Automatic Verification of Cryptographic Block Function Implementations with Logical Equivalence Checking

Li-Chang Lai[1], Jiaxiang Liu[2(✉)], Xiaomu Shi[3], Ming-Hsien Tsai[4], Bow-Yaw Wang[5], and Bo-Yin Yang[5]

[1] National Taiwan University, Taiwan
[2] Shenzhen University, China
[3] Key Laboratory of System Software (Chinese Academy of Sciences) and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, China
[4] National Taiwan University of Science and Technology, Taiwan
[5] Academia Sinica, Taiwan

Abstract. Given a fixed-size block, cryptographic block functions generate outputs by a sequence of bitwise operations. Block functions are widely used in the design of hash functions and stream ciphers. Their correct implementations hence are crucial to computer security. We propose a method that leverages logic equivalence checking from hardware verification to verify assembly implementations of cryptographic block functions. Using our technique, we verify more than two dozen assembly implementations of ChaCha20, SHA-256, and SHA-3 block functions from OpenSSL and XKCP automatically. We also compare the performance of our technique with SMT-based techniques in experiments.

Keywords: hash functions, logic equivalence checking, formal verification

1 Introduction

Hash functions are widely used in computer security. To digest an arbitrary message, hash functions often divide the message into blocks and compute by message blocks. Because block functions dominate the computation time by nature, cryptographic libraries have several implementations of a block function for different architectures to improve efficiency. In OpenSSL, for instance, the SHA-3 block function has more than 10 implementations [37].

Implementations of cryptographic block functions are results of skillful programming and engineering. They are often manually optimized for better performance on different architectures. Such optimizations increase programming complexity significantly. If any programming mistake occurs, security guarantees of hash functions can be voided. Incorrect implementations of block functions therefore can be a devastating threat to computer security.

We propose a method to verify functional equivalence of cryptographic block functions and their implementations formally using logic equivalence checking. By functional equivalence, we mean that an implementation always generates the same output as the corresponding block function on any input. Unlike testing, formal verification aims to demonstrate the equivalence of functions and implementations through logical reasoning. It ensures that a block function implementation computes correct results for not only many but all inputs. Such a strong guarantee is preferred for critical components like block functions.

Logic equivalence checking is an automatic hardware verification technique to check functional equivalence between circuits [22]. Consider two circuits with the same primary inputs and outputs. The technique verifies whether the two circuits have the same output on all possible inputs. In principle, Boolean satisfiability solvers can falsify functional equivalence by checking whether the exclusive-or of the corresponding outputs from the two circuits is asserted on some inputs. Yet the naïve approach is scalable. Over decades, logic equivalence checking has been improved and widely applied to verifying circuit synthesis and optimization. Open-source and commercial logic equivalence checking tools are widely used by the research community and industry [26,12,33].

Our idea is simple. To check functional equivalence between cryptographic block functions and their optimized implementations, we pick a reference implementation provided by designers or programmers, and compare the reference implementation with optimized implementations. In order to apply the hardware verification technique, we transform both reference and optimized implementations to circuits. A logic equivalence checking tool is applied to verifying equivalence between the circuits derived from the reference and optimized implementations. The tool applies various heuristics during verification automatically. No human guidance is needed to check equivalence between circuits.

We build a tool called CryptoLEC to realize the method and verify block functions implementations in the ChaCha20 stream cipher, the SHA-256 and SHA-3 hash functions. Precisely, we take designers' (ChaCha20 and SHA-3) or programmers' (SHA-256) reference C implementations. Optimized assembly implementations in OpenSSL [37] and XKCP [36] are compared against their reference implementations. ChaCha20 implementations are verified within seconds; SHA-3 implementations are in 15 minutes; and SHA-256 implementations are verified in an hour. Our technique is sufficiently general and effective to verify more than two dozen assembly implementations of block function.

It is worth noting that our technique is *not* based on Satisfiability Modulo Theories (SMT) solvers. SMT solvers are a general tool designed to verify a wide range of properties. They are not optimal for functional equivalence checking. SMT-based techniques moreover encode programs by specifying relations between input and output variables. Relational encoding is again more general but not optimal. We compare two SMT-based techniques in experiments. The naïve SMT-based technique successfully verifies ChaCha20 implementations, but it fails to verify any but one implementation for SHA-256 and SHA-3.

The other SMT-based technique successfully reports 11 errors out of 28 buggy implementations while our tool reports 25 errors.

We summarize our contributions as follows.

- We propose a simple, general, effective, and automatic method to verify cryptographic block functions with logic equivalence checking.
- We introduce CryptoLEC, a tool implementing our proposed method.
- We verify 25 assembly implementations of ChaCha20, SHA-256, SHA-3 block functions from OpenSSL and XKCP using CryptoLEC. We are not aware of prior formal verification works on OpenSSL and XKCP assembly implementations of the SHA-3 block function.

Related Work The problem of verifying the functional correctness or security properties of hash functions has been studied in HACL* [38] and ValeCrypt [10]. They provide verified high-level F* [32] codes (for ChaCha20, SHA-3), and then generate C and assembly implementations respectively. The work in [1] uses EasyCrypt to prove the equivalence between the reference and efficient Jasmin implementation. The Jasmin implementation is then used to generate the assembly for SHA-3. Instead of generating new codes with manual proofs of correctness, we focus on verifying existing assembly implementations automatically. Prior works that also focus on existing code are in [2] and [23]. The former manually verifies the SHA-256 hash function in C with the proof assistant Coq, but does not verify assembly implementations. The latter develops an SMT-based technique in the tool CASMVerify to verify four assembly implementations of ChaCha20 and SHA-256 from OpenSSL. SAW and Cryptol adopt abc as the equivalence checking engine for implementations such as SHA-3 in C [16], and for hardware design of the Skein hash algorithm in VHDL [14]. Implementations are checked against hand-written specifications in the Cryptol language. Axe is a symbolic execution tool that has been used to verify existing Java implementations of AES [31], but it does not consider assembly implementations.

This paper is organized as follows. Preliminaries are briefly reviewed in Section 2. Section 3 gives an introduction to our methodology. It is followed by descriptions of various block functions and the highlighted distinctions in their implementations (Section 4). Section 5 presents our formal models for the block functions, providing the detailed exploration in these case studies. Experiments are reported in Section 6. We conclude in Section 7.

2 Preliminaries

We will use both hexadecimal and binary representations as in $0xc = 0b1100 = 12$. An *n-bit word* is a bit sequence of *bit width* n. Let w be an n-bit word. $w[i]$ denotes the i-th bit in w where $0 \leq i < n$. We write $\overline{\bullet}$, $\bullet \wedge \bullet$, $\bullet \vee \bullet$, and $\bullet \oplus \bullet$ for the bitwise NOT, AND, OR, and exclusive-or (XOR) operations respectively. Let a and b be n-bit words. $\mathrm{ror}(a, i)$ is the n-bit word obtained by rotating a to the right by i bits. $\mathrm{shr}(a, i)$ is the n-bit word of a shifting to the right by i bits. $a \boxplus b$ is the n-bit arithmetic sum of a and b.

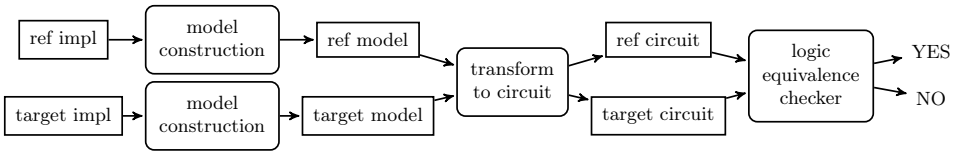

Fig. 1. Methodology Overview: General Framework

Given a bit string (called *message*) of an arbitrary length, a *hash* function computes an output bit string (called *digest*) of a fixed length. Typically, the message is divided into *blocks* of a certain size. Message blocks are processed by a *block function* consecutively to produce the digest. Block functions often compute in *rounds*. Stream ciphers may also use block functions for encryption.

Given two function descriptions, they are *functionally equivalent* if they denote the same function. That is, both descriptions yield the same result on every input. Given a *target* implementation of a block function, we want to check if it is functionally equivalent to the block function. To do so, a *reference* implementation of a block function is chosen. It suffices to check if the reference and target implementations always compute the same output on every input.

A block function often has one implementation per architectures for efficiency. We consider implementations for 32- (*armv4* and *armv7a*) and 64-bit (*aarch64* and *armv8a*) ARM architectures as well as 64-bit Intel *x86_64*, with the optional vector extensions *ssse3*, *avx*, *avx2*, *avx512*, *avx512vl*, and *shaext*.

3 Methodology

Figure 1 gives the framework of our method. To verify implementations of a block function, verifiers first need to choose a reference implementation for the block function. Our method strongly advocates that the reference implementation should be provided by block function designers or programmers, instead of the verifiers themselves. On the other hand, verifiers have in hand a target implementation to verify.

From the reference and target implementations, we construct formal models for the two implementations. A formal model is a mathematical abstraction specifying behaviors of an implementation. Program behaviors can be ambiguous. They also contain unnecessary details (such as execution time) for functional verification. To rid such ambiguities and irrelevant details, formal models are constructed to specify relevant program behaviors.

After formal models are constructed, we proceed to transform them to circuits. In general, transforming programs to circuits is a difficult problem. This transformation however is especially easy for block function implementations. For security reasons, cryptographic block functions need be constant-time. Control flows in implementations of block functions subsequently are simple. Transforming block function implementations to circuits are thus straightforward after applying standard techniques such as loop unrolling.

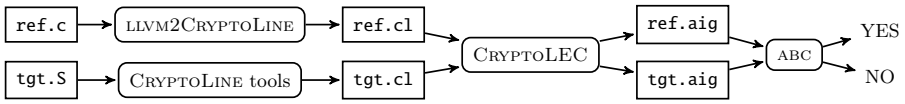

Fig. 2. Methodology Overview: Our Instantiation

With the two circuits derived from the reference and target implementations respectively, we invoke a logic equivalence checker for circuits. Logic equivalence checking [22] is a well-developed technique from hardware verification. Scalable open-source and commercial tools are widely available, for instance, [26,12,33].

Our method differs from prior works in two aspects. Previously, specifications of block functions are constructed by verifiers manually. Instead, our specifications are obtained by converting reference implementations provided by designers or programmers. Our technique greatly simplifies the effort for constructing formal specifications. More importantly, it can also reduce misinterpretations or even errors made by verifiers. After all, cryptographic block function designers are the main authority of correct specifications, not verifiers.

Secondly, our method employs logic equivalence checking for verification. The hardware verification technique has been commercialized. Logic equivalence checkers have numerous heuristics to improve their performance. Particularly, identifying potentially equivalent subcircuits is critical to its effectiveness. Once potentially equivalent subcircuits are identified, logic equivalence checking is performed by divide and conquer. By transforming formal models to circuits, our method takes advantages of heuristics from logic equivalence checking. This is the main reason for the generality and effectiveness of our method.

To evaluate our method, we provide an instantiation as depicted in Figure 2 to verify assembly implementations of block functions in the ChaCha20 stream cipher, the SHA-256 and SHA-3 hash functions. We further detail how we implemented our idea in Sections 3.1–3.3.

3.1 Model Construction

Our formal models are written in the CryptoLine modeling language [30,15,24]. CryptoLine is a domain-specific language designed for modeling cryptographic assembly programs. For the reference implementation `ref.c`, we convert its LLVM intermediate representation to the CryptoLine model `ref.cl` [21]. For the target implementation `tgt.S`, we extract its execution trace via the GNU debugger `gdb` and convert the trace to the CryptoLine model `tgt.cl`. To check equivalence between the reference and target models, `ref.cl` and `tgt.cl` need to have the same input and output variable names. Moreover, input and output variables in both models must correspond semantically to do equivalence checking.

CryptoLine The domain specific language CryptoLine is used to construct formal models [34] in this work. It is the modeling language for the automatic

verification tool CryptoLine. The CryptoLine language has modeled cryptographic primitives from elliptic curve [30,15,24,13] to post-quantum cryptography [17]. In CryptoLine, bit widths, signs of constants and variables are specified by their *types*. CryptoLine instructions consist of mnemonics and operands similar to typical assembly languages. For instance, the following CryptoLine instruction assigns an unsigned 32-bit word `0xffff` to the variable `r1`:

```
mov r1 0xffff@uint32;
```

Observe that the destination appears before the source operand. The CryptoLine type system infers the types of destination operands automatically. The variable `r1` is thus of the unsigned 32-bit word type. Now consider the conditional move instruction:

```
cmov rax carry rdx rax;
```

The instruction sets `rax` to the value of `rdx` if the bit variable `carry` is 1; `rax` is unchanged otherwise. The CryptoLine type checker ensures the source operands `rdx` and `rax` are of the same type. The destination `rax` then has the same type as both source operands.

CryptoLine supports arithmetic operations, as well as bitwise operations such as AND, OR, NOT, XOR, rotations, and shifts. A CryptoLine function is of the following form:

```
proc foo(uint32 a, uint32 b) =
{ true && true }           (*  pre-condition *)
xor x@uint32 a b;
xor y@uint32 x b;
xor x@uint32 x y;
{ true && and [x=b,y=a] }  (* post-condition *)
```

The function `foo` needs two arguments. The input variables `a` and `b` are of the type unsigned 32-bit word. The pre-condition `{ true && true }` is not used and ignored here. The post-condition `{ true && and [x = b, y = a] }` specifies that the variables `x` and `y` must be equal to `b` and `a` respectively at the end of function. Functions are invoked by the keyword `inline`. A CryptoLine program consists of functions. The `main` function is the entry point of a program.

Using the CryptoLine language, we construct reference and target models to specify the reference and target implementations respectively. The CryptoLine language however is different from x86_64 and ARM assembly. Our reference implementations moreover are written in C. They do not look like assembly at all. Some works are needed to construct formal models for implementations.

Reference Models We use the llvm2CryptoLine tool [24,13] to automate the construction of formal models for reference C implementations. We obtain the LLVM intermediate representation of the reference C implementation from the Clang compiler [21], and translate it to a CryptoLine model, not fully automatically.

The LLVM intermediate representation is designed for arbitrary C programs. Some instructions in the representation are missing in the CryptoLine language. For instance, CryptoLine does not have instructions for loops. Loops in the LLVM intermediate representation are not translatable. Lacking loops nonetheless is not a limitation for verifying block functions. To prevent side channels, loops in block functions always have a constant number of iterations. They can be unrolled by the compiler automatically. Moreover, recall formal models derived from the LLVM intermediate representation are specifications of block functions. We certainly would like to minimize (possible) errors induced by the compiler. The llvm2CryptoLine tool thus translates the LLVM intermediate representation after simple architecture-independent optimizations such as loop unrolling and constant propagation.

Target Models To construct formal models for target assembly implementations, we obtain an execution trace of the implementation with the `itrace` script from the CryptoLine tool [34]. The trace is then translated to a CryptoLine model by the `to_zdsl` script also from the tool.

More precisely, we build an executable binary code which invokes a target implementation. The `itrace` script extracts assembly instructions from the implementation using the GNU debugger `gdb` [35]. If a memory cell is accessed, the script also obtains its address. Consider the following instruction extracted from the OpenSSL Keccak-p[1600, 24] avx2 implementation:

```
vpsllvq -0x60(%r8),%ymm2,%ymm10 #! EA=0x555555555a80
```

Recall that `ymm2` and `ymm10` are 256-bit registers. Each contains four 64-bit words. The avx2 instruction `vpsllvq` shifts four 64-bit words in the `ymm2` register to the left by the offsets stored in the memory located at `-0x60(r8)` with the effective address (EA) `0x555555555a80`. Shifted results are then written to `ymm10`.

After an execution trace is obtained, we use `to_zdsl` to translate assembly instructions to CryptoLine instructions. The script requires rules for translation. The avx2 instruction `vpsllvq`, for instance, needs the following rule:

```
#! vpsllvq $1ea, $2ymm, $3ymm ->       \\
       shl $3ymm_0 $2ymm_0 $1ea;       \\
       shl $3ymm_1 $2ymm_1 $1ea[+8];   \\
       shl $3ymm_2 $2ymm_2 $1ea[+16]; \\
       shl $3ymm_3 $2ymm_3 $1ea[+24]
```

The rule matches any `vpsllvq` instruction with an effective address as its first argument (`$1ea`), and two `ymm` registers as the second and third arguments (`$2ymm` and `$3ymm` respectively). If the rule matches, the `vpsllvq` instruction is translated to four CryptoLine `shl` instructions. Each `shl` instruction shifts a 64-bit word in `$2ymm` to the left by the offset in the corresponding memory cell and writes the result to a 64-bit word in `$3ymm`. After applying the rule, we obtain the following CryptoLine fragment:

```
# vpsllvq -0x60(%r8),%ymm2,%ymm10 #! EA=L0x555555555a80
```

```
shl ymm10_0 ymm2_0 L0x555555555a80;
shl ymm10_1 ymm2_1 L0x555555555a88;
shl ymm10_2 ymm2_2 L0x555555555a90;
shl ymm10_3 ymm2_3 L0x555555555a98;
```

The first line, a comment, shows the avx2 instruction. A 256-bit `ymm`i register is modeled by four 64-bit CryptoLine variables `ymm`i`_0`, ..., `ymm`i`_3`. Memory cells are represented by variables with names corresponding to addresses (`L0x555555555a80`, ..., `L0x555555555a98`). The code shifts the four 64-bit words in `ymm2` to the left by four corresponding values in memory, writing to `ymm10`.

The model construction using `itrace` is also applicable to reference C implementations. However, the resulting execution trace is more complex compared to the LLVM intermediate representation of the implementation. Its correctness also depends on code generators during compilation. Thus we use the llvm2CryptoLine tool for reference C implementations.

3.2 Transformation to Circuits

Formal CryptoLine models differ from circuits. To apply logic equivalence checking with abc, we transform CryptoLine models (`ref.cl` and `tgt.cl`) to logic circuits (resp. `ref.aig` and `tgt.aig`) in two steps. A CryptoLine model is first converted to the BTOR format [27] then transformed to circuits in the AIGER format [28,8,9] for abc via the Boolector tool.

In hardware verification, there are two different encodings for circuits. The *relational* encoding specifies input and output signals as a relation; the *functional* encoding specifies output signals as functions on input signals. Consider an AND-gate with the output signal o and two input signals i_0, i_1. The functional encoding for the logic gate is simply $i_0 \wedge i_1$ since the output o is the logical AND function with arguments i_0 and i_1. The relational encoding for the AND-gate on the other hand is $o \Leftrightarrow i_0 \wedge i_1$. This is the characteristic function for the relation on o, i_0, i_1, or the set $\{(1,1,1),(0,0,1),(0,1,0),$ or $(0,0,0)\}$. Relational encoding can specify arbitrary relations and is hence more general. It however does not preserve circuit structures. Hardware verification techniques thus prefer functional encoding.

Most SMT solvers are not designed for hardware verification. The commonly used SMT-LIB format for SMT solvers thus uses the relational encoding [3]. To enable functional encoding, a different input format is needed. BTOR is an input format for the SMT solver Boolector [28,27]. Different from the SMT-LIB format, BTOR is designed for functional encoding. Each word in BTOR is identified by a unique number and is defined only once. Let w_i denote a word with an identification number i. Consider the following BTOR statements.

```
1 constd 64 42
2 var 64
3 sll 64 2 1
```

The first two lines define w_1 as a 64-bit constant 42 (via `constd`) and w_2 as a 64-bit variable (via `var`). The third line defines a 64-bit word w_3, which is the value of the variable w_2 shifted to the left (via `sll`) by 42 bits.

We develop the CryptoLEC tool to convert CryptoLine models to the BTOR format. From formal models in the BTOR format, CryptoLEC then employs the Boolector tool to transform them to the AIGER format [8,9]. The AIGER format is based on AND and NOT gates. Models in the AIGER format are just circuits. Particularly, arithmetic functions such as addition are transformed to AND and NOT gates. Most importantly, formal models in the AIGER format are in functional encoding.

3.3 Logic Equivalence Checking

After reference and target models are transformed to the AIGER format, we use the abc tool to verify whether the two models are equivalent [26]. abc is an open-source formal verification tool for circuits with sophisticated heuristics for logic equivalence checking. Let `ref.aig` and `tgt.aig` be the circuits derived from the CryptoLine reference and implementation models respectively. The abc command `cec ref.aig tgt.aig` automatically checks if the two circuits are logic equivalent. In logic equivalence checking, identifying potentially equivalent subcircuits is indispensable for its effectiveness. Lots of heuristics have been developed to equivalent subcircuits identification through, say, simulation. Unlike [31,23,16], we exploit existing heuristics in abc rather than reinvent wheels.

Another advantage of logic equivalence checking is to verify equivalence at the gate level. This is particularly favorable to verifying block function implementations. To disperse content information, block functions perform bitwise operations. Block functions are thus essentially logic circuits. Logic equivalence checking is hence most suitable for the task.

Recall that logic equivalence checking is a formal verification technique and hence does not use any test vector. Conventional techniques check equivalence on a multitude of test vectors. Untested vectors can induce false positive or negative results. Logic equivalence checking on the other hand verifies whether the two input logic circuits represent the same block function mathematically. No false positive nor negative result can occur.

For illustration, consider the 32-bit armv4 implementation of the Keccak-p[1600, 24] block function from OpenSSL (Section 4.3). Since the block function computes in 64-bit words, the 32-bit implementation has to represent a 64-bit word in two 32-bit words on armv4. The bit interleaving representation suggests that the Keccak-p[1600, 24] block function in fact computes in bits, not words. To prove functional equivalence between the 32-bit armv4 implementation and the block function in 64-bit words, tedious conversions between representations would be unavoidable. It is thus more natural to do proofs in bits. Bit-level proofs may be infeasible for humans, but this is where logic equivalence checking excels.

4 Block Functions

Inputs to hash functions or stream ciphers are typically much larger than a block, so block functions almost surely dominate their computations. In our case studies, we focus on verifying implementations of block functions in the ChaCha20

stream cipher [4], plus the SHA-256 [19] and SHA-3 [18] hash functions (which are NIST or National Institute of Standards and Technology standards [19,18], the latter block function is also used for post-quantum cryptosystems [11]). The ChaCha20 stream cipher is used in web browsers [29,25]. We describe the block functions and optimization tricks applied in various assembly implementations, showing difficulties in equivalence checking.

4.1 ChaCha20

ChaCha20 is a stream cipher [4]. Together with the message authentication code Poly1305 [5], it has been adopted by Google and OpenSSH [29,25].

Description The ChaCha20 block function inputs 16×32-bit words and outputs 16×32-bit words in 20 rounds, each invoking QR (Algorithm 1) 4 times.

Algorithm 1 The QR Function

Require: a, b, c, d : 32-bit word
  function $\mathrm{QR}(a, b, c, d)$
    $a \leftarrow a \boxplus b;\ d \leftarrow \mathrm{ror}(d \oplus a, 16);\ c \leftarrow c \boxplus d;$
    $b \leftarrow \mathrm{ror}(b \oplus c, 12); a \leftarrow a \boxplus b;\ d \leftarrow \mathrm{ror}(d \oplus a, 8);$
    $c \leftarrow c \boxplus d;\ b \leftarrow \mathrm{ror}(b \oplus c, 7).$
  end function

ChaCha20 first copies the input array H to the working array X. Then four QRs each round are invoked on varying (according to even and odd rounds) subsets of X. After 20 rounds, the sum of H and X is the output array H'.

Implementations The ChaCha20 designer released [6] a reference C implementation. The OpenSSL project also provides a ChaCha20 implementation in C, which serves as its reference for the (speed-optimized) ssse3, avx512vl, armv4, and aarch64 [37] assembly implementations. It uses a macro `QUARTERROUND` implementing QR and writing the 20 rounds as 10 double-rounds.

The OpenSSL ssse3 implementations of the ChaCha20 block function compute on 32-bit words which fits four in a 128-bit register.

The OpenSSL avx512vl implementation computes more than one (2, 8, or 16 according to message size) block at the same time. The 2-block implementation uses AVX2 instructions, with two working arrays in four 256-bit registers. Larger avx512vl implementations load 8 and 16 independent working sets into 16× 256- and 512-bit registers respectively, each doing a ChaCha20 block.

4.2 SHA-256

The SHA-2 family (published in 2001 and revised in [19]), defines six hash functions. We focus on the 512-bit block function for SHA-256.

Description The SHA-256 block function generates a 256-bit output digest from a 256-bit input digest represented as 8 words $H[0\ldots,7]$, and a 512-bit message block in 16 words $M[0\ldots,15]$. The main logic functions in SHA-256 are $\mathrm{Ch}(x,y,z) = (x \wedge y) \oplus (\overline{x} \wedge z)$ and $\mathrm{Maj}(x,y,z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$ on 32-bit words x, y, and z, as well as functions $\Sigma_0(x)$, $\Sigma_1(x)$, $\sigma_0(x)$, $\sigma_1(x)$ each formed of the `xor` of three rotations of x.

The SHA-256 block function repeats the round function (Algorithm 2) 64 times. The latter takes as input 8 words $a, b, \ldots, h$, a 32-bit round key w generated from a nonlinear feedback shift register, and the round index r.

Algorithm 2 The SHA-256 Round Function

Require: $a, b, \ldots, h$: 32-bit word (initialized to digest H)
Require: w : 32-bit word
Require: r : round index $0 \leq r < 64$
  function SHA256_ROUND$(a, b, c, d, e, f, g, h, w, r)$
    $T_1 \leftarrow h \boxplus \Sigma_1(e) \boxplus \mathrm{Ch}(e, f, g) \boxplus K_r \boxplus w$
    $T_2 \leftarrow \Sigma_0(a) \boxplus \mathrm{Maj}(a, b, c)$
    $(a', b', c', d', e', f', g', h') \leftarrow (T_1 \boxplus T_2, a, b, c, d \boxplus T_1, e, f, g)$
    return $(a', b', c', d', e', f', g', h')$
  end function

Implementations The SHA-256 block function, unlike ChaCha20, does not have an official reference implementation from its designers (the NSA). Most cryptographic libraries however provide their own reference C implementations.

OpenSSL's 64-bit C implementation is a reference implementing Algorithm 2 (the SHA-256 round function) as a macro — by rotating variables, the assignments at the end of Algorithm 2 are skipped.

OpenSSL's shaext implementation uses new instructions:

`sha256msg1` and `sha256msg2` do 4 σ_0's and σ_1's in parallel.
`sha256rnds2` does two rounds of Algorithm 2, with two 128-bit inputs holding digest words (a, b, e, f) and (c, d, g, h) and `xmm0` implicitly holding precomputed $w_r + K_r$ and $w_{r+1} + K_{r+1}$. It returns the new (a, b, e, f), with the original (a, b, e, f) becoming the new (c, d, g, h).

OpenSSL's two avx2 implementations both use 256-bit avx2 instructions to compute SHA-256(H, M). The multi-block implementation divides each avx2 register into 8×32-bit data paths, facilitating eight simultaneous independent SHA-256(H, M) computations. The single block avx2 implementation uses the avx2 registers for round keys w and the x86_64 general-purpose registers to compute Algorithm 2, with paths of computation interleaved for performance. Both single- and multi-block versions replace $\mathrm{Maj}(x, y, z) = \mathrm{Ch}(x \oplus y, z, y)$ to compute the former in 4 instructions.

OpenSSL's aarch64 implementation computes Maj in 4 instructions $((y \oplus z) \wedge (x \oplus y)) \oplus y = \mathrm{Maj}(x, y, z)$. The aarch64 architecture supports the NEON extension with 128-bit vector registers. Similar to avx2, it computes the w_r's using NEON instructions and the round function (Algorithm 2) using scalar instructions in interleaved fashion.

4.3 SHA-3

The SHA-3 family is based on the Keccak algorithm selected from the SHA-3 Cryptographic Hash Algorithm Competition in 2012 [18]. It defines four hash functions SHA3-224, SHA3-256, SHA3-384, and SHA3-512 with 224-, 256-, 384-, and 512-bit digests respectively. Despite of variances in digest sizes, all SHA-3 hash functions employ the Keccak-p[1600, 24] block function.

Implementations The Keccak team releases the official 64-bit C and multi-block ssse3, avx2, avx512vl implementations for the Keccak-p[1600, 24] block function in the eXtended Keccak Code Package (XKCP) [36]. The OpenSSL project also provides Keccak-p[1600, 24] implementations in C, x86_64, avx2, avx512vl, armv4, and aarch64 assembly [37].

The XKCP 64-bit reference C implementation follows [18] almost verbatim. The `KeccakP1600Round` C function implements Keccak-p[1600, 24] and uses `KeccakP1600_Permute_24rounds` which calls 24 times `KeccakP1600Round`.

The OpenSSL x86_64 implementation does not quite follow [18], some lanes are stored complemented to reduce NOT operations.

The OpenSSL avx2 implementation uses avx2 extension instructions.

The OpenSSL avx512vl implementation uses the avx512vl table look-up instruction `vpternlogq` to compute XOR of three bits [7].

The OpenSSL armv4 implementation differs because armv4 is a 32-bit architecture while Keccak-p[1600, 24] however computes in 64-bit lanes. To implement 64-bit bit rotations on 32-bit armv4 architecture, OpenSSL armv4 implementation uses the bit interleaving representation [7].

5 Block Function Models

In order to verify implementations of block functions (Section 4), we build reference and target models for block functions and their implementations respectively (Section 3). Once the models are constructed, the equivalence checking between two of them is carried out by CryptoLEC automatically. We explain how these formal models are constructed.

5.1 Reference Models

Reference models serve as specifications for block functions. Instead of writing reference models ourselves, we obtain reference models by translating reference C implementations for block functions.

ChaCha20 block function The ChaCha20 designer provides a reference C implementation for the ChaCha20 block function [6]. We translate the function `salsa20_wordtobyte` in `chacha.c` to a CryptoLine model via llvm2CryptoLine. We use the `pragma` directive to force Clang to unroll the loop automatically. All 16 input words including keys are modeled by input model variables. Our verification hence is valid for all keys.

SHA-256 block function The C implementation `sha256.c` from OpenSSL is used to construct the reference model for the SHA-256 block function. Similar to ChaCha20, the `pragma` directive is needed to unroll the loop for the last 48 rounds before using the llvm2CryptoLine tool.

Keccak-p[1600, 24] block function To construct the reference model for the Keccak-p[1600, 24] block function, we take the 64-bit C reference implementation `KeccakP-1600-reference.c` from the XKCP [36] project. llvm2CryptoLine is used to obtain a CryptoLine function for the round function implementation `KeccakP1600Round`. The CryptoLine function is then invoked 24 times in our reference model for the Keccak-p[1600, 24] block function.

5.2 Target Models

Our target models are obtained by translating execution traces of assembly implementations. Specific inputs in the traces are generalized to input model variables. Most assembly instructions are translated to CryptoLine instructions easily. Two special instructions however deserve explanations.

The shaext `sha256rnds2` instruction The instruction repeats the SHA-256 round function twice on the digest values loaded in 128-bit registers (Algorithm 2). The CryptoLine language does not support such complicated instructions. A CryptoLine model based on the Intel manual [20] for `sha256rnds2` is constructed.

In our model, a 128-bit register is represented by four 32-bit CryptoLine variables. The Ch, Maj, Σ_0, and Σ_1 functions are specified by CryptoLine functions. For instance, the following CryptoLine function specifies Ch:

```
proc Ch(uint32 x, uint32 y, uint32 z, uint32 o)=
{ true && true }
and xy@uint32 x y;
not nx@uint32 x;
and nxz@uint32 nx z;
xor o@uint32 xy nxz;
{ true && true }
```

The `Ch` function has three unsigned 32-bit input arguments `x`, `y`, `z`, and an unsigned 32-bit output argument `o`. It computes the bitwise AND of `x` and `y`, and stores the result in `xy`. Similarly, the bitwise AND of `z` and the bitwise NOT of `x` is written to `nxz`. The output argument `o` is the bitwise XOR of `xy` and `nxz`. Hence $\texttt{o} = \texttt{xy} \oplus \texttt{nxz} = (\texttt{x} \wedge \texttt{y}) \oplus (\overline{\texttt{x}} \wedge \texttt{z}) = \mathrm{Ch}(\texttt{x}, \texttt{y}, \texttt{z})$ and other functions are defined similarly. Our model for `sha256rnds2` is built upon these auxiliary functions.

The avx512vl `vpternlogq` *instruction* Recall that the `vpternlogq` instruction takes three 256-bit registers as 256 indices and an 8-bit constant as a table. It computes 256 bits by looking up the table with indices. CryptoLine does not have such an instruction. We again write a CryptoLine model for `vpternlogq`.

Our model first splits each 256-bit register into 256 bit variables. 256 triples of indices are obtained. Let $(\texttt{x}, \texttt{y}, \texttt{z})$ be one of the triples. We want to find the bit in the 8-bit constant table T with the index $0b\texttt{xyz}$. Observe that the bit variable `x` is 1 if and only if the index $0b\texttt{xyz}$ is 4, 5, 6, or 7. The output bit must be among the most significant four bits of T. Similarly, the output must be among the least significant four bits of T if `x` is 0. We use the CryptoLine conditional assignment to obtain the mask.

```
cmov mask_x x 0xf0@uint8 0x0f@uint8;
```

The 8-bit unsigned variable `mask_x` is `0xf0` if `x` is 1; it is `0x0f` otherwise. Likewise, the bit variable `y` is 1 if and only if the index $0b\texttt{xyz}$ is 2, 3, 6, 7; `z` is 1 if and only if the index $0b\texttt{xyz}$ is 1, 3, 5, 7. Define the following masks:

```
cmov mask_y y 0xcc@uint8 0x33@uint8;
cmov mask_z z 0xaa@uint8 0x55@uint8;
and mask_xy@uint8 mask_x mask_y;
and mask_xyz@uint8 mask_xy mask_z;
```

The 8-bit mask `mask_xyz` is the bitwise AND of `mask_x`, `mask_y`, and `mask_z`. Note that all bits in `mask_xyz` are 0 except the bit with the index $0b\texttt{xyz}$. The output bit is the bitwise AND of the table T with `mask_xyz`. The 256-bit result of the `vpternlogq` instruction is obtained by collecting 256 output bits for all triples.

6 Evaluation

To evaluate the effectiveness of our technique, we implement our approach in CryptoLEC and verify 28 target implementations against their reference implementations in experiments. We carry out our experiments on an Ubuntu 22.04.2 Linux server with four 1.5 GHz AMD EPYC 7763 64-core CPUs and 2 TB RAM. CryptoLEC employs Boolector 3.2.2 to generate AIGER files [28,27], and employs abc (commit: 311b9b03) to check logic equivalence between two circuits in the AIGER format [26].

In addition to our technique, a naïve SMT-based technique is tested to verify block function implementations. The CryptoLine tool employs the SMT solver Boolector to check equalities (Section 3.1) [28,30,15]. To verify implementations by SMT solvers, we create another CryptoLine program for each

target implementation. The program contains the models for the target implementation and its reference implementation. We then specify equalities between corresponding variables in the two implementations. For each equality, CryptoLine creates a dedicated thread to verify it with the SMT solver.

Table 1. Experimental Results

Block function	Impl	$Size_{ASM}$	$Size_{CL}$	$Time_{LEC}$	$Time_{SMT}$	Block function	Impl	$Size_{ASM}$	$Size_{CL}$	$Time_{LEC}$	$Time_{SMT}$
ChaCha20 OpenSSL	reference	-	1738	-	-	Keccak-p[1600, 24] OpenSSL	reference	-	671	-	-
	cc	1239	1278	< 1	< 1		cc	5934	6011	59	> 7200
	ssse3	476	1347	< 1	< 1		x86_64	4818	6283	663	> 7200
	avx512vl (2x)	331	1288	< 1	1		avx2	2642	7942	257	> 7200
	avx512vl (8x)	1074	4141	2	18		avx512vl	2024	7774	733	> 7200
	avx512 (16x)	1074	8187	6	74		armv4	8009	10688	111	> 7200
	armv4	1221	1271	< 1	< 1		aarch64	3375	5646	1	> 7200
	aarch64	1022	1064	< 1	< 1	Keccak-p[1600, 24] XKCP	reference	-	671	-	-
SHA-256 OpenSSL	reference	-	3893	-	-		armv7a	3548	7381	71	> 7200
	cc	4185	4274	2992	> 7200		armv7a (2x)	6235	10892	2	6
	shaext	240	731	1907[1]	> 7200		armv8a	3496	4483	115	> 7200
	avx2	2127	3684	1149[1]	> 7200		ssse3 (2x)	3418	8252	112	> 7200
	avx (4x)	3729	9121	1912[1]	> 7200		avx2 (4x)	5204	23417	205	> 7200
	avx2 (8x)	3870	18064	1923[1]	> 7200		avx512 (2x)	2908	7098	567	> 7200
	armv4	2043	2119	> 7200	> 7200		avx512 (4x)	2910	12888	794	> 7200
	aarch64	1779	2716	3591	> 7200		avx512 (8x)	2882	24276	821	> 7200

[1] The shaext and avx2 are verified against the aarch64 implementation; the avx (4x) and avx2 (8x) are against 4 and 8 copies of the aarch64 implementation respectively.

Table 1 shows the experimental results. In the table, the column *Block function* indicates which block function is verified. *Impl* shows names of implementations. $Size_{ASM}$ is the number of assembly instructions in the implementation. $Size_{CL}$ gives the number of CryptoLine instructions in the formal model. $Time_{LEC}$ shows the total time for logic equivalence checking in CryptoLEC; $Time_{SMT}$ gives the total time needed for the naïve SMT-based technique in CryptoLine. They include the time for parsing CryptoLine models, translation to AIGER format, and verification time from abc or Boolector in seconds. It is dominated by the verification time.

For ChaCha20 and Keccak-p[1600, 24], the reference implementations are provided by their respective designers [6,36]. The reference implementation for SHA-256 is the OpenSSL C implementation [37]. We use llvm2CryptoLine to obtain reference models from reference implementations (Section 3.1). Sizes of reference models are indicated in the row *reference* for each block function.

For each block function, we verify a C and several assembly implementations in OpenSSL or XKCP against their reference implementations. The C implementations are from OpenSSL. We compile each C implementation to a binary executable and verify the x86_64 assembly code from the compiled C implementation (marked by row *cc*). The assembly implementations range over different architectures: 32- (*armv4* and *armv7a*) and 64-bit (*aarch64* and *armv8a*) ARM architectures as well as 64-bit Intel *x86_64*, with the optional vector extensions *ssse3*, *avx*, *avx2*, *avx512*, *avx512vl*, and *shaext*. A multi-block implementation computing N block functions is marked with the suffix (Nx).

Table 1 shows that CryptoLEC verifies all but one target implementations against their respective reference implementations in an hour. All ChaCha20

implementations are verified within seconds. All Keccak-p[1600, 24] are verified in 15 minutes. Note that the OpenSSL aarch64 and XKCP armv7a (2x) implementations for Keccak-p[1600, 24] are verified in seconds. Heuristics in logic equivalence checking perform exceptionally well in both cases.

SHA-256 implementations are harder to verify. CryptoLEC fails to verify the OpenSSL SHA-256 armv4 implementation within two hours. It takes about an hour to verify the OpenSSL SHA-256 C and aarch64 implementations. Once the aarch64 implementation is verified, it is used as the reference implementation to verify other implementations. For multi-block implementations, several threads are created. The multi-block avx and avx2 implementations are equivalent to four and eight copies of the OpenSSL SHA-256 aarch64 implementation respectively. The verification of OpenSSL SHA-256 shaext, avx2, multi-block avx, and multi-block avx2 is finished in about 32 minutes.

In comparison, the SMT-based technique only verifies the XKCP Keccak-p[1600, 24] armv7a (2x) block function and the simplest ChaCha20 block function implementations successfully. Among ChaCha20 implementations, its verification time increases significantly for avx512vl multi-block implementations. The conventional technique fails to verify all but one Keccak-p[1600, 24] and SHA-256 implementations within two hours. This is perhaps not surprising. In addition to equality, SMT solvers also support arbitrary logic formulae over inequality. Logic equivalence checking on the other hand is highly optimized for checking equality in circuits. Our technique exploits the more mature technology in logic equivalence checking. Experimental results suggest that the specialized technique is both general and effective in verifying logic equivalence between more than two dozen pairs of reference and assembly implementations.

The SMT-based tool CASMVerify [23] is also tested in our experiments. However, CASMVerify only supports its domain-specific language plus a subset of x86 and ssse3. All the 28 target implementations in Table 1 contain instructions not supported by CASMVerify. Thus, we translate the x86 and ssse3 implementations of ChaCha20 and SHA-256 from the benchmarks of CASMVerify to CryptoLine. For each of ChaCha20 and SHA-256, we use CryptoLEC and CASMVerify to verify the equivalence between its x86 and ssse3 implementations. For ChaCha20, CryptoLEC and CASMVerify take 0.36 seconds and 2088.35 seconds respectively. For SHA-256, CryptoLEC and CASMVerify take 2084.13 seconds and 3707.23 seconds respectively. The results show that CASMVerify does not compete in ChaCha20 and SHA-256.

Another advantage of our automatic technique is to generate witnesses for buggy implementations. To demonstrate, we compare 28 buggy assembly implementations for either ChaCha20 or SHA-256 against their respective correct assembly implementations. These buggy implementations come from the CASMVerify benchmarks with hard to find program mutations [23]. A timeout of 2 hours is set for each buggy implementation. In this experiment, CryptoLEC and CryptoLine successfully report errors for 25 buggy implementations with witnessing inputs respectively in 103.39 seconds and in 106.13 seconds on average. Programmers can then use witnesses to fix programming errors. The other

SMT-based tool CASMVerify on the other hand fails to report errors for 17 cases within the timeout.

7 Conclusions

We apply logic equivalence checking to verifying optimized implementations of block functions. We take reference implementations provided by designers or programmers as specifications of block functions. Optimized implementations are formally verified to generate the same output as the specification on every input. Instead of SMT solvers, our automatic technique employs logic equivalence checking from hardware verification. Experimental results suggest that our technique is simple, general, and scalable. We plan to explore other applications of logic equivalence checking in cryptographic program verification in the future.

Acknowledgments The authors in Academia Sinica are partially funded by National Science and Technology Council grants NSTC110-2221-E-001-008-MY3, NSTC111-2221-E-001-014-MY3, NSTC111-2634-F-002-019, NSTC112-2634-F-002-004, the Sinica Investigator Award AS-IA-109-M01, Taiwan Academic Cybersecurity Center, and the Intel Fast Verified Postquantum Software Project. The authors in Shenzhen University and ISCAS are partially funded by Shenzhen Science and Technology Innovation Commission (JCYJ20210324094202008), the Natural Science Foundation of Guangdong Province (No. 2022A1515011458 and No. 2022A1515010880), and Shanghai 2023 Science and Technology Innovation Action Plan: Special Project for Key Technical Breakthrough of Blockchain (No. 23511100800). The author in National Taiwan University of Science and Technology finished part of the work when he was employed at National Institute of Cyber Security.

References

1. Almeida, J.B., Baritel-Ruet, C., Barbosa, M., Barthe, G., Dupressoir, F., Grégoire, B., Laporte, V., Oliveira, T., Stoughton, A., Strub, P.Y.: Machine-checked proofs for cryptographic standards: Indifferentiability of sponge and secure high-assurance implementations of SHA-3. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. pp. 1607–1622 (2019)
2. Appel, A.W.: Verification of a cryptographic primitive: SHA-256. ACM Transactions on Programming Languages and Systems **37**(2), 7:1–7:31 (April 2015)
3. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Tech. rep., Department of Computer Science, The University of Iowa (2017)
4. Bernstein, D.J.: ChaCha, a variant of Salsa20. In: The State of the Art of Stream Ciphers. vol. 8, pp. 3–5 (2008)
5. Bernstein, D.J.: The Poly1305-AES message-authentication code. In: Gilbert, H., Handschuh, H. (eds.) Fast Software Encryption. LNCS, vol. 3557, pp. 32–49. Springer (2005)
6. Bernstein, D.J.: ChaCha20 reference C version. `https://cr.yp.to/streamciphers/timings/estreambench/submissions/salsa20/chacha20/ref/chacha.c` (2008)

7. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V., Keer, R.V.: Keccak implementation overview (May 2012), http://keccak.noekeon.org/
8. Biere, A.: The AIGER and-inverter graph (AIG) format version 20071012. Tech. rep., Institute for Formal Models and Verification, Johannes Kepler University (2011)
9. Biere, A., Heljanko, K., Wieringa, S.: AIGER 1.9 and beyond. Tech. rep., Institute for Formal Models and Verification, Johannes Kepler University (2011)
10. Bond, B., Hawblitzel, C., Kapritsos, M., Leino, K.R.M., Lorch, J.R., Parno, B., Rane, A., Setty, S.T., Thompson, L.: Vale: Verifying high-performance cryptographic assembly code. In: USENIX Security Symposium. vol. 152 (2017)
11. Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS - Kyber: a CCA-secure module-lattice-based KEM. In: Smith, M., Piessens, F. (eds.) IEEE European Symposium on Security and Privacy. pp. 353–367. IEEE (2018)
12. Cadence: Conformal overview. https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/conformal-overview.html (2023)
13. Chen, R., Liu, J., Shi, X., Tsai, M.H., Wang, B.Y., Yang, B.Y.: llvm2CryptoLine: Verifying arithmetic in cryptographic C programs. In: Chandra, S., Blincoe, K., Tonella, P. (eds.) Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023. pp. 2167–2171. ACM (2023). https://doi.org/10.1145/3611643.3613096
14. Erkök, L., Carlsson, M., Wick, A.: Hardware/software co-verification of cryptographic algorithms using Cryptol. In: 2009 Formal Methods in Computer-Aided Design. pp. 188–191. IEEE (2009)
15. Fu, Y.F., Liu, J., Shi, X., Tsai, M.H., Wang, B.Y., Yang, B.Y.: Signed cryptographic program verification with typed CryptoLine. In: Wang, X., Katz, J. (eds.) 26th ACM SIGSAC Conference on Computer and Communications Security. pp. 1591–1606. ACM, London, UK (November 2019)
16. Hanson, P., Winters, B., Mercer, E., Decker, B.: Verifying the SHA-3 implementation from OpenSSL with the software analysis workbench. In: Model Checking Software: 28th International Symposium, SPIN 2022, Virtual Event, May 21, 2022, Proceedings. pp. 97–113. Springer (2022)
17. Hwang, V., Liu, J., Seiler, G., Shi, X., Tsai, M.H., Wang, B.Y., Yang, B.Y.: Verified NTT multiplications for NISTPQC KEM lattice finalists: Kyber, SABER, and NTRU. IACR Transactions on Cryptographic Hardware and Embedded Systems **2022**, 718–750 (2022)
18. Information Technology Laboratory, N.I.o.S., Technology: Sha-3 standard: Permutation-based hash and extendable-output functions. https://dx.doi.org/10.6028/NIST.FIPS.202 (August 2015), fIPS PUB 202
19. Information Technology Laboratory, National Institute of Standards and Technology: Secure hash standard (SHS). https://dx.doi.org/10.6028/NIST.FIPS.180-4 (August 2015), FIPS PUB 180-4
20. Intel®: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sha-extensions.html (2013)
21. Lattner, C.: LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, University of Illinois at Urbana-Champaign (December 2002)
22. Lavagno, L., Martin, G., Scheffer, L.: Electronic Design Automation for Integrated Circuits Handbook - 2 Volume Set. CRC Press, Inc., USA (2006)

23. Lim, J.P., Nagarakatte, S.: Automatic equivalence checking for assembly implementations of cryptography libraries. In: 2019 IEEE/ACM International Symposium on Code Generation and Optimization. pp. 37–49 (2019)
24. Liu, J., Shi, X., Tsai, M.H., Wang, B.Y., Yang, B.Y.: Verifying arithmetic in cryptographic C programs. In: Lawall, J., Marinov, D. (eds.) 34th IEEE/ACM International Conference on Automated Software Engineering. pp. 552–564. IEEE, San Diego, CA, USA (November 2019)
25. Miller, D.: ChaCha20 and Poly1305 in OpenSSH. http://blog.djm.net.au/2013/11/chacha20-and-poly1305-in-openssh.html (2013)
26. Mishchenko, A.: ABC: System for sequential logic synthesis and formal verification (2023), https://github.com/berkeley-abc/abc.git
27. Niemetz, A., Preiner, M., Wolf, C., Biere, A.: Btor2 , BtorMC and Boolector 3.0. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 10981, pp. 587–595. Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_32, https://doi.org/10.1007/978-3-319-96145-3_32
28. Niemetz, A., Preiner, M., Biere, A.: Boolector 2.0. J. Satisf. Boolean Model. Comput. **9**(1), 53–58 (2014). https://doi.org/10.3233/sat190101, https://doi.org/10.3233/sat190101
29. Nir, Y., Langley, A.: ChaCha20 and Poly1305 for IETF protocols. https://www.rfc-editor.org/rfc/rfc8439.html (June 2018)
30. Polyakov, A., Tsai, M.H., Wang, B.Y., Yang, B.Y.: Verifying arithmetic assembly programs in cryptographic primitives. In: Schewe, S., Zhang, L. (eds.) CONCUR. pp. 4:1–4:16. LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2018)
31. Smith, E.W.: Axe: An automated formal equivalence checking tool for programs. Ph.D. thesis, Stanford University (2011)
32. Swamy, N., Hriţcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.Y., Kohlweiss, M., et al.: Dependent types and multi-monadic effects in F. In: Proceedings of the 43rd annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 256–270 (2016)
33. Synopsys: Formality equivalence checking. https://www.synopsys.com/implementation-and-signoff/signoff/formality-equivalence-checking.html (2023)
34. The CryptoLine Project: https://github.com/fmlab-iis/cryptoline (2023)
35. The GDB developers: GDB: The GNU project debugger. https://sourceware.org/gdb/ (2023)
36. The Keccak Team: The eXtended Keccak Code Package (XKCP). https://github.com/XKCP/XKCP (2014)
37. The OpenSSL Project: The OpenSSL library. https://github.com/openssl/openssl (2023)
38. Zinzindohoué, J.K., Bhargavan, K., Protzenko, J., Beurdouche, B.: HACL*: A verified modern cryptographic library. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 1789–1806 (2017)