Automatic Equivalence Checking for Assembly Implementations of Cryptography Libraries

Jay P. Lim
Department of Computer Science
Rutgers University
Piscataway, USA
jpl169@cs.rutgers.edu

Santosh Nagarakatte
Department of Computer Science
Rutgers University
Piscataway, USA
santosh.nagarakatte@cs.rutgers.edu

Abstract—This paper presents an approach and a tool, CASM-VERIFY, to automatically check the equivalence of highly optimized assembly implementations of cryptographic algorithms. The key idea of this paper is to decompose the equivalence checking problem into several small sub-problems using a combination of concrete and symbolic evaluation. Given a reference and an optimized implementation, CASM-VERIFY concretely executes the two implementations on randomly generated inputs and identifies likely equivalent variables. Subsequently, it uses symbolic verification using an SMT solver to determine whether the identified variables are indeed equivalent. Further, it decomposes the original query into small sub-queries using a collection of optimizations for memory accesses. These techniques enable CASM-VERIFY to verify the equivalence of assembly implementations (e.g., x86 and SSE) of various algorithms such as SHA-256, ChaCha20, and AES-128 for a message block.

Index Terms—Formal verification, Cryptography

I. INTRODUCTION

Mainstream libraries for cryptography (e.g., OpenSSL and BoringSSL) implement Transport Layer Security (TLS) and Secure Socket Layer (SSL) protocols for secure communication. These protocols use a wide range of cryptographic algorithms such as symmetric key ciphers, public-key ciphers, and hash functions. They are highly optimized given that these components are performance critical. These libraries have several thousand lines of manually optimized assembly code for high performance. Further, the implementations of these algorithms utilize a wide range of optimizations: (1) heavy unrolling of loops to avoid branch penalty, (2) carefully crafted use of vector instructions, (3) use of instructions to avoid side-channels, and (4) optimizations to ensure constant-time execution. The end-result is that the implementation looks drastically different from the specification.

Although these systems undergo intensive testing, bugs are pretty common as testing does not guarantee the absence of errors for all inputs. Recently, OSS-Fuzz [1] and numerous other projects have found bugs in implementations of OpenSSL. For example, OSS-Fuzz found a carry propagation bug [2] in the Curve25519 implementation of OpenSSL in May 2017. Surprisingly, this bug was present since OpenSSL 1.0.2.

As these libraries are widely used, checking the correctness of the optimized code with respect to a reference standard is important. A promising method to attain this goal is to implement verified cryptographic algorithms using programming languages or program logics developed with program verification in consideration [3], [4], [5], [6], [7]. Recent projects have successfully implemented correct TLS protocols [8], [9] and ciphers [10], [11], [12]. Correct by construction approach is an ideal approach to implement new algorithms. However, there is a huge corpus of existing implementations that have been hand-optimized for various architectures. The SAW [13] and Axe [14] projects verify the correctness of cryptographic algorithms written in high-level languages such as Java or C. Unfortunately, there is a disconnect between the high-level language implementations and the hand-optimized assembly used by mainstream libraries.

Alternatively, checking the equivalence of general-purpose programs is a well-studied problem. A common approach for checking the equivalence of two programs with different loop structure is to unroll the loops and identify straight-line program segments that are equivalent by observing branch conditions [15] or by observing program states during concrete executions [16]. Subsequently, they generate symbolic expressions for the straight-line fragments and solve the constraints between them using a Satisfiability Modulo Theory (SMT) solver. These techniques are ineffective in the context of assembly implementations of cryptographic algorithms because the implementation can be thousands of instructions long. Automated verification with an SMT solver would not terminate as the resulting symbolic expressions are too complex. One way to address this challenge is to manually annotate intermediate variables that should be equivalent between the optimized implementation and the reference implementation and prove the equivalence of the implementations compositionally. However, requiring the users to annotate possibly equivalent variables is likely infeasible in our context, because there can be thousands of such variables. In the context of hardware verification, SAT sweeping was used to identify equivalent nodes in circuit boards and to optimize equivalent nodes [17]. We adapt this idea to the context of cryptographic algorithms with memory operations.

This paper presents a set of techniques and a tool, CASM-VERIFY, to automatically verify the two assembly implemen-

1Artifact available at: https://doi.org/10.5281/zenodo.2229779
ments or an assembly and a reference implementation of cryptographic algorithms. Our key idea is to leverage a combination of concrete execution and symbolic evaluation coupled with query decomposition optimizations to reduce the large verification condition into smaller sub-problems. Given two implementations, CASM-VERIFY constructs directed acyclic graphs (DAG) for them and identifies likely equivalent variables by concretely evaluating the DAGs with random inputs. Subsequently, it checks whether these likely variables are indeed equivalent by constructing symbolic expressions in terms of the leaf nodes. When two nodes are indeed equivalent, it merges the nodes in the DAG. We propose a sound, yet fast equivalence checking method when the two nodes have common descendants (see Section III-D), which we call quick check optimization. We propose novel optimizations of memory read operations by leveraging equivalence information from the DAGs and by converting memory accesses to nested if-then-else nodes (see Section V).

Our prototype, CASM-VERIFY, is open source and is publicly available [18]. We have used CASM-VERIFY to verify various assembly implementations of SHA-256, ChaCha20, and AES-128 from OpenSSL are equivalent to the reference implementation that we developed based on the official specification. CASM-VERIFY can also check the equivalence of two different assembly implementations (e.g., x86 vs SSE). We have also discovered a possible ambiguity in the specification of ChaCha20. Our experiments show that CASM-VERIFY is effective in detecting incorrect incremental changes to assembly implementations of cryptographic algorithms.

II. HIGH LEVEL SKETCH OF OUR APPROACH

Our goal is to automatically show the equivalence of two implementations of cryptographic algorithms. Our approach is tailored to cryptographic algorithms that use loops with static loop counts, subtle bit-manipulation operations, and look-up tables. We support two scenarios: (1) the user provides a reference implementation in our tool’s domain specific language and an optimized implementation and (2) the user provides two distinct x86 implementations of a cryptographic algorithm. In both scenarios, the user also provides a precondition and a postcondition that relate the input variables and the output variables, respectively. The precondition specifies equivalent input variables in the two implementations. The postcondition identifies the output variables that must be equivalent for the two implementations to be equivalent.

Challenges with existing equivalence checking techniques. Although automated equivalence checking is a widely studied area [19], [15], [20], [16], [21], [22], [23], [24], [25], [26], [27], [28], the problem is challenging in the context of cryptographic algorithms for the following reasons: (1) each implementation has thousands of hand-optimized instructions, (2) different ciphers use memory to store lookup tables and keys, and (3) existing code does not provide information about equivalent intermediate variables. Hence, existing approaches that encode the equivalence as constraints in first-order logic and use SMT solvers do not terminate (Section V).

Modular decomposition with our approach. Our approach also relies on SMT solvers to reason about the equivalence of two implementations. We create directed acyclic graph (DAG) representations of the entire implementations. When the program has loops, we unroll them because the loop trip-counts are statically known especially with cryptographic algorithms. Essentially, our problem is to show the equivalence of two sets of large DAGs (i.e., one for the reference and the other for the optimized implementation).

Our key idea is to decompose the large formula that encodes the equivalence of two implementations into smaller sub-formulae and check the equivalence of these two implementations using these sub-formulae. To decompose the formula, we need to identify intermediate equivalent nodes. We address the challenge of identifying intermediate equivalent variables for modular decomposition by using a combination of concrete execution and a subsequent symbolic verification. We use concrete execution to identify likely equivalent variables inspired by prior approaches to invariant generation [29], [16]. It prunes the space of variables that we need to explore. Verification condition generation using symbolic evaluation checks if these likely equivalent variables are indeed equivalent, providing soundness.

Illustration. We illustrate our approach with a simple pedantic example in Figure 1 where we want to check the equivalence of two implementations in Figure 1(a) and Figure 1(b), which we call P1 and P2, respectively. The assembly implementation in Figure 1(b) optimizes the computation of var1 (line S1 in Figure 1(a)) by computing \((a \gg 4) \oplus a\) \(\gg 9\) (lines I1-I4), and var2 (line S2) by computing \(((b \oplus c) \& a) \oplus c\) (lines I5-I7), where \(\gg\) is the rotate right operation and \(\oplus\) is the exclusive-or operation.

Our tool, CASM-VERIFY, creates a DAG for both P1 and P2, which is shown in Figure 2. Our tool handles flag registers and instructions operating on operands with different bit-widths. To ease exposition, we restrict ourselves to the pedantic example in Figure 2.

Equivalence checking. The goal of equivalence checking is to verify that the output nodes of P1 and P2 are equivalent. A common method for equivalence checking is to generate
Fig. 2. DAG representations of the two implementations. A leaf node is either an input variable or a concrete constant. All other nodes represent intermediate variables used to calculate the output. If the program has multiple output variables, then the DAGs can have multiple root nodes. The intermediate nodes represent an operation on the child nodes. For example, \( P_1 \) represents the output variable of \( T_1 \) (i.e., \( \text{result} \) in Figure 1(a)) and the node \( O_2 \) represents the output variable of \( P_2 \) (i.e., \( \text{ebx} \) in Figure 1(b)). All intermediate nodes represent an operation on the child nodes. For example, \( T_6 \) is computed as \( T_1 \oplus T_2 \), where \( \oplus \) is exclusive-or operation.

The verification conditions by symbolic evaluation of the DAGs in a particular first-order theory. The verification conditions encode the root nodes based on the leaf nodes. The precondition for our example is:

\[
((a = \text{eax}) \land (b = \text{ebx}) \land (c = \text{ecx}))
\]

(Pre)

The verification conditions for \( O_1 \) and \( O_2 \) are:

\[
((a \gg 13) \oplus (a \gg 9)) + ((a \& b) \oplus (\neg a \land c))
\]

(EQ1)

\[
(((\text{eax} \gg 4) \oplus \text{eax}) \gg 9) + (((\text{ebx} \lor \text{ecx}) \land \text{eax}) \lor \text{ecx})
\]

(EQ2)

The next step is to create a verification condition to check the equivalence of two DAGs.

\[
\forall_{a,b,c,eax,ebx,ecx} \text{Pre} \implies (\text{EQ1} \implies \text{EQ2})
\]

Typically SMT solvers are used to check the validity of the above formula. If the negation of the above formula is \text{unsat}, then the original formula is valid for all inputs. However, in the context of our domain, SMT solvers do not terminate with an answer or run out of memory because each DAG contains thousands of nodes.

**Query Decomposition.** Our contribution is a collection of techniques to decompose the problem of checking the equivalence of the above formula into smaller sub queries, which can be easily checked by an SMT solver. First, we create an alternative, yet equivalent formula by moving the verification conditions for the intermediate nodes to the premise, which is shown below. The left hand side of the \( \implies \) is a conjunction of precondition and the encoding of each node \( n \in (P_1 \cup P_2) \). The right hand side specifies which variables should be equivalent.

\[
\forall_{a,b,c,eax,ebx,ecx} \text{Pre} \implies (T_1 = a \gg 13) \land (T_2 = a \gg 9) \land (T_3 = a \& b) \land (T_4 = \neg a) \land (T_5 = T_4 \land c) \land (T_6 = T_1 \oplus T_2) \land (T_7 = T_3 \lor T_5) \land (T_8 = \text{ebx} \lor \text{ecx}) \land (T_9 = \text{eax} \gg 4) \land (T_{10} = \text{eax} \lor T_8) \land (T_{11} = T_9 \lor \text{eax}) \land (T_{12} = T_10 \lor \text{ecx}) \land (T_{13} = T_{11} \gg 9) \land (O_1 = T_6 \lor T_7) \land (O_2 = T_{13} \lor T_{12})
\]

\[
\implies (O_1 = O_2)
\]

(EQ3)

The [EQ3] query can be easily constructed from the DAGs and enables easier debugging.

Second, our tool automatically finds equivalent intermediate nodes to perform query decomposition. For example, if we can deduce that \( T_6 = T_{13} \) and \( T_7 = T_{12} \), we can easily prove the equivalence of \( O_1 \) and \( O_2 \) in Figure 2. CASM-VERIFY identifies likely equivalent nodes in \( P_1 \) and \( P_2 \) by concretely executing the respective DAGs with random inputs generated using the SMT solver (i.e., models that satisfy the precondition). CASM-VERIFY subsequently verifies that these likely equivalent nodes are indeed equivalent by generating verification conditions for their equivalence. The nodes are checked in reverse topological order (i.e., nodes that are closer to the leaf nodes are checked first). When CASM-VERIFY proves the equivalence of intermediate nodes, it merges the two nodes in the DAG (see Figure 4). The verification of subsequent nodes use the merged DAG.

Third, we propose a technique to accelerate the equivalence checking with the merged DAG. When we are checking the equivalence of two nodes that have common descendants, we construct a query that ignores the entire sub-tree under the common descendant (i.e., the node can take any value). The operations in the sub-tree under the common descendant constrains the range of values seen by the common descendant. Our optimization ensures that the two nodes are equivalent for all values of the common descendant. Hence, our technique is sound (i.e., when our technique states two nodes are equivalent, they are indeed equivalent for all inputs). If we cannot show equivalence when the common descendants are unconstrained, we construct a verification condition where only the input variables are unconstrained (i.e., similar to [EQ3]). Section [III-D] provides a detailed algorithm.

Fourth, we propose optimizations to reduce the size of the query in the presence of memory operations. When the program uses memory locations, the memory operations create a chain of nodes. We propose a limited form of Ackermannization, which converts operations from theory of arrays into a set of nested if-then-else expressions (see Section [IV] for more details). These optimizations enable us to show the equivalence of large programs.

**III. QUERY DECOMPOSITION FOR EQUIVALENCE**

Given two implementations, CASM-VERIFY performs automatic equivalence checking by simplifying the queries given to the SMT solver. First, it identifies likely equivalent nodes with concrete execution using random inputs. Second, it constructs queries to check whether the identified nodes are...
Indeed equivalent for all inputs. Third, it simplifies the DAG by merging equivalent nodes and generates simpler queries to prove the postcondition.

A. Identifying Likely Equivalent Nodes

To enable query decomposition, we need to identify intermediate nodes in the DAG that are equivalent. We can subsequently simplify the DAG by merging equivalent nodes. Requiring the user to provide such information is typically infeasible because there are thousands of intermediate nodes.

Sample input generation using counter-example guided enumeration. Inspired by data-driven approaches for generating likely invariants [29, 16], CASM-VERIFY generates random inputs for the leaf nodes in the DAG using the SMT solver. Since any such random input has to satisfy the precondition, CASM-VERIFY asks the SMT solver for a model that satisfies the precondition. A single input is typically not sufficient to identify likely equivalent nodes. Hence, CASM-VERIFY uses counter-example guided model enumeration iteratively to generate multiple random inputs that satisfy the precondition. Initially, any input that satisfies the precondition is used. In the subsequent iterations, CASM-VERIFY asks the SMT solver to provide models that satisfy the precondition and are distinct from the previous random inputs generated.

Once a set of sample inputs is generated, CASM-VERIFY evaluates the DAG using the concrete values for the leaf nodes. It groups all intermediate nodes that produce identical values for all inputs in the sample set of inputs as likely equivalent nodes. The likely equivalent nodes are indistinguishable from each other with respect to the set of sample inputs (i.e., they may not be equivalent for other inputs). Using concrete execution quickly prunes the set of intermediate nodes that we need to check using expensive SMT solver queries.

B. DAG Simplification for Equivalence Checking

Once we identify likely equivalent nodes, the next step is to check if these nodes are indeed equivalent. If they are equivalent, then we merge the two nodes in the DAG, which reduces the number of nodes and produces simpler formulae for the verification of subsequent nodes. Figure 3 provides our algorithm to prove the postcondition given the DAG, the set of likely equivalent nodes, and the precondition. It returns true if the postcondition is valid (i.e., the two implementations are equivalent) and false otherwise.

DAG simplification in reverse topological order. When we check the equivalence of nodes and simplify the DAG, we perform it in the reverse topological order. We check nodes that are closer to leaves first and subsequently explore nodes that are farther away. This approach ensures that most of the descendants of a node are already merged when a node close to the root is encountered. We assign a rank to each node, which indicates the maximum distance of the node from the descendant leaf nodes.

To simplify the DAG, the algorithm in Figure 3 identifies a representative node that has the lowest rank for each likely equivalent set. These nodes are aggregated in the set $R$. Figure 3 maintains the invariant that any two nodes of $R$ are not equivalent to each other. The algorithm in Figure 3 maintains all other nodes in the DAG as set $L$ (line 3), which is also sorted by their rank. Subsequently, it checks if a node in $L$ is equivalent to a representative node (lines 5-15).

If the two nodes are equivalent, then it merges $u$ and $v$ in the DAG. The merge operation changes all parent nodes of $u$ to point to $v$ (line 12). This change is also reflected in the postcondition. If $u$ appears in the post condition, it is replaced by $v$. If $u$ is not equivalent to $v$, we add $u$ to $R$, and examine the next node in $L$ (line 14). Before checking equivalence, we also perform optimization of memory accesses (see Section IV).

Illustration. Let’s consider the case where the likely equivalent sets from concrete execution with sample inputs for the implementations in Figure 1 are: \{$a, eax\}$, \{$b, ebx\}$, \{$c, ecx\}$, \{$T6, T13\}$, \{$T7, T12\}$, and \{$O1, O2\}$. The resulting $R$ and $L$ sets are:

$R = \{a, b, c, T6, T7, O1\}$

$\text{CheckEQ}(u, v, I, \text{pre})$

$\Psi \leftarrow \{u, v\} \cup \text{Descendant}(u) \cup \text{Descendant}(v)$

$\Phi \leftarrow \bigwedge_{s \in \Psi} \text{Encode}(s)$

$\varepsilon \leftarrow \forall_{t} (\text{pre} \land \Phi) \implies (u = v)$

$r \leftarrow \text{CheckSat}(\neg \varepsilon)$

$\text{return } r = \text{unsat}$
D. Quick Check Equivalence

The algorithm in Figure 3 verifies the equivalence of two nodes $u$ and $v$ using the CheckEQ function. It first constructs a set $\Psi$ that consists of $u$, $v$, and all descendant nodes of $u$ and $v$ in the DAG (line 21 in Figure 3). Subsequently, it creates a constraint for each node using the Encode() function and creates a conjunction. The Encode($\psi$) function returns true if $\psi$ is a leaf node. If $\psi$ is an intermediate node or a root node, Encode($\psi$) yields the symbolic expression for the value of $\psi$ in terms of its child nodes.

Therefore, $\Phi$ is a conjunction of predicates that collectively evaluates the value of $u$ and $v$ in terms of the leaf nodes. For example, when we are trying to verify the equivalence of $T6$ and $T13$ in Figure 4(a), we would produce:

$$
\Psi = \{T1, T2, T6, T9, T11, T13, 4, 9, 13, a\}
$$

$$
\Phi = (T1 = a \gg 13) \land (T2 = a \gg 9) \land (T6 = T1 \gg T2) \land (T9 = a \gg 4) \land (T11 = a \gg T9) \land (T13 = a \gg 9)
$$

We check equivalence by checking the validity of the formula for all valuations of the input variables when the precondition is satisfied. For verifying the equivalence of $T6$ and $T13$, we prove the validity of the following formula:

$$
\varepsilon = \forall a, eax, ebx, ecx (pre \land \Phi) \implies (T6 = T13)
$$

C. Verifying the Equivalence of Two Nodes

The algorithm in Figure 3 verifies the equivalence of two nodes $u$ and $v$ using the CheckEQ function. It first constructs a set $\Psi$ that consists of $u$, $v$, and all descendant nodes of $u$ and $v$ in the DAG (line 21 in Figure 3). Subsequently, it creates a constraint for each node using the Encode() function and creates a conjunction. The Encode($\psi$) function returns true if $\psi$ is a leaf node. If $\psi$ is an intermediate node or a root node, Encode($\psi$) yields the symbolic expression for the value of $\psi$ in terms of its child nodes.

Therefore, $\Phi$ is a conjunction of predicates that collectively evaluates the value of $u$ and $v$ in terms of the leaf nodes. For example, when we are trying to verify the equivalence of $T6$ and $T13$ in Figure 4(a), we would produce:

$$
\Psi = \{T1, T2, T6, T9, T11, T13, 4, 9, 13, a\}
$$

$$
\Phi = (T1 = a \gg 13) \land (T2 = a \gg 9) \land (T6 = T1 \gg T2) \land (T9 = a \gg 4) \land (T11 = a \gg T9) \land (T13 = a \gg 9)
$$

We check equivalence by checking the validity of the formula for all valuations of the input variables when the precondition is satisfied. For verifying the equivalence of $T6$ and $T13$, we prove the validity of the following formula:

$$
\varepsilon = \forall a, eax, ebx, ecx (pre \land \Phi) \implies (T6 = T13)
$$

Query simplification due to node merges. When we discover a pair of equivalent nodes, the parents of the two nodes point to one sub-tree under them. The query to check the validity of the formula will not use a large number of nodes in the original DAG. Since Figure 4(c) is the DAG obtained after merging equivalent nodes, the formula to check the equivalence of $O1$ and $O2$ will not include symbolic expressions corresponding to the nodes $T8, T9, \ldots$, and $T13$.

Verifying the postcondition. Once all equivalent nodes are merged, verifying the postcondition is straightforward. The postcondition states the pairs of output nodes that should be equivalent. If they are equivalent, then our algorithm would have merged these nodes and the postcondition is trivially satisfied. Otherwise, the two output variables in the postcondition would be distinct. Hence, the algorithm in Figure 3 checks if all the output nodes have been merged.
function QuickCheckEQ(u, v, I, pre):
    \( \Psi \leftarrow \{u, v\} \cup \text{Descendant}(u) \cup \text{Descendant}(v) \)
    \( \Upsilon \leftarrow (\text{Descendant}(u) \cap \text{Descendant}(v)) \)
    \( \Phi \leftarrow \wedge_{v \in \Psi} \text{Encode}(\psi) \)
    \( \varepsilon \leftarrow \forall_{I, \Upsilon} (\text{pre} \wedge \Phi) \implies (u = v) \)
    \( r \leftarrow \text{CheckSat}(\neg \varepsilon) \)
    return \( r = \text{unsat} \)

Fig. 5. The QuickCheckEQ algorithm verifies the equivalence of \( u \) and \( v \) by universally quantifying the common descendant nodes of \( u \) and \( v \). If QuickCheckEQ returns true, then \( u \) and \( v \) are equivalent. If it returns false, then we cannot conclude whether \( u \) and \( v \) are equivalent or not.

Fig. 6. An example DAG to illustrate the usefulness of QuickCheck optimization. We want to check the equivalence of \( T7 \) and \( T10 \). \( T3 \), \( c \), and \( d \) are common descendants of both \( T7 \) and \( T10 \). The symbolic expression to check equivalence of \( T7 \) and \( T10 \) can be written in terms of \( T3 \), \( c \), and \( d \).

Common descendants are not used for verification condition generation (see Figure[5]). Hence, the queries are much smaller and can be solved quickly by SMT solvers.

When our optimization QuickCheckEQ states that two nodes are equivalent, they are indeed equivalent because we have shown their equivalence for any value of the common descendant node. The sub-tree under this node only constrains the values that the node can produce. When our optimization cannot prove the equivalence of two nodes, we cannot conclusively state that they are not equivalent. We have to resort to the default CheckEQ function that constructs the verification condition using the leaf nodes.

Illustration. We illustrate the query simplification with our quick check optimization. Consider the DAG in Figure[6]. Let us consider the case where we want to show the equivalence of \( T7 \) and \( T10 \). The validity check with our quick check optimization is:

\[
\forall_{T3,c,d} ((T4 = T3 \land c) \land (T5 = \neg T3) \land (T6 = T5 \land d) \land (T7 = T4 \uplus T6) \land (T8 = c \oplus d) \land (T9 = T3 \land T8) \land (T10 = T9 \oplus d)) \implies (T7 = T10)
\]

Note that the above formula generated by our quick check optimization universally quantifies the common descendant \( T3 \) and the input variables that are used in the formula. It does not use any node in the sub-tree under \( T3 \).

IV. SIMPLIFICATION WITH MEMORY ACCESSES

Memory accesses are common in specifications of cryptographic algorithms, especially for various look-up tables. Further, assembly implementations can also have spill code apart from regular memory accesses. CASM-VERIFY reasons about programs in the presence of memory accesses. CASM-VERIFY uses the theory of arrays to generate verification conditions for memory accesses. Hence, every write operation creates a new array that is exactly identical to the original array except for the index where the element is written. A write operation to the array \( A_0 \) at index \( i \) with the value \( v \) results in the creation of a new array \( A_1 \), such that \( A_1[i] = v \), and the value of \( A_1 \) is equivalent to \( A_0 \) in all other index planes. More formally,

\[
A_1 \leftarrow \text{write}(A_0, i, v) \implies A_1[j] = \begin{cases} v & \text{if } j = i \\ A_0[j] & \text{otherwise} \end{cases}
\]

Long paths in the DAG due to memory operations.

As a consequence of using the theory of arrays to encode memory operations, programs that perform a series of memory operations can have long paths in the DAG. Implementations of cryptographic algorithms can perform hundreds of memory read/write operations.

Consider a sequence of memory accesses shown in Figure[7(a)], which writes three values, \( v1 \), \( v2 \), and \( v3 \) to the array \( A \) at indices \( i1 \), \( i2 \), and \( i3 \), respectively. Subsequently, it reads the value in the array at index \( i4 \) and stores it in
The DAG representation of the sequence of accesses is shown in Figure 7(b). The DAG representation contains four array nodes, A0, A1, A2, and A3. A0 represents the state of array A before executing any instruction. A1, A2, and A3 represent the states of A after executing the first, second, and the third write operation, respectively. To reason about the value of T14, the verification condition, by default, includes constraints about all nodes in the DAG irrespective of the values of the indices (i.e., i1,...,i4) or the values (v1,...,v4). Our memory optimization leverages the equivalence relationship between the index nodes to simplify the read nodes.

Optimizing memory reads in the DAG. To optimize memory read nodes (i.e., MemoryOpt in the algorithm in Figure 3), we first transform the DAG into a collection of if-then-else nodes, which is a limited form of Ackermannization [30]. It allows us to reason about array operations using a simple bitvector theory rather than a combination of multiple theory solvers.

Figure 7(c) presents the equivalent DAG after transforming the memory operations into nested if-then-else nodes. In Figure 7(b), the child of the read operation at index i4 (i.e., T14) is a memory write operation A3 that writes value v3 at index i3. We convert it into an if-then-else node with three children: comparison node (i4 = i3), value node v3 for the if part, and nested if-then-else tree for the else part as shown in Figure 7(c). This process is repeated until all the memory write operations are converted into if-then-else nodes.

Using node equivalence information to prune the if-then-else nodes. Given a DAG with if-then-else nodes, we use the equivalence checking procedure in Figure 3 to determine the aliasing information between the memory read and write operations and perform dead branch elimination. If the indices are equivalent for all inputs, we eliminate the false branch. If the indices are distinct for every input, we remove the true branch. If the indices may be equal for some inputs, then we keep both branches. Based on the aliasing information inferred from Figure 7(d), the DAG in Figure 7(c) is optimized to the DAG in Figure 7(e).

V. EXPERIMENTAL EVALUATION

We describe our prototype, our methodology for evaluating the equivalence of two implementations, and the effectiveness of our tool with both existing and mutated implementations of cryptographic algorithms.

A. Prototype

Our prototype, CASM-VERIFY, is implemented in Python and uses the Z3 SMT solver. The implementations of cryptographic algorithms can be provided either in x86 assembly or in our tool’s domain specific language. CASM-VERIFY provides a simple C-like imperative language to specify reference implementations. It supports common logical, arithmetic, and bitwise operations. It also supports memory accesses as reads/writes over an array. The DSL supports fixed iteration loops, mathematical functions, and ternary operators, which are common in the specification of cryptographic algorithms. The DSL constructs have one-to-one correspondence to theory of Bit-Vectors and theory of array operations in first-order logic (i.e., SMT-LIB theories).

CASM-VERIFY translates the assembly implementation to the internal DSL. CASM-VERIFY’s translator precisely captures side-effects of each assembly instruction. CASM-VERIFY’s translator for assembly instructions supports x86 32/64-bit modes and SSE instructions. The translator tracks changes in the 64-bit registers, its sub-registers, 128-bit xmm registers, flag registers, and memory. When identifying equivalent nodes, CASM-VERIFY only considers the equivalence of variables of the same size. Hence, it extracts the appropriate bits before comparing equivalence.

CASM-VERIFY is open-source. It is publicly available at https://github.com/rutgers-apl/CASM-Verify.

B. Applications and Methodology

We evaluated our prototype with eight different assembly implementations of three different algorithms in OpenSSL: x86_64 and SSE implementations of SHA-256 hashing algorithm, x86_64 and SSE implementation of ChaCha20 stream cipher, and x86_64 implementation of AES-128 encryption, decryption, and the two key expansion implementations used for encryption and decryption, respectively. We compared these eight existing implementations against reference implementations that we wrote in CASM-VERIFY’s DSL. We based our reference implementation on the available standards: FIPS 180 [31] for SHA-256, Bernstein’s paper [32] for ChaCha20, and FIPS 197 [33] for AES-128. Additionally, we also evaluated the equivalence of x86_64 and SSE implementations of SHA-256 and ChaCha20 using our prototype. We use a 12-hour total time limit for CASM-VERIFY and a five minute limit for each SMT query generated by CASM-VERIFY. If our tool experiences a time-out at any stage, we conclude that CASM-VERIFY cannot successfully verify the benchmark. For SHA-256 and ChaCha20 implementations that predominantly read/write word-sized values, we model memory using an array of 32-bit values. For AES-128 implementations, we model memory using an array of 8-bit values.

C. Effectiveness in Checking Equivalence

CASM-VERIFY was able to verify the equivalence for all of our experiments within the time limit. DAG simplification and optimizations are important to verify the implementations. Figure 8 reports the time taken by the existing techniques (leftmost bar of each cluster) and our tool with DAG simplification and memory access optimizations (rightmost bar of each cluster) for the 10 different configurations in Figure 9. Figure 8 reports that the default equivalence checking technique times out with all applications. In contrast, we are able to successfully verify equivalence with CASM-VERIFY. To understand the benefit of our optimizations, we also evaluated the applications with two additional configurations: (1) CASM-VERIFY without quick check and memory read optimization (second bar from the left of each cluster in Figure 8).
and (2) CASM-VERIFY without memory read optimization (third bar from the left of each cluster in Figure 8). Only 3 out of the 10 verification scenarios completed without quick check and memory read optimizations. Addition of quick check optimization with query decomposition allows CASM-VERIFY to verify 4 out of the 10 verification scenarios. Inclusion of memory read optimization enables CASM-VERIFY to verify all of them, although it adds some slowdown with the AES-key-enc scenario. We hypothesize that the cost of optimizing memory read operations outweighs the complexity reduction of verification condition in this scenario. In summary, our optimizations together enable successful automated verification of implementations of cryptographic algorithms.

Figure 9 reports the statistics of our verification scenarios. CASM-VERIFY was able to verify 8 out of 10 benchmarks within an hour, while SHA-equiv took over two hours and AES-key-dec took over three hours to complete. CASM-VERIFY spent the majority of the time optimizing memory read operations for both benchmarks, because they contain hundreds of memory write operations. Almost all likely equivalent nodes obtained from concrete execution were indeed equivalent. The difference between the third and fourth columns in Figure 9 provides the number of likely equivalent nodes that were not equivalent with symbolic verification. Only 15 pairs of likely equivalent nodes were not equivalent when we verified with symbolic expressions, which illustrate the usefulness of concrete execution with sample inputs.

D. Evaluation with Program Mutations

To test the ability of our tool to detect incorrect implementations, we injected bugs in the program using a custom-program mutator. Figure 10 presents various kinds of program mutations performed by our mutator to test the effectiveness of our tool. Among these, the first class of mutations are representative of bugs that developers make while implementing the program in an assembly language. These include (1) using a wrong but a similarly named instruction, (2) using...
mov registers. For example, we can replace a
Figure 10(g) illustrate such mutations.
they are indeed not equivalent. Figure 10(e), Figure 10(f), and
transformation creates an implementation that is only incorrect
to the destination operand if the zero flag is not set. This
cmov

casm-v

on inputs where

create an implementation that is only incorrect

addl %eax, 20(%rsp)

andl %eax, %ebx

subl %edi, %ebx

movl %eax, 20(%rsp)

cmovzl %eax, %r15d

andl %eax, %ebx

addl %edi, %ebx

movl %eax, %eax

addl %edi, %ebx

addl %edi, %ebx

cmovnzl %eax, %r15d

Figure 10(g) reports an interesting mutation where %r10 is
shifted right by 7 bits and subtracted from %rbx. The first
sub instruction will not set the carry flag provided
%rbx is greater than 2^7. This gives us 0.79% chance for the first
sub instruction to set the carry flag. We modified the second
sub instruction to a subtract with borrow (sbb)
instruction, which subtracts with the carry flag. However,
casm-verify reported that this new implementation is a
correct implementation. Upon further inspection, we found out
that the new implementation was indeed correct. There was a
mov instruction a few instructions above this region of code:

movq %r10,%rbx

It implies %rbx ≥ %r10 is true for all inputs. Hence, the first
sub instruction will never set the carry flag. Replacing the second
sub instruction with a sbb instruction does not
change the semantics of the program in this case.

Semantically equivalent program mutations. For the last
set of experiments, we tested the capability of our tool to verify
that the implementations are indeed correct with semantically
equivalent program mutations. Figure 10(h) illustrates one
such method where a mov instruction is replaced with a
sequence of cmovnz and cmovnzd instructions. Regardless of the
zero flag, the mutated program copies the value of
%eax to %r15d and is semantically equivalent to the original
program. Our tool was able to verify the equivalence of all
such semantically equivalent implementations.

VI. RELATED WORK

There is a large body of work on verifying algorithms for
cryptography and equivalence checking for general purpose
programs. We describe closely related work in this section.

Verification of existing cryptographic algorithms. There is a
large body of work that individually verifies the
correctness of widely used implementations. SHA-256 [6],
HMAC/SHA-256 [7] implementation from OpenSSL, and
HMAC-DRBG [34] from mbedTLS have been verified using
Verifiable C [35] and the Coq proof assistant [36]. Chen et
al. [37] verified the core part of Curve25519 [38] written in
qhasm [39] using Boolector [40] and the Coq proof assistant.
Proving the correctness of individual implementations can
provide a strong end-to-end guarantee. In contrast to interac-
tive verification, we present a first step toward automatically
checking the equivalence of cryptographic implementations
with minimal user input.

SAW [41], [13] verifies cryptographic algorithms implemented
in Java and LLVM bytecode against the reference
implementation written in Cryptol [42]. SAW uses rewrite
rules, memoization, and hash-consed DAG structures to
identify semantically identical terms to accelerate the verifica-
tion process. In contrast to SAW, CASM-VERIFY checks the
equivalence of cryptographic algorithms implemented in

a wrong register, (3) mistakes with source and destination
operands (i.e., confusion between AT&T and Intel syntax),
and (4) use of wrong offsets with displacement addressing
mode. We created 40 distinct test cases using our mutator
with the ChaCha20 implementation. CASM-VERIFY was able
to detect that all these test cases were not equivalent to the
reference implementation. Most of them were detected during
concrete execution with random inputs for identifying likely
equivalent nodes.

Hard to find program mutations. For the second set of bug
injection experiments, we transformed the original assembly
implementations such that the implementations are correct for
most of the inputs, but incorrect for some inputs. In this
case, the concrete execution will identify output variables to
be likely equivalent, but CASM-VERIFY should detect that
they are indeed not equivalent. Figure 10(e), Figure 10(f), and
Figure 10(g) illustrate such mutations.

Figure 10(e) utilizes assembly instructions that use flag
registers. For example, we can replace a mov instruction
with a cmov instruction, particularly cmovnzd instruction. The
cmovnz instruction copies the value in the source operand
to the destination operand if the zero flag is not set. This
transformation creates an implementation that is only incorrect
on inputs where %r15d + %r12d = 0.

Figure 10(f) illustrates an example that uses add with carry
(adc) instruction to create a mutant. Here, we have two
consecutive add instructions. The value stored in the memory
at address %rbp is a constant value 0x06ca6351. Therefore,
the chance of the first instruction to set the carry flag is roughly
2.7% (If %r12d > 0xf9359caf). Hence, the implementation
will be incorrect for 2.7% of all possible inputs when we
replace the second add instruction with adc.

We created a total of 42 distinct implementations with such
mutations. Most of them were not caught during concrete
execution but CASM-VERIFY still reported that they are
not equivalent in all these cases because it performs sound
symbolic verification.

Figure 10(g) illustrates such mutations that create bugs that are hard to detect with
testing. (h) illustrates a mutation that does not change the semantics of the
program but creates nodes that are almost equivalent but cannot be merged.
assembly and uses SMT Solvers to identify equivalent nodes automatically.

Axe [14] is another tool that verifies cryptographic algorithms implemented in Java against the reference implementation written in Java or given as a mathematical specification. Similar to CASM-VERIFY, Axe converts both implementations to DAGs, uses concrete execution to identify likely equivalent intermediate variables, formally verifies the equivalence of these variables using STP solver [43], and merges equivalent nodes. Axe also reduces the complexity of the verification condition by heuristically unconstraining intermediate variables, similar to CASM-VERIFY’s quick check optimization. Unlike a high level language such as Java, assembly languages have a finite number of registers and the majority of the data (i.e., look up table, keys, and messages) must be stored in memory. As a consequence, reasoning about assembly implementations requires reasoning about a long chain of memory accesses. In contrast to Axe, CASM-VERIFY uses memory read optimization to reduce the complexity of reasoning about memory accesses.

**Correct by construction implementations of cryptographic algorithms.** Another method of guaranteeing correct implementations of cryptographic algorithms is to use programming languages designed for efficient verification to verify the implementation during development. Project Everest uses F* [3] to implement the record layer of TLS 1.2 [44] and TLS1.3 [9] protocols, the underlying cryptographic algorithms [10], and elliptic curve algorithms [45]. Vale [11] language formalizes assembly instructions in Dafny [4] to implement cryptographic algorithms. Implementations can be automatically proven using the specification written in Dafny. Jasmin [5] is a programming language inspired by qasm for generating memory safe and constant-time implementations of cryptographic algorithms. Jasmin compiler is formally verified to preserve the safety properties. Developing verified implementations using languages designed for verification is desirable for implementing new algorithms. In contrast, CASM-VERIFY is aimed towards verifying existing widely used assembly implementations and incremental modifications to them.

**Verification of assembly code.** Bedrock [46], x86proved [47], and BoogieX86 [48] provide tools to reason about the correctness of assembly language with complex control flows and data structures at the cost of manual programmer effort. We plan to use the combination of these ideas with CASM-VERIFY. DDEC [16] verifies the equivalence between an x86 program and the optimized x86 program by identifying likely correlating program points and invariants, also known as simulation relation, using execution traces from real program executions. Simulation relation is used to identify correlating program fragments and generate verification conditions that can compositionally prove the equivalence of source and target programs. CASM-VERIFY can leverage these ideas to prove algorithms with loops. When compared to them, CASM-VERIFY performs significant simplification with quick check and memory read optimizations and automatically checks the equivalence of two implementations.

**Tools from compiler verification.** Translation validation in the context of compiler verification also utilize simulation relations or symbolic verification to check the equivalence of the source program and the compiled program [19], [24], [15], [20], [49], [50], [26], [27], [28]. Recently, Dahiya et al. [20], [51] compare the graph representation of program paths and use synthesis techniques to identify simulation relations. CASM-VERIFY can use precondition inference techniques [52] to identify appropriate preconditions in our context.

Buchwald et al. [53] synthesized a set of rules for instruction selection using counterexample-guided inductive synthesis techniques. They reason about one machine instruction at a given instant of time and can efficiently model memory by representing only the memory region that the instruction accesses. However, CASM-VERIFY needs to reason about multiple machine instructions, so it models memory accesses using nested if-then-else expressions of bitvector values.

**Query decomposition.** Gupta et al. [23] extend the work of Dahiya et al. [20], [51] and simplify verification conditions by identifying all equivalent sub-expressions of the verification condition using an SMT solver while using counter examples to prune the search space. This technique is similar in spirit to CASM-VERIFY. However, their approach is not sufficient to verify implementations of cryptographic algorithms as shown by the need for our quick check and memory read optimizations. Feng et al. [54] verifies the equivalence of embedded software by identifying cut-points (i.e., points with equivalent memory state) to simplify the verification condition. Rather than the entire memory state, CASM-VERIFY tracks equivalence of values at a memory location.

CASM-VERIFY is inspired by SAT Sweeping [17], and its variants [55], [56], [57], [17], which are used to check the equivalence of circuit boards represented in And-Inverter Graphs. Beyond these ideas, CASM-VERIFY also reasons about memory and provides memory read optimization to further simplify verification conditions.

**Beyond equivalence checking.** A number of tools have been developed to detect side-channel vulnerabilities that can leak secret information through memory or through caches [58], [59], [60], [61]. Notably, et-verif [60] verifies that cryptographic algorithms in optimized LLVM bitcode are constant time and Barthe et al. [61] verifies that the assembly program compiled from CompCert [62] is constant time. We plan to extend our tool to check such properties in the future.

**VII. Conclusion**

This paper presents a set of techniques to automatically check the equivalence of two implementations of cryptographic algorithms. We address the challenge of verifying the validity of a large formula that encodes the equivalence of an implementation to its reference implementation, where each implementation can contain thousands of instructions,
by decomposing it into smaller formulae. We propose the use of concrete inputs to identify likely equivalent variables and subsequent symbolic reasoning for these likely equivalent variables. Our optimizations for quick equivalence checks and memory reads enable the use of CASM-VERIFY to verify SHA-256, ChaCha20, and AES-128 from OpenSSL. We plan to check the equivalence of implementations of various algorithm in qhasm and incorporate reasoning about constant-time implementations as future work.

APPENDIX

A. Abstract

CASM-VERIFY is open source and publicly available [8]. We also provide an archival link of the artifact. The artifact contains the source code of CASM-VERIFY, all benchmarks used for the experimental evaluation, and scripts to automatically run experiments and reproduce our results. To ease installation effort, we include a Dockerfile in the artifact that automatically builds a Docker image containing the required software and CASM-VERIFY.

B. Artifact Check-list (Meta-information)

- **Algorithm:** SAT Sweeping, Equivalence Checking, Symbolic Evaluation, Ackermannization.
- **Program:** Python3 and Z3
- **Data Set:** All benchmarks are included in the artifact.
- **Run-time Environment:** Experiments were performed on macOS High Sierra. We have verified that CASM-VERIFY works on Ubuntu as well.
- **Hardware:** Experiments were performed on a machine with 2.6GHz Intel Core i5 and 8GB memory. Similar hardware should produce comparable results.
- **Metrics:** The included scripts report on the expected results as well as the estimated amount of time required for each experiment.
- **Output:** CASM-VERIFY outputs whether the implementation and the reference implementation are equivalent or not. Results are output to the console.
- **Experiments:** Build the Docker image, run the Docker image, run the test scripts, and observe the results.
- **How Much Time Is Needed to Prepare Workflow (Approximately)?** The Docker image builds in less than 5 minutes.
- **How Much Time Is Needed to Complete Experiments (Approximately)?** All experiments described in Section V together take approximately 45 hours.
- **Publicly Available?** Yes.

C. Description

1) **How Delivered:** The artifact can be downloaded from the archive at [https://doi.org/10.5281/zenodo.2229779](https://doi.org/10.5281/zenodo.2229779)

2) **Hardware Dependencies:** Our experiments were performed on a machine with 2.6GHz Intel Core i5 with 8GB memory. Any similar hardware should produce comparable results.

3) **Software Dependencies:** CASM-VERIFY is written in Python3 and uses Z3. Both Python3 and Z3 are automatically installed in the Docker image via Dockerfile.

D. Installation

1) **Installation Using Docker:** Install Docker by going to [https://docs.docker.com/install/](https://docs.docker.com/install/) Select the corresponding OS system on the left side bar under Docker CE, and follow the instructions. For macOS users, increase the memory usage limit to 8GB by choosing the advanced tab in preferences and adjusting the memory usage limit.

Next, download the artifact from the archive and extract it using the following commands:

```
$ tar -zxvf CASM_VERIFY.tar.gz
$ cd CASM_VERIFY
```

Finally, the Docker image can be built and run using the following commands:

```
$ docker build --t casmverify
$ docker run -it casmverify
```

2) **Installation Without Using Docker:** To evaluate the artifact without using Docker, install Python3 and Z3. In Ubuntu, this can be done using the following commands:

```
$ sudo apt-get install python3 python3-pip
$ python3 -m pip install z3-solver
```

In macOS, use homebrew to install the required software:

```
$ brew install python
$ python3 -m pip install z3-solver
```

Then, download the archive and extract it using the following commands:

```
$ tar -zxvf CASM_VERIFY.tar.gz
$ cd CASM_VERIFY
```

E. Experiment Workflow

The artifact provides separate scripts for each experiment performed in Section V.

a) **Test1_benchmark.sh:** This script runs experiments that check the equivalence of various assembly implementations of cryptographic algorithms found in OpenSSL using CASM-VERIFY. The result of this experiment produces Figure 9 and the right most bar of each cluster in Figure 8. It takes approximately 10.5 hours.

b) **Test2_nodeMergeOnly.sh:** This script runs experiments to perform equivalence checking using CASM-VERIFY with the quick check and memory read optimizations disabled. The result of this experiment produces the second bar from the left of each cluster in Figure 8. It takes approximately 4.5 hours.

c) **Test3_quickCheck.sh:** This script runs experiments to perform equivalence checking without using CASM-VERIFY’s memory read optimization. The result of this experiment produces the third bar from the left of each cluster in Figure 8. This script takes approximately 6.5 hours.

d) **Test4_developMistakeBug:** This script tests the ability of CASM-VERIFY to effectively detect bugs during development. The implementations are mutated with bugs that developers can make while implementing an algorithm in an assembly language. This script runs for approximately 25 minutes.

e) **Test5_hardToFindBug.sh:** This script tests the ability of CASM-VERIFY to effectively detect hard to find bugs. The implementations are mutated with various bugs that appear to be correct for most inputs, but incorrect for some inputs. This script runs for approximately 18.25 hours.

f) **Test6_additionalEquivalenceTest.sh:** This script tests the ability of CASM-VERIFY to correctly verify the mutated implementations when the mutations preserve the semantics. This script runs for approximately 4.5 hours.

g) **Test7_naiveQuery.sh:** Instead of using CASM-VERIFY, this experiment verifies the equivalence of two implementations using a single query. This experiment corresponds to the leftmost bar of each cluster in Figure 8. Note that all ten benchmarks will not complete within the time limit (12 hours).
F. Evaluation and Expected Result

The scripts print the expected outputs and the estimated amount of time required to complete each experiment. The results can also be compared to the data presented in Section [V] Depending on the platform and the hardware, the time taken by each experiment may vary by a small amount.

G. Experiment Customization

The source code and the benchmarks are provided in the artifact. The software dependencies for CASM-VERIFY—Python3 and Z3—are available on all major operating systems: Windows, Linux, and macOS. Hence, these experiments can be run on any of these platforms.

CASM-VERIFY can be run with different assembly implementations. However, the user needs to provide the precondition, the postcondition, and the reference implementation. We provide an additional micro benchmark not used in our evaluation to showcase this feature in test/sha2rnd directory. In order to run this benchmark, use the following command:

```
$ python3 main.py --pre test/sha2rnd/pre 
 --post test/sha2rnd/post 
 --p1 test/sha2rnd/dsl --p1lang dsl 
 --p2 test/sha2rnd/asm --p2lang asm
```

Every script we provide uses similar commands to run CASM-VERIFY. All the benchmarks used in the experiments are located in text directory. Note that CASM-VERIFY currently supports x86_64 assembly language with AT&T syntax. We plan to extend our support for other architectures in the future. For more information on how to use CASM-VERIFY, please refer to the link: https://github.com/rutgers-apl/CASM-Verify

ACKNOWLEDGMENTS

We thank David Cash, Adarsh Yoga, Reza Soltaniyeh, Nader Moradi, and the CGO reviewers for their feedback on this paper. This paper is based on work supported in part by NSF CAREER Award CCF-1453086.

REFERENCES


