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] Chinese Academy of Sciences, China
[4] National Institute of Cyber Security, 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 to verify assembly implementations of cryptographic block functions. Logic equivalence checking is a well-established technique from hardware verification. Using our proposed method, we verify 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 the conventional SMT-based technique in experiments.

Keywords: hash functions, stream ciphers, cryptographic programs, logical equivalence checking, formal verification

1 Introduction

Hash functions are widely used in computer security. Digital signatures and message authentication codes are but some applications of hash functions [4,16]. To digest an arbitrary message, hash functions often divide the message into blocks and compute by message blocks. Such block functions are common in the design of hash functions and stream ciphers. Because block functions dominate the computation time by nature, cryptographic libraries can 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]. They are critical to computer security.

Through careful design and analysis, cryptographic hash functions often enjoy some provable measure of resilience against various attacks (see, for example, [8,12,29]). Their implementations moreover are results of skillful programming and engineering. In fact, implementations are often manually optimized for better performance on different architectures. Such optimizations can increase programming complexity significantly. If any programming mistake occurs, implementations may not compute cryptographic block functions correctly. All security guarantees of hash functions are then voided. Incorrect implementations of block functions therefore can be a devastating threat to computer security.

In this paper, we propose a method to verify functional equivalence of cryptographic block functions and their implementations formally, by taking advantages of a well-established automatic technique. By functional equivalence, we mean that an implementation always generates the same output as the corresponding block function on any input. In contrast to testing, formal verification aims to demonstrate the equivalence of functions and implementations through logical reasoning. Formal verification consequently ensures that an implementation of a block function computes correct results for not only many but all inputs. It is believed that such a strong guarantee is preferred for critical components like block functions.

To check functional equivalence between cryptographic block functions and their implementations, we apply logic equivalence checking. Logic equivalence checking is a well-established automatic formal verification technique to check functional equivalence between circuits [26]. The technique has been developed for several decades and widely applied to verify the correctness of circuit synthesis and optimization. Open-source and commercial tools for logic equivalence checking are available and widely used by the research community and industry [31,15,40].

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 develop a tool called CryptoLEC to realize the method and verify implementations of block functions in the ChaCha20 stream cipher, the SHA-256 and SHA-3 hash functions. Precisely, we take reference C implementations from the designers (ChaCha20 and SHA-3) or C programmers (SHA-256). Optimized assembly implementations in the OpenSSL and XKCP are compared against their reference implementations. ChaCha20 assembly implementations (avx512vl multi-blocks, ssse3, armv4, aarch64) are verified within seconds; SHA-3 implementations (ssse3 multi-blocks, avx2, avx2 multi-blocks, avx512vl, avx512vl multi-blocks, armv4, aarch64) are verified in a quarter hour; and SHA-256 implementations (shaext, avx2, avx multi-blocks, avx2 multi-blocks, avx512vl, aarch64) are verified in an hour. Our technique is sufficiently general and effective to verify two dozen industrial 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 suitable for our purposes. To evaluate our proposed technique, we compare the SMT-based technique in experiments. The SMT-based technique successfully verifies ChaCha20 implementations, but it fails to verify any but one implementation for SHA-256 and SHA-3. Recall that the ChaCha20 block function is considerably simpler than the others. The SMT-based technique is ineffective in our experiments.

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 24 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 by several tools and techniques, including Hacl* [42] and ValeCrypt [13]. They provide F* [39] verified high-level codes (for ChaCha20, SHA-3), and then generate C and assembly implementations respectively. Another notable study focuses on generating efficient assemblies for SHA-3 [1]. It uses EasyCrypt to prove the equivalence between the reference and the efficient Jasmin implementation, which is then used to generate the assembly. 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 [27]. 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 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 [20], and for hardware design of the Skein hash algorithm in VHDL [18]. Those implementations are checked against the hand-written specifications in Cryptol language. Axe is a symbolic execution tool that has been used to verify existing Java implementations of AES [38], 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. $\text{ror}(a, i)$ is the n-bit word obtained by rotating a to the right by i bits. $\text{shr}(a, i)$ 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.

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.

For efficiency, a block function often has implementations for different architectures. We will consider implementations for the 32- and 64-bit ARM architectures *armv4* and *aarch64* respectively. The 64-bit Intel *x86_64* instruction set, 128-bit Supplemental Streaming Single-Instruction-Multiple-Data (SIMD) Extensions 3 (*ssse3*), 128-bit Advanced Vector Extensions (*avx*), 256-bit Advanced Vector Extensions 2 (*avx2*), 512-bit Advanced Vector and Vector Length Extensions (*avx512* and *avx512vl* respectively), and SHA Extensions (*shaext*) are also considered.

3 Methodology

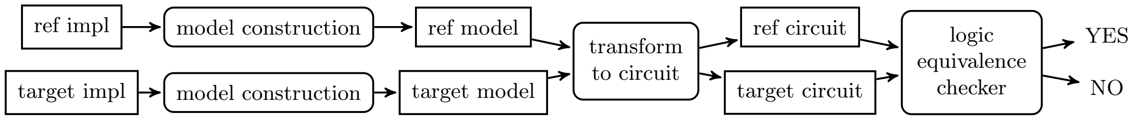

Fig. 1. Methodology Overview: General Framework

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

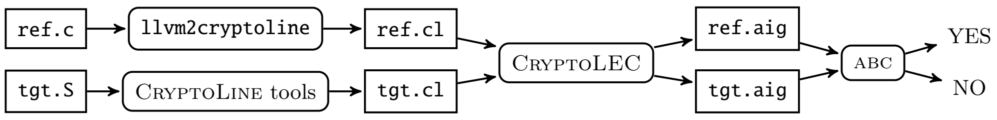

Fig. 2. Methodology Overview: Our Instantiation

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 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.

With the two circuits derived from the reference and target implementations respectively, we invoke a logic equivalence checker for circuits. Logic equivalence checking [26] is a well-developed technique from hardware verification. Scalable open-source and commercial tools are widely available, for instance, [31,15,40]. Our method exploits heuristics in logic equivalence checking for free.

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.

Instantiating the Framework 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.

Specifically, we take assembly programs from OpenSSL [37] and XKCP [41] as target implementations in our case studies. As for reference implementations, we take reference C implementations from block function designers or programmers. Let us call the reference C implementation `ref.c` and the target implementation `tgt.S`. We choose the CryptoLine modeling language [35,19] to define formal models. Thus the tool `llvm2cryptoline` [28] is applied to convert `ref.c` into the reference model `ref.cl`. We also leverage the tools equipped with CryptoLine to convert `tgt.S` into the target model `tgt.cl`. Given two CryptoLine models `ref.cl` and `tgt.cl`, we develop a tool called CryptoLEC that transforms them to circuits and invokes a logic equivalence checker fully automatically. The open-source tool abc [31] is adopted as the underlying logic equivalence checker. The AIGER format [10,11] is used for circuit representation. As a result, our technique successfully verifies two dozen OpenSSL and XKCP assembly implementations for block functions in ChaCha20, SHA-256, and SHA-3 automatically.

We further detail how our method is implemented in the following Sections 3.1–3.3.

3.1 Model Construction

Our formal models are written in the CryptoLine modeling language [35,19,28]. 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` [25]. 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. Otherwise, equivalence checking will fail.

CryptoLine The domain specific language CryptoLine is used to construct formal models [36] in this work. It is the modeling language for the automatic verification tool CryptoLine. The CryptoLine language has modeled cryptographic primitives from elliptic curve [35,19,28] to post-quantum cryptography [21]. 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. Although destination types of arithmetic operations are easy to infer, they are usually meaningless for bitwise operations. CryptoLine consequently requires explicit destination types for bitwise operations. Consider

```
  xor r2@uint32 r1 0xffff@uint32;
```

The type of the destination operand `r2` is unsigned 32-bit word. The CryptoLine type checker moreover requires both source operands to have the same type. It would be a type error if the source operand `r1` were not an unsigned 32-bit word. Another instruction whose destination type needs to be specified is the non-deterministic assignment. The following instruction assigns an unsigned 64-bit number to the variable `r3` non-deterministically:

```
  nondet r3@uint64;
```

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 to automate the construction of formal models for reference C implementations such as the reference implementations of the ChaCha20, SHA-256 and Keccak-p[1600, 24] (employed in SHA-3) block functions [28]. Roughly, the tool takes a reference C implementation, obtains the LLVM intermediate representation of the implementation from the Clang compiler [25], and translates it to a CryptoLine model.

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 [36]. 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` [17]. 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=L0x5..5a80
```

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=L0x5..5a80
```

```
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 is a comment showing 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 derived from their addresses (`L0x555555555a80`, …, `L0x555555555a98`). The code shifts the four 64-bit words in `ymm2` to the left by values in respective memory cells, and writes the results 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 are different from circuits. To apply logic equivalence checking, they are transformed to circuits in a suitable format. We transform CryptoLine models to logic circuits in two steps. A CryptoLine model is converted to the BTOR format [32] first. The Boolector tool is then used to transform formal models in the BTOR format to circuits in the AIGER format [33,10,11].

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, namely, $\{(o, i_0, i_1) | (o, i_0, i_1) = (1,1,1), (0,0,1), (0,1,0), \text{ or } (0,0,0)\}$. 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 [33,32]. Different from the SMT-LIB format, BTOR is designed for functional encoding. In BTOR, constants and variables are declared with identification numbers and bit widths. They will be referred by identification numbers. For example, the following two lines declare a 64-bit constant `42` with the identification number `1` and a 64-bit variable `too` with the identification number `2`:

```
  1 constd 64 42
```

```
2 var 64 too
```

New variables are defined by the identification numbers, operations, bit widths, and arguments. Consider

```
3 sll 64 2 1
```

It defines a 64-bit variable with the identification number `3`. The keyword `sll` denotes the logical left-shift operation. The new variable has the value of the variable `too` (with the identification number `2`) shifted to the left by `42` (with the identification number `1`) bits.

We develop the CryptoLEC tool to convert CryptoLine models to the BTOR format. Except the non-deterministic assignment instruction, CryptoLine instructions are converted to BTOR instructions easily. For non-deterministic assignments, destination variables are converted to fresh BTOR variables of the same bit widths.

From formal models in the BTOR format, CryptoLEC employs the Boolector tool to transform them to the AIGER format [10,11]. 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 [31]. 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` verifies whether the two circuits are logic equivalent automatically. In logic equivalence checking, identifying potentially equivalent subcircuits is an indispensable component for its effectiveness. Different from [38,27,20], we do not reinvent the wheel but take advantages of existing heuristics in abc for free.

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.

To illustrate our points, 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. If one attempts proving 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 prove 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 [5], plus the SHA-256 [22] and SHA-3 [23] hash functions (which are NIST or National Institute of Standards and Technology standards [22,23], the latter block function is also used for post-quantum cryptosystems [14]). The ChaCha20 stream cipher is used in web browsers [34,30]. We describe the block functions and optimization tricks applied in various assembly implementations, showing the hardness of equivalence checking.

4.1 ChaCha20

ChaCha20 is a stream cipher proposed in [5]. Together with the message authentication code Poly1305 [6], it has been adopted by Google and OpenSSH [34,30].

Description The ChaCha20 block function takes 16×32-bit words as inputs and outputs 16×32-bit words in 20 rounds. Four times each round it invokes QR (Algorithm 1), which applies 32-bit additions, XORs, and right-rotations to four variables.

Algorithm 1 The QR Function

```
Require: $a, b, c, d$ : 32-bit word
  function 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 (Algorithm 2) 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 [7] a reference C implementation. The OpenSSL project also provides a ChaCha20 implementation in C, which serves for us as the reference for the (speed-optimized) ssse3, avx512vl, armv4, and aarch64 [37] assembly implementations.

Algorithm 2 The ChaCha20 Block Function

```
Require: $H$ : 32-bit word array of size 16
  function ChaCha20($H$)
      for all $0 \le i < 16$ do
          $X_i \leftarrow H_i$
      end for
      for all $0 \le i < 20$ do
          if $i$ is even then
              QR($X_0, X_4, X_8, X_{12}$); QR($X_1, X_5, X_9, X_{13}$);
              QR($X_2, X_6, X_{10}, X_{14}$); QR($X_3, X_7, X_{11}, X_{15}$).
          else
              QR($X_0, X_5, X_{10}, X_{15}$); QR($X_1, X_6, X_{11}, X_{12}$);
              QR($X_2, X_7, X_8, X_{13}$); QR($X_3, X_4, X_9, X_{14}$).
          end if
      end for
      for all $0 \le i < 16$ do
          $H'_i \leftarrow X_i \boxplus H_i$
      end for
      return $H'$
  end function
```

The designer's reference implementation is almost Algorithm 2 line by line, except with a macro `QUARTERROUND` implementing QR and writing the 20 rounds as 10 double-rounds (avoiding `if`s).

The* OpenSSL *ssse3 and avx512vl implementations of the ChaCha20 block function compute on 32-bit words which fits four in a 128-bit register. The OpenSSL ssse3 implementation hence loads the working array X in four 128-bit `xmm` registers to speed up computation, with the 128-bit `xmm`i register holding $X_{4i}, \ldots, X_{4i+3}$. 4 simultaneous 32-bit operations on `xmm` registers are just 1 ssse3 instruction. E.g., if `xmm0`= $[X_0, X_1, X_2, X_3]$ and `xmm1`= $[X_4, X_5, X_6 X_7]$. Then `xmm0` will become $[X_0 \boxplus X_4, X_1 \boxplus X_5, X_2 \boxplus X_6, X_3 \boxplus X_7]$ after `paddd xmm1, xmm0`. Using SIMD instructions, the OpenSSL ChaCha20 ssse3 implementation computes four `QR` functions in parallel.

The OpenSSL ChaCha20 avx512vl implementation also exploits SIMD — but it computes more than one input (2, 8, or 16 according to message size) block at the same time. The 2-block implementation uses AVX2 instructions (like ssse3 in duplex) and loads two working arrays in four 256-bit registers. Larger avx512vl implementations divide 8 and 16 independent working sets into 16× 256- and 512-bit registers respectively, each computing a ChaCha20 block function.

4.2 SHA-256

The SHA-2 family (published in 2001 and revised in [22]), defines six hash functions: SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, and SHA-512/256 with digest sizes 224, 256, 384, 512, 224, and 256 respectively. SHA-

224 and SHA-256 (which we focus on) compute on 512-bit blocks; the others 1024-bit blocks.

Description The SHA-256 block function generates a 256-bit output digest from a 256-bit input digest and a 512-bit message block, computing only on words (in Section 4.2 always 32 bits). The input digest is represented as 8 words $H[0], \ldots, H[7]$, and each message block M as 16 words $M[0], \ldots, M[15]$.

The following functions are defined for words x, y, and z:

$$\begin{aligned}
\mathrm{Ch}(x,y,z) &= (x \wedge y) \oplus (\overline{x} \wedge z) \\
\mathrm{Maj}(x,y,z) &= (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) \\
\Sigma_0(x) &= \mathrm{ror}(x,2) \oplus \mathrm{ror}(x,13) \oplus \mathrm{ror}(x,22) \\
\Sigma_1(x) &= \mathrm{ror}(x,6) \oplus \mathrm{ror}(x,11) \oplus \mathrm{ror}(x,25) \\
\sigma_0(x) &= \mathrm{ror}(x,7) \oplus \mathrm{ror}(x,18) \oplus \mathrm{shr}(x,3) \\
\sigma_1(x) &= \mathrm{ror}(x,17) \oplus \mathrm{ror}(x,19) \oplus \mathrm{shr}(x,10)
\end{aligned}$$

SHA-256 also defines 64 constants $K_0, K_1, \ldots, K_{63}$ as the initial 32 bits of fractional parts of the cube roots of the first 64 primes. E.g., $K_0 = \left\lfloor 2^{32} \left(\sqrt[3]{2} - 1 \right) \right\rfloor =$ 0x428a2f98. Algorithm 3 shows the round function SHA256_ROUND, called 64 times by the SHA-256 block function (Algorithm 4).

In SHA256_ROUND the inputs are 8 words $a, b, \ldots, h$, a 32-bit working word (akin to a “round key”) w, and a round index r. We apply various 32-bit functions to words $a, b, \ldots, h$, the constant K_r, and the working word w to get temporary variables T_1 and T_2. 8 output words are then obtained by shifting input words and (32-bit) sums with temporary variables.

Algorithm 3 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
```

$M[i]$ in round $i < 16$ is both the working word w_i and used to initialize a non-linear feedback shift register (NLFSR) whose feedback uses σ_0, σ_1, and 32-bit addition (see below). This NLFSR returns w_i for $16 \leq i < 64$. Then each round we update words $a, b, \ldots, h$ using SHA256_ROUND. Finally, the output digest $H = H \oplus (a, b, \ldots, h)$ after 64 iterations.

Algorithm 4 The SHA-256 Block Function

```
Require: $M$ : input 32-bit word array of size 16
Require: $H$ : digest 32-bit word array of size 8
  function SHA256$(H, M)$
      $(a, b, c, d, e, f, g, h) \leftarrow (H[0], H[1], H[2], H[3],$
                                          $H[4], H[5], H[6], H[7])$
      for $r \leftarrow 0$ to 64 do
          if $r < 16$ then                              ▷ compute the working word
              $w_r \leftarrow M[r]$
          else
              $w_r \leftarrow \sigma_1(w_{r-2}) \boxplus w_{r-7} \boxplus \sigma_0(w_{r-15}) \boxplus w_{r-16}$
          end if
          $(a, b, c, d, e, f, g, h) \leftarrow$
              SHA256_ROUND$(a, b, c, d, e, f, g, h, w_r, r)$
      end for
      $H'[0] \leftarrow a \boxplus H[0], H'[1] \leftarrow b \boxplus H[1], H'[2] \leftarrow c \boxplus H[2],$
      $H'[3] \leftarrow d \boxplus H[3], H'[4] \leftarrow e \boxplus H[4], H'[5] \leftarrow f \boxplus H[5],$
      $H'[6] \leftarrow g \boxplus H[6], H'[7] \leftarrow h \boxplus H[7].$
      return $H'$
  end function
```

Implementations The block function SHA-256(H, M), 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 should be the baseline for comparisons of its optimized-for-efficiency assembly implementations (Intel x86 shaext, avx2, armv4, and aarch64). The reference C implementation follows Algorithm 4 except for implementing Algorithm 3 (the SHA-256 round function) as a macro — by rotating variables, the assignments at the end of Algorithm 3 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 3, 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 is more complicated. The working word w_r in Algorithm 4 needs to be computed for rounds $r \geq 16$ from the input block $M[\bullet]$. Therefore the avx2 registers are used to compute the $w_\bullet$ and the x86_64

GPRs (general-purpose registers) to compute Algorithm 3, with the 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 (where the naïve way needs 5) using

$$((y \oplus z) \wedge (x \oplus y)) \oplus y = \mathrm{Maj}(x, y, z)$$

Also $\mathrm{Ch}(x, y, z) = (x \wedge y) \vee (\overline{x} \wedge z)$.

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 3) 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 [23]. 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.

Description The Keccak-p[1600, 24] block function receives 1600 input bits and generates 1600 output bits in 24 rounds. The 1600 bits are represented by a 3-dimensional bit array $A[x, y, z]$ with $0 \leq x, y < 5$ and $0 \leq z < 64$. For fixed x, y, z, we call the 5-bit word $A[\bullet, y, z]$ a *row*, the 5-bit word $A[x, \bullet, z]$ a *column*, the 64-bit word $A[x, y, \bullet]$ a *lane*, the 5×64 bit array $A[\bullet, y, \bullet]$ a *plane*, and the 5×5 bit array $A[\bullet, \bullet, z]$ a *slice*.

Each round of Keccak-p[1600, 24] consists of five functions θ, ρ, π, χ, and ι. Algorithm 5 shows the $\theta(A)$ function. It first computes the parity of each column. Every bit of a column is the XOR of itself and two nearby columns.

The $\rho(A)$ function is shown in Algorithm 6. It simply rotates every lane other than $A[0, 0, \bullet]$ by different amounts.

The $\pi(A)$ function moves around entire lanes, alternatively, it permutes bits on each slice (Algorithm 7). Observe that the lane at $((x + 3y) \bmod 5, x)$ is moved to (x, y) which is itself moved to $(y, 2x + 3y \bmod 5)$. In other words, $\pi(A)$ is invertible. The lane $A[0, 0, \bullet]$ remains unchanged after permutation.

For each row, $\chi(A)$ XORs into each bit the conjunction of the bit two positions down and the complement of the next bit (Algorithm 8).

At round r, the $\iota(A, r)$ function modifies the 64-bit lane $A[0, 0, \bullet]$ by the round constant RC (Algorithm 10). The constant RC of round r is a sparse 64-bit bit array with only 7 possible non-zero bit positions $RC[2^i - 1] = \textsc{rc}(7r + i)$, $i = 0 \cdots 6$. Where $\textsc{rc}(t)$ is a linear feedback shift register (Algorithm 9) with the polynomial being $1 + x^4 + x^5 + x^6 + x^8$. Subsequently, at most 7 bits may change at the lane $A[0, 0, \bullet]$ by the $\iota(A, r)$ function. Note that the round constant RC only depends on the round number r. It can be pre-computed for efficiency.

Algorithm 5 The $\theta(A)$ Function

```
Require: $A$ : bit array of size $5 \times 5 \times 64$
  function $\theta(A)$
      for all $0 \leq x < 5, 0 \leq z < 64$ do
          $C[x,z] \leftarrow A[x,0,z] \oplus A[x,1,z] \oplus A[x,2,z] \oplus$
                      $A[x,3,z] \oplus A[x,4,z]$
      end for
      for all $0 \leq x < 5, 0 \leq z < 64$ do
          $D[x,z] \leftarrow C[(x-1) \bmod 5, z] \oplus$
                      $C[(x+1) \bmod 5, (z-1) \bmod 64]$
      end for
      for all $0 \leq x < 5, 0 \leq y < 5, 0 \leq z < 64$ do
          $A'[x,y,z] \leftarrow A[x,y,z] \oplus D[x,z]$
      end for
      return $A'$
  end function
```

Algorithm 6 The $\rho(A)$ Function

```
Require: $A$ : bit array of size $5 \times 5 \times 64$
  function $\rho(A)$
      for all $0 \leq z < 64$ do
          $A'[0,0,z] \leftarrow A[0,0,z]$
      end for
      $(x,y) \leftarrow (1,0)$
      for $t \leftarrow 0$ to $23$ do
          for all $0 \leq z < 64$ do
              $A'[x,y,z] \leftarrow$
                          $A[x,y,(z-(t+1)(t+2)/2) \bmod 64]$
              $(x,y) \leftarrow (y, 2x+3y) \bmod 5$
          end for
      end for
      return $A'$
  end function
```

Algorithm 7 The $\pi(A)$ Function

```
Require: $A$ : bit array of size $5 \times 5 \times 64$
  function $\pi(A)$
      for all $0 \leq x < 5, 0 \leq y < 5, 0 \leq z < 64$ do
          $A'[x,y,z] \leftarrow A[(x+3y) \bmod 5, x, z]$
      end for
      return $A'$
  end function
```

Algorithm 8 The $\chi(A)$ Function

```
Require: $A$ : bit array of size $5 \times 5 \times 64$
  function $\chi(A)$
      for all $0 \leq x < 5, 0 \leq y < 5, 0 \leq z < 64$ do
          $A'[x, y, z] \leftarrow A[x, y, z] \oplus$
            $(\underline{A[(x+1) \bmod 5, y, z]} \wedge A[(x+2) \bmod 5, y, z])$
      end for
      return $A'$
  end function
```

Algorithm 9 The $\textsc{rc}(t)$ Function (LFSR)

```
Require: $t$ : integer
  function $\textsc{rc}(t)$
      $(R_0, R_1, R_2, R_3, R_4, R_5, R_6, R_7) \leftarrow (1, 0, 0, 0, 0, 0, 0, 0)$
      for $i \leftarrow 1$ to $t \bmod 255$ do
          $(R_0, R_1, R_2, R_3, R_4, R_5, R_6, R_7) \leftarrow$
              $(R_7, R_0, R_1, R_2, R_3 \oplus R_7, R_4 \oplus R_7, R_5 \oplus R_7, R_6)$
      end for
      return $R_0$
  end function
```

Finally, Algorithm 11 gives the Keccak-p[1600, 24] block function. It is defined by 24 repetitions of $\iota \circ \chi \circ \pi \circ \rho \circ \theta$.

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) [41]. The OpenSSL project also provides Keccak-p[1600, 24] implementations in C, x86_64, avx2, avx512vl, armv4, and aarch64 assembly [37].

Algorithm 10 The $\iota(A, r)$ Function

```
Require: $A$ : bit array of size $5 \times 5 \times 64$
Require: $r$ : round index
  function $\iota(A, r)$
      $A' = A$
      for $j \leftarrow 0$ to $6$ do
          $A'[0, 0, 2^j - 1] \leftarrow A'[0, 0, 2^j - 1] \oplus \textsc{rc}(j + 7r)$
      end for
      return $A'$
  end function
```

The* XKCP *64-bit reference C implementation follows Algorithm 11 almost verbatim. Concretely, the C functions `theta`, `rho`, `pi`, `chi`, and `iota` implement the functions θ, ρ, π, χ, and ι respectively. The $5 \times 5 \times 64$ bit array A

Algorithm 11 The Keccak-p[1600, 24] Block Function

```
Require: $A$ : bit array of size $5 \times 5 \times 64$
  function Keccak-p[1600, 24]($A$)
      $A' \leftarrow A$
      for $r \leftarrow 0$ to 23 do
          $A' \leftarrow \iota(\chi(\pi(\rho(\theta(A')))), r)$
      end for
      return $A'$
  end function
```

moreover is represented by a 64-bit word array of size 25 in row-major order. $\rho(A)$ is implemented by 64-bit rotations with various offsets `(t+1)*(t+2)/2 % 64`.

The rotation offsets in the $\rho(A)$ function and the round constants in $\iota(A, r)$, instead of being computed on the fly, are pre-computed and stored in arrays to be retrieved and used by the C functions `rho` and `iota`.

The `KeccakP1600Round` C function implements Keccak-p[1600, 24] and uses `KeccakP1600_Permute_24rounds` which calls 24 times `KeccakP1600Round`, which in term just calls the functions `theta`, `rho`, `pi`, `chi`, and `iota` in sequence.

The* OpenSSL *x86_ 64 implementation computes the Keccak-p[1600, 24] block function in 13×64-bit registers. The 10 bit arrays $C[\bullet, \bullet]$ and $D[\bullet, \bullet]$ in $\theta(A)$ (Algorithm 5) are kept in 10 registers. One register is a pointer to the array of pre-computed round constants in $\iota(A, r)$ (Algorithm 10). There are two scratch registers. To reduce memory access, this Keccak-p[1600, 24] implementation does not quite follow Algorithm 11 — the implementation combines θ, ρ, π, χ, and ι together, rearranges instructions, and uses registers carefully. Each round reads each lane from memory at most twice to minimize latency. A trick is also employed where some lanes are stored complemented to reduce NOT operations. For instance, a plane $A'[\bullet, y, \bullet]$ can be computed by the following equations in the $\chi(A)$ function (Algorithm 8):

$$\begin{aligned}
A'[0, y, \bullet] &= A[0, y, \bullet] \oplus (\overline{A[1, y, \bullet]} \land A[2, y, \bullet]) \\
A'[1, y, \bullet] &= A[1, y, \bullet] \oplus (\overline{A[2, y, \bullet]} \land A[3, y, \bullet]) \\
A'[2, y, \bullet] &= A[2, y, \bullet] \oplus (\overline{A[3, y, \bullet]} \land A[4, y, \bullet]) \\
A'[3, y, \bullet] &= A[3, y, \bullet] \oplus (\overline{A[4, y, \bullet]} \land A[0, y, \bullet]) \\
A'[4, y, \bullet] &= A[4, y, \bullet] \oplus (\overline{A[0, y, \bullet]} \land A[1, y, \bullet])
\end{aligned}$$

Suppose we store the input lanes $A[2, y, \bullet]$ and $A[4, y, \bullet]$ by their complements $\overline{A[2, y, \bullet]}$ and $\overline{A[4, y, \bullet]}$ respectively. By De Morgan's law, the plane $A'[\bullet, y, \bullet]$ can be computed by

$$\begin{aligned}
\overline{A'[0, y, \bullet]} &= A[0, y, \bullet] \oplus (A[1, y, \bullet] \lor \overline{A[2, y, \bullet]}) \\
A'[1, y, \bullet] &= A[1, y, \bullet] \oplus (\overline{A[2, y, \bullet]} \land A[3, y, \bullet]) \\
\overline{A'[2, y, \bullet]} &= A[2, y, \bullet] \oplus (A[3, y, \bullet] \lor \overline{A[4, y, \bullet]}) \\
A'[3, y, \bullet] &= A[3, y, \bullet] \oplus (\overline{A[4, y, \bullet]} \land A[0, y, \bullet]) \\
A'[4, y, \bullet] &= \overline{A[4, y, \bullet]} \oplus (A[0, y, \bullet] \lor \overline{A[1, y, \bullet]}).
\end{aligned}$$

Note that the output lanes $A'[0, y, \bullet]$ and $A'[2, y, \bullet]$ are now represented by their complements $\overline{A'[0, y, \bullet]}$ and $\overline{A'[2, y, \bullet]}$ respectively, and 80% of the NOT operations are canceled if the complementing transform is propagated throughout all functions [9].

The* OpenSSL *avx2 implementation uses avx2 extension instructions. For example the `vpxor` instruction computes bitwise XOR over two 256-bit registers; and the `vpsrlq` instruction computes logical right rotation of four 64-bit words in a register. The avx2 extensions moreover provide various instructions for rearranging words in 256-bit registers.

$A[0,0,\bullet]$	$A[0,0,\bullet]$	$A[0,0,\bullet]$	$A[0,0,\bullet]$
$A[4,0,\bullet]$	$A[3,0,\bullet]$	$A[2,0,\bullet]$	$A[1,0,\bullet]$
$A[0,3,\bullet]$	$A[0,1,\bullet]$	$A[0,4,\bullet]$	$A[0,2,\bullet]$
$A[4,2,\bullet]$	$A[3,4,\bullet]$	$A[2,1,\bullet]$	$A[1,3,\bullet]$
$A[4,3,\bullet]$	$A[3,1,\bullet]$	$A[2,4,\bullet]$	$A[1,2,\bullet]$
$A[4,1,\bullet]$	$A[3,2,\bullet]$	$A[2,3,\bullet]$	$A[1,4,\bullet]$
$A[4,4,\bullet]$	$A[3,3,\bullet]$	$A[2,2,\bullet]$	$A[1,1,\bullet]$

Currently, the OpenSSL Keccak-p[1600, 24] avx2 implementation uses the lane arrangement above. The avx2 implementation loads four 64-bit lanes in a 256-bit register. Seven 256-bit registers suffice to load 25 64-bit lanes in $A[\bullet, \bullet, \bullet]$.

Observe first that four copies of the lane $A[0, 0, \bullet]$ are loaded in the first register. This is because $A[0, 0, \bullet]$ is not permuted by the $\rho(A)$ nor $\pi(A)$ functions (Algorithms 6 and 7 respectively). It therefore is treated specially. The remaining 24 64-bit lanes are distributed in six 256-bit registers. Secondly, the arrangement seems peculiar. It is indeed designed for the $\pi(A)$ function (Algorithm 7). Consider the content of the fourth register: $A[4, 2, \bullet], A[3, 4, \bullet], A[2, 1, \bullet], A[1, 3, \bullet]$. In the $\pi(A)$ function, they are assigned to $A[4 + 3 \cdot 2 \bmod 5, 4, \bullet], A[3 + 3 \cdot 4 \bmod 5, 3, \bullet], A[2 + 3 \cdot 1 \bmod 5, 2, \bullet], A[1 + 3 \cdot 3 \bmod 5, 1, \bullet]$ or $A[0, 4, \bullet]$, $A[0, 3, \bullet]$, $A[0, 2, \bullet]$, $A[0, 1, \bullet]$ respectively. These lanes are precisely the content of the third register after permutation. In other words, the $\pi(A)$ function is permutations on register contents. It is easily implemented by avx2 word arrangement instructions.

The* OpenSSL *avx512vl implementation uses the same lane arrangement as the OpenSSL avx2 implementation, and further uses the avx512vl table look-up instruction `vpternlogq` to compute XOR of three bits [9]. The instruction takes three 256-bit registers and an 8-bit constant as inputs. The three 256-bit registers represent 256 triple of bits. Each bit triple in turn represents an integer from 0 to 7. A triple thus serves as a read index into the 8-bit constant array. Consider the 8-bit constant $0x96$. The following table shows the outputs for all bit triples:

$0bxyz$	7 6 5 4 3 2 1 0
x	1 1 1 1 0 0 0 0
y	1 1 0 0 1 1 0 0
z	1 0 1 0 1 0 1 0
out	1 0 0 1 0 1 1 0 $= 0x96$

Note that the output bit is the XOR of the bit triple. The `vpternlogq` instruction therefore can compute 256 XORs of three bits simultaneously.

The* OpenSSL *armv4 implementation differs because armv4 is a 32-bit architecture while Keccak-p[1600, 24] however computes in 64-bit lanes. A naïve 64-bit word represented in two 32-bit registers would lead to expensive rotations, needed for $\theta(A)$ and $\rho(A)$ (Algorithms 5–6).

To implement 64-bit bit rotations on 32-bit armv4 architecture, OpenSSL armv4 implementation uses the bit interleaving representation [9]. A 64-bit word $w = 0bb_{63}b_{62}\cdots b_0$ is represented by two 32-bit words $w_0 = 0bb_{62}b_{60}\cdots\ b_{2i}\cdots b_0$ and $w_1 = 0bb_{63}b_{61}\cdots\ b_{2i+1}\cdots b_1$. The 32-bit word w_0 consists of bits with even indices and w_1 of bits with odd indices (written $w = [w_0, w_1]$). With the bit interleaving representation, bit rotations can be implemented efficiently. Consider rotating w to right by two bits. We have

$$\begin{aligned}\mathrm{ror}(w, 2) &= 0bb_1b_0b_{63}b_{62}\cdots b_5b_4b_3b_2\\ &= [0bb_0b_{62}\cdots b_4b_2, 0bb_1b_{63}\cdots b_5b_3].\end{aligned}$$

Note that $0bb_0b_{62}\cdots b_4b_2$ and $0bb_1b_{63}\cdots b_5b_3$ are simply w_0 and w_1 rotated to right by one bit respectively. Rotations by odd number of bits are almost as easy. Consider

$$\begin{aligned}\mathrm{ror}(w, 3) &= 0bb_2b_1b_0b_{63}\cdots b_6b_5b_4b_3\\ &= [0bb_1b_{63}\cdots b_5b_3, 0bb_2b_0\cdots b_6b_4].\end{aligned}$$

Even bits of the result are w_1 rotated to right by one bit; odd bits of the result are w_0 rotated to right by two bits. One simply swaps the even and odd bits with proper rotations.

The* XKCP *implementations by the Keccak team [41] include (in addition to a reference C implementation) several Keccak-p[1600, 24] assembly implementations written with C intrinsic functions for ssse3, avx2, and avx512 instructions. Intrinsic functions can be reordered by C compilers. XKCP implementations are thus optimized automatically.

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 [7]. We translate the function `salsa20_wordtobyte` in `chacha.c` to a CryptoLine model using `llvm2cryptoline`. We use the `pragma` directive to force the Clang compiler to unroll the loop automatically.

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 [41] project. The `llvm2cryptoline` tool 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. Most assembly instructions are translated to CryptoLine instructions easily. Two special instructions however deserve explanations.

The shaext `sha256rnds2` *instruction* The instruction performs two rounds of the SHA-256 round function on the digest values loaded in 128-bit registers (Algorithm 3). The CryptoLine language does not support such complicated instructions. A CryptoLine model based on the Intel manual for `sha256rnds2` is constructed [24].

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

$$\mathtt{o} = \mathtt{xy} \oplus \mathtt{nxz} = (\mathtt{x} \wedge \mathtt{y}) \oplus (\overline{\mathtt{x}} \wedge \mathtt{z}) = \mathrm{Ch}(\mathtt{x}, \mathtt{y}, \mathtt{z}).$$

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 $(\mathtt{x}, \mathtt{y}, \mathtt{z})$ be one of the triples. We want to find the bit in the 8-bit constant table T with the index $0b\mathtt{xyz}$. Observe that the bit variable `x` is 1 if and only if the index $0b\mathtt{xyz}$ 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 $0b\mathtt{xyz}$ is 2, 3, 6, 7; `z` is 1 if and only if the index $0b\mathtt{xyz}$ 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 $0b\mathtt{xyz}$. 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 [33,32], and employs abc (commit: 311b9b03) to check logic equivalence between two circuits in the AIGER format [31].

In addition to our technique, a naïve SMT-based technique is tested to verify block function implementations. The CryptoLine tool employs the SMT

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[6]	> 7200		armv8a	3496	4483	115	> 7200
	avx2	2127	3684	1149[6]	> 7200		ssse3 (2x)	3418	8252	112	> 7200
	avx (4x)	3729	9121	1912[6]	> 7200		avx2 (4x)	5204	23417	205	> 7200
	avx2 (8x)	3870	18064	1923[6]	> 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.

solver BOOLECTOR to check equalities (Section 3.1) [33,35,19]. 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 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 (CRYPTOLEC); Time_{SMT} gives the total time needed for the SMT-based technique (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 [7,41]. 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*).

OPENSSL provides several assembly implementations for each block function. Specifically, both 32- and 64-bit ARM architectures have dedicated assembly implementations (rows *armv4* and *aarch64*, respectively). For the Intel family, the row *x86_ 64* indicates the basic 64-bit x86 instruction set (KECCAK-p[1600, 24]). The ssse3 implementation is provided for CHACHA20 (row *ssse3*). The avx2 implementations are found for SHA-256 and KECCAK-p[1600, 24] (row *avx2*); the avx512vl implementations are available for Keccak-p[1600, 24] (row *avx512vl*). The rows *avx512vl (2x)*, *avx512vl (8x)*, and *avx512 (16x)* denote multi-block implementations computing two, eight, and sixteen ChaCha20 block functions in the avx512vl extensions respectively. Similarly, the rows *avx (4x)* and *avx2 (8x)* stand for the multi-block implementations for computing four and eight SHA-256 block functions respectively. Finally, a shaext implementation for SHA-256 is provided in OpenSSL (row *shaext*).

XKCP also offers several assembly implementations for the Keccak-p[1600, 24] block function. In addition to the 64-bit ARM architecture (row *armv8a*), 32-bit ARM with the NEON extensions is provided (row *armv7a*). It moreover has multi-block implementations. Specifically, two copies of the Keccak-p[1600, 24] block function are computed in parallel with the ARM NEON extensions (row *armv7a (2x)*). The ssse3 implementation computes two Keccak-p[1600, 24] block functions simultaneously (row *ssse3 (2x)*). The avx2 implementation computes four copies at the same time (row *avx2 (4x)*). The avx512 implementations can compute two, four, or eight copies in parallel (rows *avx512 (2x), avx512 (4x), avx512 (8x)* respectively).

Table 1 shows that CryptoLEC verifies all but one target implementations against their respective reference implementations in an hour. Among the three block functions, ChaCha20 is the simplest and easiest to verify. All ChaCha20 implementations are verified within seconds. All Keccak-p[1600, 24] are verified in 15 minutes. Note that the OpenSSL Keccak-p[1600, 24] aarch64 and XKCP Keccak-p[1600, 24] armv7a (2x) implementations 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. Experimental results suggest that the specialized technique is both general and effective in verifying logic equivalence between two dozen pairs of reference and assembly implementations.

Another advantage of our automatic technique is to generate witnesses for buggy implementations. To demonstrate, we randomly remove one instruction from the XKCP Keccak-p[1600, 24] armv7a (2x) implementation and compare the modified implementation against the reference C implementation. The underlying logic equivalence checker abc reports the error with a witnessing input in seconds. The SMT-based technique also reports the error with a witness shortly. Programmers can then use witnesses to fix programming errors. Both techniques not only verify block function implementations automatically, they are also debugging tools should errors occur in implementations.

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.

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. Bellare, M., Canetti, R., Krawczyk, H.: Message authentication using hash functions— the HMAC construction. CryptoBytes **1** (1996)
5. Bernstein, D.J.: ChaCha, a variant of Salsa20. In: The State of the Art of Stream Ciphers. vol. 8, pp. 3–5 (2008)
6. 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)
7. Bernstein, D.J.: ChaCha20 reference c version. `https://cr.yp.to/streamciphers/timings/estreambench/submissions/salsa20/chacha20/ref/chacha.c` (2008)
8. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: On the indifferentiability of the sponge construction. In: Smart, N.P. (ed.) Advances in Cryptology - EUROCRYPT 2008, 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Istanbul, Turkey, April 13-17, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4965, pp. 181–197. Springer (2008).

https://doi.org/10.1007/978-3-540-78967-3_11, https://doi.org/10.1007/978-3-540-78967-3_11

9. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V., Keer, R.V.: Keccak implementation overview (May 2012), http://keccak.noekeon.org/
10. Biere, A.: The aiger and-inverter graph (aig) format version 20071012. Tech. rep., Institute for Formal Models and Verification, Johannes Kepler University (2011)
11. Biere, A., Heljanko, K., Wieringa, S.: Aiger 1.9 and beyond. Tech. rep., Institute for Formal Models and Verification, Johannes Kepler University (2011)
12. Biham, E., Shamir, A.: Differential Cryptanalysis of the Data Encryption Standard. Springer (1993)
13. 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)
14. 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)
15. Cadence: Conformal overview. https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/conformal-overview.html (2023)
16. Cramer, R., Shoup, V.: Signature schemes based on the strong RSA assumption. In: ACM conference on Computer and communications security. pp. 46—-51. ACM (November 1999)
17. developers, T.G.: GDB: The GNU project debugger. https://sourceware.org/gdb/ (2023)
18. 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)
19. 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)
20. 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)
21. 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)
22. Information Technology Laboratory, N.I.o.S., Technology: Secure hash standard (shs). https://dx.doi.org/10.6028/NIST.FIPS.180-4 (August 2015), fIPS PUB 180-4
23. 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
24. Intel®: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sha-extensions.html (2013)
25. Lattner, C.: LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, University of Illinois at Urbana-Champaign (December 2002)
26. Lavagno, L., Martin, G., Scheffer, L.: Electronic Design Automation for Integrated Circuits Handbook - 2 Volume Set. CRC Press, Inc., USA (2006)

27. 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)
28. 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)
29. Mella, S., Daemen, J., Assche, G.V.: New techniques for trail bounds and application to differential trails in Keccak. IACR Transactions on Symmetric Cryptology **1**, 329–357 (2017)
30. Miller, D.: ChaCha20 and Poly1305 in OpenSSH. `http://blog.djm.net.au/2013/11/chacha20-and-poly1305-in-openssh.html` (2013)
31. Mishchenko, A.: ABC: System for sequential logic synthesis and formal verification (2023), `https://github.com/berkeley-abc/abc.git`
32. 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`
33. 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`
34. Nir, Y., Langley, A.: ChaCha20 and Poly1305 for IETF protocols. `https://www.rfc-editor.org/rfc/rfc8439.html` (June 2018)
35. 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)
36. CryptoLine Project, T.: `https://github.com/fmlab-iis/cryptoline` (2023)
37. Project, T.O.: The OpenSSL library. `https://github.com/openssl/openssl` (2023)
38. Smith, E.W.: Axe: An automated formal equivalence checking tool for programs. Ph.D. thesis, Stanford University (2011)
39. 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)
40. Synopsys: Formality equivalence checking. `https://www.synopsys.com/implementation-and-signoff/signoff/formality-equivalence-checking.html` (2023)
41. Keccak Team, T.: The extended Keccak code package (xkcp). `https://github.com/XKCP/XKCP` (2014)
42. 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)