array a at index p and subsequently reading the value at index q results in the value v if the indices p and q are identical. Otherwise, the write operation does not influence the result of the read operation.

Read-write array theory can model only operations on a single element; it can not model multiple operations on a set of array elements, and a loop invocation of read-write operations will add verification overhead to the underlying verification engine.

2) SET-COPY ARRAY THEORY

Operations on a trunk of array elements are usually seen as set and copy semantics. A set operation updates a set of continuous array elements with the same value, which can be seen as a set of write operations on these elements; a copy operation has two steps: first, it reads the values from a source set of continuous array elements, and then, it updates the destination set of array elements. The set-copy array theory is proposed to model set and copy operations, which is very important for verification of cryptographic hash function implementations.

The set-copy array theory $T_{set-copy}$ extends $T_{read-write}$ with definitions shown in Table 2; this theory is the extension of [26]. In addition to parameters *ELE*, *IDX*, and *ARR*, the set operation in $T_{set-copy}$ introduces the *SIZE* parameter to indicate the total number of elements to be updated, and *IDX* is the beginning location of the set operation; the copy operation shares the same type of parameters as the set operation, but requires two *IDX* parameters to indicate the beginning and ending indices of the elements. The beginning index should not be higher than the ending index.

TABLE 2. Set-copy array theory.

Sorts	ELE: elements IDX: indices ARR: arrays SIZE = IDX
Functions	set: ARR × IDX × ELE × SIZE → ARR copy: ARR × IDX × ARR × IDX × SIZE → ARR

By definition of array set operation, the semantics of the set operation is given by the following read-over-set axioms:

$$p \leq r < p + s \Rightarrow read(set(a, p, v, s), r) = v$$
$$\neg(p \leq r < p + s) \Rightarrow read(set(a, p, v, s), r) = read(a, r)$$

The first axiom means that after the set operation to write a number s of elements starting from index p with a value v, the read operation on any element between the index p and index p + s should return value v; the second axiom means that the value of any element outside the set operation range should remain unchanged.

By definition of array copy operation, the semantics of the copy operation is given by the following read-over-copy axioms:

$$\begin{aligned} p \le r < p+s &\Rightarrow read(copy(a,p,b,q,s),r) \\ &= read(a, q+(r-p)) \\ \neg(p \le r < p+s) &\Rightarrow read(copy(a,p,b,q,s),r) \\ &= read(a,r) \end{aligned}$$

The first axiom means that after the operation of copying values of elements in range p to p + s, to elements in range q to q + s, the read operation in range p to p + s will return the same value as the read operation on corresponding elements in range q to q + s; the second axiom requires that the copy operation not affect elements out side the given range.

3) MEMSET-MEMCPY THEORY

The set-copy array theory $T_{set-copy}$ can be extended to modeling of memory operations [26] , such as the memset and memcpy functions in the C standard library. In addition to the corresponding array theory extension, the modeling of memset and memcpy requires a precise memory model of the verification engine; VERIHASH currently supports basic memory semantics such as memory allocation, memory free, memset and memcpy. Support for pointer analysis will be designed in the future.

With the extended array theory, analysis of vast operatioins on arrays and bits is conducted in the ELTS model, thus helps ensuring protocol consistency of the implementation. Array operations such as read, write, memset, and memcpy can be modeled in ELTS and verified by the back-end verification engine.

C. COMPOSITIONAL VERIFICATION

Compositional verification is based on ELTS semantics and is the key to supporting large-scale analysis. In VERIHASH, compositional verification is proposed as three steps. First, the compoistion definition is conducted on ELTS to define the behavior of ELTS compositions. Second, the structural composition rules are given to describe how to divide and composite different components of the implementation. Third, the composition framework demonstrates how an implementation is verified compositionally.

1) COMPOSITION DEFINITION

Composition of ELTS models is defined as follows.

We have two ELTS models $S = (L, l_0, V, \mathbb{V}_0, C, c_0, B, T, P, E)$ and $S' = (L', l_0', V', \mathbb{V}_0', C', c_0', B', T', P', E')$, the compositions of two ELTS models are the compositions of their variables, states, transitions, and code blocks. We say that S can transform to S' with the transition of code block c and guard b, denoted as $S \xrightarrow{c,b} S'$, if and only if $(l_0, \mathbb{V}_0, c, b, l_0', \mathbb{V}_0') \in \delta$ $L = L'$, $V = V'$, and $T = T'$. The ELTS composition operator $\|$ is defined between two ELTS system; it is commutative and associative on both sides, and comprises variables, states, transitions, and code blocks of the two ELTS systems.

There are two types of composition operators, the parallel composition operator $\|_{PAR}$ and the sequential composition operator $\|_{SEQ}$. Formally, let $S_1 = (L^1, l_0^1, V^1, \mathbb{V}_0^1, C^1, c_0^1, B^1, T^1, P^1, E^1)$ and $S_2 = (L^2, l_0^2, V^2, \mathbb{V}_0^2, C^2, c_0^2, B^2, T^2, P^2, E^2)$ be two ELTS systems; the general composition $S_1 \| S_2$ is defined according to the different semantics of the operators.

A parallel composition $S_1 \|_{PAR} S_2$ forms a new ELTS system $S = (L, l_0, V, \mathbb{V}_0, C, c_0, B, T, P, E)$ in which the location set is defined as $L = L^1 \times L^2$, $l_0 = (l_0^1, l_0^2)$, the variable set is defined as $V = V^1 \times V^2$, initial values of the variable set are defined as $\mathbb{V}_0 = (\mathbb{V}_0^1, \mathbb{V}_0^2)$, the code block set is defined as $C = C^1 \times C^2$, the initial code block is defined as $c_0 = (c_0^1, c_0^2)$, the guard condition set is defined as $B = B^1 \times B^2$, the protocol consistency property set is defined as $P = P^1 \times P^2$, and the erroneous location set is defined as $E = E^1 \times E^2$. The transition set T in the new ELTS S is defined as follows:

$$\frac{S_1 \xrightarrow{c,b} S_1', c \notin C^2, b \notin B^2}{S_1 \|_{PAR} S_2 \xrightarrow{c,b} S_1' \|_{PAR} S_2}$$

$$\frac{S_1 \xrightarrow{c,b} S_1', S_2 \xrightarrow{c,b} S_2', c \notin C^2}{S_1 \|_{PAR} S_2 \xrightarrow{c,b} S_1' \|_{PAR} S_2'}$$

A sequential composition $S_1 \|_{SEQ} S_2$ forms a new ELTS system $S = (L, l_0, V, \mathbb{V}_0, C, c_0, B, T, P, E)$ in which the location set is defined as $L = L^1 \cup L^2 \cup \{l_S EQ\}$, $l_0 = l_0^1$, the variable set is defined as $V = V^1 \cup V^2$, the initial values of the variable set are defined as $\mathbb{V}_0 = \mathbb{V}_0^1$, the code block set is defined as $C = C^1 \cup C^2 \cup \{c_S EQ\}$, the initial code block is defined as $c_0 = c_0^1$, the guard condition set is defined as $B = B^1 \cup B^2 \cup \{b_S EQ\}$, the protocol consistency property set is defined as $P = P^1 \cup P^2$, and the erroneous location set is defined as $E = E^1 \cup E^2$. The transition set T in the new ELTS S is defined as follows:

$$\frac{S_1 \xrightarrow{c,b} S_1', c \in C^1, b \in B^1}{S_1 \|_{SEQ} S_2 \xrightarrow{c,b} S_1' \|_{SEQ} S_2}$$

$$\frac{S_2 \xrightarrow{c,b} S_2', c \in C^2, b \in B^2}{S_1 \|_{SEQ} S_2 \xrightarrow{c,b} S_1 \|_{SEQ} S_2'}$$

2) COMPOSITION RULES

With the definition of composition of ELTS systems, the composition rules can be constructed on ELTS compositions to model the composition and decomposition of different implementation structure.

Composition rules are defined as follows:

assignment axiom

$$\{P[x := e]\}x := e\{P\}$$

if-then-else rule

$$\frac{\{P \wedge b\}C_1\{Q\}, \{R \wedge \neg b\}C_2\{Q\}}{\{P\}\text{if } b \text{ then } C_1 \text{ else } C_2 \text{ fi}\{Q\}}$$

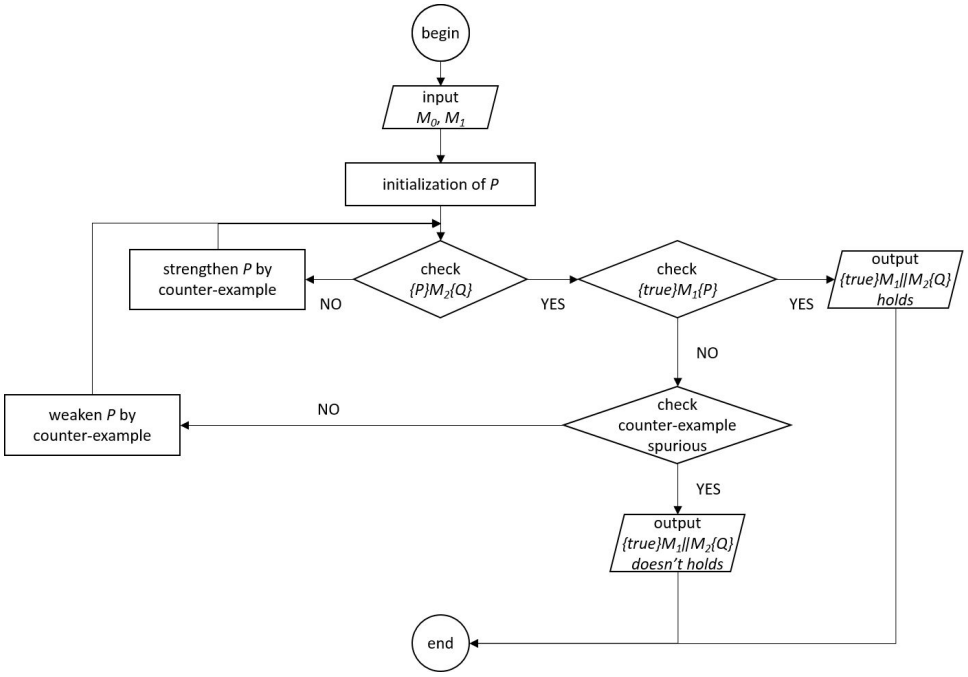


FIGURE 2. Compositional verification framework of VERIHASH.

while rule

$$\frac{\{P \wedge b\}C\{P\}}{\{P\}\text{while } b \text{ do } C \text{ od}\{P \wedge \neg b\}}$$

consequence rule

$$\frac{(P \Rightarrow P^{'}), \{P^{'}\}C\{Q^{'}\}, (Q^{'} \Rightarrow Q)}{\{P\}C\{Q\}}$$

Same as [27], the composition rules are defined under a Hoare logic alike assumption:

- A composition formula is a triple $\{P\}S\{Q\}$, where S is an ELTS code block, Q is a post-guard-condition of the code block, and P is a pre-guard-condition of the code block.
- The composition formula indicates that if the ELTS code block S satisfies P, then the system must also guarantee Q.
- An ELTS *ELTS* can be seen as an assumption or as a property itself. To be used as a pre-guard-condition for code block S, *ELTS* itself is composited with S, which can be seen as an abstraction of the ELTS code block S. To be used as a post-guard-condition, *ELTS* is turned into a post-conditionand then composited with ELTS code block S.
- To verify an assume-guarantee composition formula $\{P\}S\{Q\}$, where both P and Q are guard conditions of ELTS, a new ELTS $P \parallel S \parallel Q_err$ is constructed to verify if erroneous states are reachable.

3) COMPOSITION FRAMEWORK

Given an implementation I that is composited by two ELTS systems, S_1 and S_2, S_1 is the predecessor of S_2 structurally, I is seen as the composition of S_1 and S_2, denoted as $\{true\}S_1 \parallel S_2\{Q\}$. As mentioned above, if both composition formulas $\{true\}S_1\{P\}$ and $\{P\}S_2\{Q\}$ hold, then the composition of $\{true\}S_1 \parallel S_2\{Q\}$ holds, which means that the implementation I satisfies the protocol consistency property Q.

The key of this composition framework is to obtain an appropriate pre-guard-condition P, that can bridge the divided ELTS systems. To construct an appropriate pre-guard-condition P, a procedure of continuous composition is shown in Figure 2.

The composition procedure is designed as follows: (i) at the beginning of each iteration point i, a pre-guard condition P_i is constructed based on the structure composition rules and the interaction results of previous computation; (ii) the verification engine verifies the composition formula $\{P_i\}S_2\{Q\}$ to see whether it is satisfiable; (iii) if the verification result is proved to be satisfiable, it means that the current pre-guard-condition P_i is too weak and should be strengthened; the next pre-guard-condition P_{i+1} is computed, and information from the current verification result will be reused to help construct a stronger composition formula; (iv) if the verification result is proved to be unsatisfiable, the current pre-guard-condition P_i is strong enough to verify the protocol consistency property; the previous composition formula $\{true\}S_1\{P_i\}$ will be verified to check satisfiability, and the iteration continues.

VI. EXPERIMENT RESULTS

The proposed verification framework VERIHASH is implemented on the basis of popular open source software verifier CPAchecker. We evaluate our verification framework on two SHA-3 cryptographic hash function implementations, the

TABLE 3. Verification results of *Keccack*.

Verification results of *RHash*					
Functions	Array Bounds	Pointer Safety	Memory Leak	Protocol Consistency	time (s)
keccak_chi	PASSED	PASSED	PASSED	PASSED	17.68
keccak_pi	PASSED	PASSED	PASSED	PASSED	22.65
keccak_theta	PASSED	PASSED	PASSED	PASSED	10.26
rhash_keccak_final	PASSED	PASSED	PASSED	PASSED	1.98
rhash_keccak_init	PASSED	PASSED	PASSED	PASSED	15.44
rhash_sha3_final	PASSED	PASSED	PASSED	PASSED	87.31
rhash_sha3_permutation	PASSED	PASSED	PASSED	PASSED	160.05
rhash_sha3_process_block	ERROR	PASSED	PASSED	ERROR	198.10

winner of NIST SHA-3 competition Keccack and an open source project RHash. We successfully verified the core parts of SHA-3 computation, and reproduced a bug in the published edition of RHash, verification results of the two implementations are discussed to demonstrate the verification ability of VERIHASH.

A. SETUP

The experiments are performed on a machine with a 3.4 GHz 64-bit 8 Core CPU and 32GB of memory, running an Ubuntu 16.04. We set a 15GB memory limit and a 900s timeout for the analysis of each benchmark. To help measure the time usage accurately, we employ benchexec [28] as the benchmarking toolset.

The assurance of protocol consistency is another very important aspect of program verification. The usual practice of protocol consistency verification is inserting certain number of assertions in the source code, and let the verification tool perform an assertion checking. While almost all modern program verification tools support assertion verification, a successful assertion verification requires a deep understanding of the program and sophisticated design of source code insertion.

B. PREPROCESS OF SOURCE CODE

Both *Keccack* and *RHash* are structured by Makefile, which is common in open source implementations. The tool we choose to analysis implementations' source code is CIL from UC Berkeley, it supports Makefile structured project parsing and source code merging.[1]

During preprocessing of the source code, the source code trimming phase is performed manually. This is because source code trimming requires domain specific knowledge of the SHA-3 cryptographic hash function protocol.

For *Keccack*, it has multiple implementations for different platforms, it even has optimized assembly code which is not yet supported by our approach, it also has test cases and implementations of tools for different purpose, so trimming of *Keccack*'s source code requires clean removal of the elements discussed above.

For *RHash*, as it's a hashing tool supporting multiple algorithms and SHA-3 is one of its algorithm implementations, so the source code trimming goal is to remove unrelated algorithms.

[1] cilly -possible -option -we -used project/

C. DISCUSSION

The core of SHA-3 protocol defines a permutation R on an array with five functions, where $R = \iota \circ \chi \circ \pi \circ \rho \circ \omega$. n_r rounds permutation of R constructs KECCACK-f[b], the KECCACK function can be then constructed as KECCACK[c] when taking KECCACK-f[1600] as underlying function. By choosing the different value of c and padding rule, the whole family of SHA-3 functions can be generated.

The protocol consistency verification can be performed according to the construction process as below: in permutation phase, VERIHASH verified five permutation functions θ, ρ, π, χ, and ι; in message construction phase, VERIHASH verified the sponge function; in wrapper function phase, VERIHASH verified the wrapping functions of SHA3-224, SHA3-256, SHA3-384, and SHA3-512.

As shown in Table 3, we formally verified all permutation functions and wrapper functions of SHA-3 implementations, and a bug is reproduced during the verification of *RHash*: there are unsafe operations in the function rhash_sha3_process_block at line 234, the continuing access to the array *block* will make the visiting pointer out of array bounds, which then makes the final computation of SHA-3 state *hash* incorrect. This bug was submitted to *RHash* website and was confirmed by the developer, in the following releases of *RHash*, this bug was fixed by a guarded array access condition statement.

VII. CONCLUSION

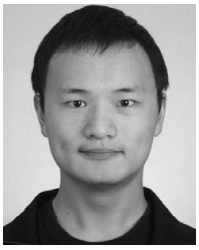
We have proposed a fully automatic software verification framework VERIHASH that applies software verification to protocol consistency proof of cryptographic hash function implementations, solves challenges of semantic modeling, analysis of operations on vast arrays and bits, and supports large-scale analysis via compositional verification. We formally verified two SHA-3 implementations, *Keccack* and *RHash*. To the best of our knowledge, this is the first formal verification of SHA-3 implementations. We successfully verified the core parts of the two implementations and reproduced a bug in the published edition of RHash.

There are several directions for possible future research. (i) Here, we focused on the verification of C source code; formal verification on assembly code is more challenging

and may expose new flaws in implementations; (ii) We verified the source code only in the x86_64 Linux environment; verification on other platforms such as Windows and embedded systems can provide more meaningful proof of SHA-3 programs. (iii) There are still additional properties that play important roles in cryptographic algorithms and should be verified: e.g., the verification of consistency against side-channel attacks requires a verification strategy well-designed in time, memory and hardware.

REFERENCES

[1] Y.-F. Chen *et al.*, "Verifying curve25519 software," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2014, pp. 299–309.
[2] Y. Chevalier and L. Vigneron, "A tool for lazy verification of security protocols," in *Proc. 16th Annu. Int. Conf. Autom. Softw. Eng. (ASE)*, Nov. 2001, pp. 373–376.
[3] Y. Jiang *et al.*, "Design and optimization of multiclocked embedded systems using formal techniques," *IEEE Trans. Ind. Electron.*, vol. 62, no. 2, pp. 1270–1278, Feb. 2015.
[4] Y. Jiang *et al.*, "Design of mixed synchronous/asynchronous systems with multiple clocks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 8, pp. 2220–2232, Aug. 2015.
[5] Y. Jiang, H. Song, R. Wang, M. Gu, J. Sun, and L. Sha, "Data-centered runtime verification of wireless medical cyber-physical system," *IEEE Trans. Ind. Informat.*, vol. 43, no. 3, pp. 949–957, Mar. 2017.
[6] Y. Jiang *et al.*, "Bayesian-network-based reliability analysis of PLC systems," *IEEE Trans. Ind. Electron.*, vol. 60, no. 11, pp. 5325–5336, Nov. 2013.
[7] Y. Jiang *et al.*, "Use runtime verification to improve the quality of medical care practice," in *Proc. 38th Int. Conf. Softw. Eng. Companion*, 2016, pp. 112–121.
[8] H. Zhang, Y. Jiang, W. N. N. Hung, X. Song, M. Gu, and J. Sun, "Symbolic analysis of programmable logic controllers," *IEEE Trans. Comput.*, vol. 63, no. 10, pp. 2563–2575, Oct. 2014.
[9] M. Y. Vardi and P. Wolper, "An automata-theoretic approach to automatic program verification," in *Proc. 1st Symp. Logic Comput. Sci. (LICS)*, 1986, pp. 322–331.
[10] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*. Cambridge, MA, USA: MIT Press, 1999.
[11] A. Valmari, "The state explosion problem," in *Lectures on Petri Nets I: Basic Models*. Berlin, Germany: Springer, 1998, pp. 429–528.
[12] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Adv. Comput.*, vol. 58, no. 5, pp. 117–148, 2003.
[13] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *J. ACM*, vol. 50, no. 5, pp. 752–794, 2003.
[14] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proc. 4th ACM SIGACT-SIGPLAN Symp. Principles Programm. Lang.*, 1977, pp. 238–252.
[15] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani, "Automatic predicate abstraction of C programs," *ACM SIGPLAN Notices*, vol. 36, no. 5, pp. 203–213, 2001.
[16] R. Rivest, *The MD5 Message-Digest Algorithm*, RFC 1321, 1992.
[17] V. Rijmen and E. Oswald, "Update on SHA-1," in *Proc. Cryptogr. Track RSA Conf.*, 2005, pp. 58–71.
[18] M. Stevens, "Fast collision attack on MD5," *IACR Cryptol. ePrint Archive*, vol. 53, no. 9, p. 104, 2006.
[19] X. Wang, Y. L. Yin, and H. Yu, "Finding collisions in the full SHA-1," in *Proc. Annu. Int. Cryptol. Conf.*, 2005, pp. 17–36.
[20] S. Srivastava, S. Gulwani, and J. S. Foster, "VS^3: SMT solvers for program verification," in *Computer Aided Verification*. Grenoble, France: Springer, 2009, pp. 702–708.
[21] S. Srivastava, S. Gulwani, and J. S. Foster, "From program verification to program synthesis," *ACM SIGPLAN Notices*, vol. 45, no. 1, pp. 313–326, 2010.
[22] M. Das, S. Lerner, and M. Seigle, "ESP: Path-sensitive program verification in polynomial time," *ACM SIGPLAN Notices*, vol. 37, no. 5, pp. 57–68, 2002.
[23] A. Shaikhli, I. Fakhri, M. A. Alahmad, and K. Munthir, "Hash function of finalist SHA-3: Analysis study," *Int. J. Adv. Comput. Sci. Inf. Technol.*, vol. 2, no. 2, pp. 1–12, 2014.
[24] J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu, "Learning assumptions for compositional verification," in *Proc. 9th Int. Conf. Tools Algorithms Construction Anal. Syst.*, Berlin, Germany, 2003, pp. 331–346. [Online]. Available: http://dl.acm.org/citation.cfm?id=1765871.1765903
[25] J. McCarthy, "Towards a mathematical science of computation," in *Program Verification*. Dordrecht, The Netherlands: Springer, 1993, pp. 35–56.
[26] S. Falke, F. Merz, and C. Sinz, "Extending the theory of arrays: Memset, memcpy, and beyond," in *Proc. Fifth Working Conf. Verified Software: Theories, Tools, Experiments*. Atherton, CA, USA, 2013, pp. 108–128.
[27] A. Pnueli, "In transition from global to modular temporal reasoning about programs," in *Proc. Logics Models Concurrent Syst.*, 1985.
[28] D. Beyer, "Reliable and reproducible competition results with benchexec and witnesses (report on sv-comp 2016)," in *Proc. TACAS*, 2016, pp. 887–904.



DEXI WANG received the B.S. degree in software engineering from Tongji University, Shanghai, China, in 2012. He is currently pursuing the Ph.D. degree in software engineering at Tsinghua University, Beijing, China. His current research interests include software verification, model checking, and their applications in software implementations.

YU JIANG received the B.S. degree in software engineering from the Beijing University of Posts and Telecommunications, Beijing, China, in 2010, and the Ph.D. degree in computer science from Tsinghua University, Beijing, in 2015. Since 2016, he has been an Assistant Professor at Tsinghua University. His research interests include domain-specific modeling, formal verification, and their applications in embedded systems.

HOUBING SONG received the M.S. degree in civil engineering from The University of Texas at El paso, El Paso, TX, USA, in 2006, and the Ph.D. degree in electrical engineering from the University of Virginia, Charlottesville, VA, USA, in 2012. Since 2012, he has been an Assistant Professor at the West Virginia University Institute of Technology, Montgomery, WV, USA. His research interests include cyber-physical systems, signal processing for communications and networking, and cloud computing/edge computing.

FEI HE received the B.S. degree in computer science from the National University of Defense Technology, Changsha, China, in 2002, and the Ph.D. degree in computer science from Tsinghua University, Beijing, China, in 2008. Since 2011, he has been an Assistant Professor at Tsinghua University. His research interests include formal verification, model checking, and their applications in embedded systems.

MING GU received the B.S. degree in computer science from the National University of Defense Technology, Changsha, China, in 1984, and the M.S. degree in computer science from the Chinese Academy of Science, Shengyang, China, in 1986. Since 1993, she has been a Professor at Tsinghua University. Her research interests include formal methods, middleware technology, and distributed applications.

JIAGUANG SUN received the B.S. degree in automation science from Tsinghua University in 1970. He is currently a Professor at Tsinghua University. He is dedicated to teaching, and research and development activities in computer graphics, computer-aided design, formal verification of software, and system architecture.