PDF hosted at the Radboud Repository of the Radboud University Nijmegen

The following full text is a preprint version which may differ from the publisher's version.

For additional information about this publication click this link.
http://hdl.handle.net/2066/132669

Please be advised that this information was generated on 2019-08-15 and may be subject to change.
Verifying Curve25519 Software

Yu-Fang Chen\textsuperscript{1}, Chang-Hong Hsu\textsuperscript{2}, Hsin-Hung Lin\textsuperscript{3}, Peter Schwabe\textsuperscript{4}, Ming-Hsien Tsai\textsuperscript{1}, Bow-Yaw Wang\textsuperscript{1}, Bo-Yin Yang\textsuperscript{1}, and Shang-Yi Yang\textsuperscript{1} *

\textsuperscript{1} Institute of Information Science
Academia Sinica
128 Section 2 Academia Road, Taipei 115-29, Taiwan
yfc@iis.sinica.edu.tw, mhtsai208@gmail.com, bywang@iis.sinica.edu.tw, by@crypto.tw, ilway25@crypto.tw

\textsuperscript{2} University of Michigan, Ann Arbor, USA
hsuch@umich.edu

\textsuperscript{3} Faculty of Information Science and Electrical Engineering
Kyushu University
744, Motoka, Nishi-ku, Fukuoka-Shi, Fukuoka, 819-0395, Japan
h-lin@ait.kyushu-u.ac.jp

\textsuperscript{4} Radboud University Nijmegen
Digital Security Group
PO Box 9010, 6500GL Nijmegen, The Netherlands
peter@cryptojedi.org

Abstract. This paper presents results on formal verification of high-speed cryptographic software. We consider speed-record-setting hand-optimized assembly software for Curve25519 elliptic-curve key exchange in \cite{11}. Two versions for different microarchitectures are available. We successfully verify the core part of computation, and reproduce a bug in a previously published edition. An SMT solver for the bit-vector theory is used to establish almost all properties. Remaining properties are verified in a proof assistant. We also exploit the compositionality of Hoare logic to address the scalability issue. Essential differences between both versions of the software are discussed from a formal-verification perspective.

1 Introduction

Cryptography is the backbone of modern information security. Implementations of cryptographic primitives now form key software libraries which are run all the time, everywhere, by all kind of computers big and small. It makes sense to optimize crypto software at the assembly level because repetitive portions of programs are tiny (e.g., the inner loop of AES or Advanced Encryption Standard which encodes terabytes every day has but a few lines). Inefficiency may also translate directly into insecurity, by forcing a scale-down of the primitive due to usability concerns \cite[Sec. 9, p.5]{18}. Finally, information leaks such as dependency of execution time on secret data violates the model of security, and such leaks are easier to avoid with hand-crafted assembly instructions. Keys in earlier Linux kernel disk encryption can leak in 65ms \cite{31,33} due to such timing attacks.

It is important that critical software components in any domain be correct, and the cost of failure for crypto libraries is uniquely magnified further by users being adversarial – i.e., whereas bugs are stumbled upon accidentally in normal use for most software, attackers

\* This work was supported by the National Science Foundation under grant 1018836 and by the Netherlands Organisation for Scientific Research (NWO) through Veni 2013 project 13114. Permanent ID of this document: 55ab6668ce87d857c02a5b2d56d7da38. Date: 2014.04.28.; it was supported by the Academia Sinica Institute of Information Science under BY’s Specialty Project and Career Advancement Award.
routinely seek them out and exploit them to filch secrets [16]. Hand-crafted highly optimized assembly makes thorough code auditing extremely tedious and time consuming. It would be ideal if one could take such a cryptographic library and formally verify it to be correct (and even free of instructions which might time differently depending on secret data).

In this paper, we report results on verifying the correctness of such an implementation for the Curve25519 elliptic-curve Diffie-Hellman key-exchange protocol [10], which had initially set speed records for key exchange at the 128-bit security level for 32-bit Intel processors. Cryptography on this curve has since become something of a reference point for the “high-speed-crypto” community, having seen highly optimized implementations for diverse architectures, including the Cell Broadband Engine and the ARM Cortex-A8 [19,14].

Curve25519 has also seen wide adoption from the applied-security community, including Tor [32] (switching from RSA-1024 for speed and security [27]!), Apple’s iOS operating system [5], the Networking and Cryptography library (NaCl) [13], and the latest version of OpenSSH [1]. In addition, it is being considered by an IETF draft [25] for inclusion into TLS (https).

New record speeds for standard 64-bit Intel and AMD processors were set by Gaudry and Thomé [22] and later by Bernstein, Duif, Lange, Schwabe, and Yang [11,12], currently still the fastest Curve25519 implementation on 64-bit Intel and AMD processors. This speed record holder (without conditional branches or secret-index table look-ups) is the target for verification in this paper.

To achieve our goals, we propose a methodology that integrates compositional reasoning, SMT solvers, and proof assistants. Elliptic-curve cryptography performs arithmetic operations in large finite fields. Any verification has to analyze tens of non-linear arithmetic operations on values with hundreds of bits. Despite significant developments in the past decade, such tasks are still infeasible for SMT solvers to verify automatically. Proof assistants, on the other hand, require significant human guidance. Analyzing thousands of low-level assembly instructions is perhaps too inhumane for average enthusiasts.

We attempt to get the best out of these techniques in our integrated methodology. Applying the compositional proof rule in Hoare logic, we divide low-level implementations of cryptographic software into parts. The SMT solver Boolector is used to establish low-level properties of each part automatically. For instance, it is very feasible to verify automatically that instructions are free of overflow, although it can be extremely difficult to prove that the same instructions perform modular multiplication over a large finite field. After low-level properties are established, we use the proof assistant Coq to verify high-level arithmetic properties about cryptographic software. Thanks to SMT solvers, it is not needed to show manually that low-level properties such as free-of-overflow do not occur. Proof assistants can focus on algebraic properties such as modular congruence. Human efforts are thus significantly reduced.

In short, we propose a hybrid methodology that is more scalable than pure SMT solving and theorem proving. With our methodology, we have verified the Montgomery ladder step of a speed-record-holding implementation of the widely used Curve25519 key exchange protocol. The Montgomery ladder step consists of 1277 and 1383 low-level instructions in the radix-$2^{64}$ and radix-$2^{51}$ implementations respectively. To the best of our knowledge, we believe this work is the first on formal verification of the core computation of manually optimized low-level implementations for a real-world cryptographic protocol.
Related work. In [3], a SmartMIPS (a superclass of MIPS32) implementation for a pseudorandomnumber generator is formally proved to be cryptographically secure with the proof assistant Coq. The authors adopt the formalization of MIPS32 instructions in [2] and give a formal reduction proof for their own implementation of the pseudorandom number generator. We combine automatic SMT solvers and proof assistants to verify the core computation of a record-holding implementation of a real-world security protocol. We do not address the security issues in this work, but the scale and impact can be far more significant.

The project CryptVer aims to verify implementations of cryptographic algorithms written in an efficient high-level language [23]. Their approach is to specify cryptographic algorithms and implement them by formal deductive compilation. In contrast, we formally verify manually optimized low-level implementations for cryptographic software. Equivalence checking of low-level code for digital signal processing is discussed in [20]. The authors compare implementations with golden models. We verify the correctness of low-level code directly.

Another approach to formally verified cryptographic software are special-domain compilers. A recent example of this is [4], where Almeida et al. introduce security-aware compilation of a subset of the C programming language.

As stated before, cryptographic software must be more than correct, it must avoid information leaks against its security model, i.e. be “side-channel resistant”. As a consequence, countermeasures against side-channel attacks have also been formalized. For example, Bayrak et al. [7] use SAT solving for the automated verification of power-analysis countermeasures. Molnar et al. [29] describe a tool for static analysis for control-flow vulnerabilities and their automatic removal.

Organization of this paper. Section 2 gives the necessary background on Curve25519 elliptic-curve Diffie-Hellman key exchange. Section 3 reviews the two different approaches for assembly implementations of arithmetic in the field $F_{2^{255} - 19}$ used in [11]. Section 4 gives an overview of the tools we use for verification. Section 5 details our methodology. Section 6 presents and discusses our results. We conclude the paper and point to future work in Section 7.

2 Curve25519

Cryptography based on the arithmetic on elliptic curves was proposed independently by Miller and Koblitz [26,28]. We only give a very brief introduction to elliptic-curve cryptography here; please refer to (for example) [6,24] for more information.

Let $F_q$ be the finite field with $q$ elements. For $a_1, a_2, a_3, a_4, a_6 \in F_q$, an equation of the form $E : y^2 + a_1 xy + a_3 y = x^3 + a_2 x^2 + a_4 x + a_6$ defines an elliptic curve $E$ over $F_q$ (with certain conditions, cf. [24, Chapter 3]). The set of points $(x, y) \in F_q \times F_q$ together with a “point at infinity” form a group of size $\ell \approx q$, usually written additively. The group law is efficiently computable through a few operations in the field $F_q$, so given a point $P$ and a scalar $k \in Z$ it is easy to do a scalar multiplication $k \cdot P$, which requires a number of addition operations that is linear in the length of $k$ (i.e., logarithmic in $k$).

In contrast, for a sufficiently large finite field $F_q$, a suitably chosen curve, and random points $P$ and $Q$, computing the discrete logarithm $\log_P Q$, i.e., to find $k \in Z$ such that $Q = k \cdot P$, is hard. More specifically, for elliptic curves used in cryptography, the best known algorithms takes time $\Theta(\sqrt{\ell})$. Elliptic-curve cryptography is based on this difference in the complexity for computing scalar multiplication and computing discrete logarithms. A user
who knows a secret $k$ and a system parameter $P$ computes and publishes $Q = k \cdot P$. An attacker who wants to break security of the scheme needs to obtain $k$, i.e., compute $\log_P Q$.

Curve25519 is an elliptic-curve Diffie-Hellman key exchange protocol proposed by Bernstein in 2006 [10]. It is based on arithmetic on the elliptic curve $E : y^2 = x^3 + 486662x^2 + x$ defined over the field $\mathbb{F}_{2^{255} - 19}$.

2.1 The Montgomery ladder

Scalar multiplication in Curve25519 uses a so-called differential-addition chain proposed by Montgomery in [30] that uses only the $x$-coordinate of points. It is highly regular, performs one ladder step per scalar bit, and can be run in constant time; the whole loop is often called Montgomery ladder. An overview of the structure of the Montgomery ladder and the operations involved in one ladder-step are given respectively in Algs. 1 and 2. The inputs and outputs $x_P, X_1, X_2, Z_2, X_3, Z_3$, and temporary values $T_i$ are elements in $\mathbb{F}_{2^{255} - 19}$. The performance of the computation is is largely determined by the performance of arithmetic operations in this field.

**Algorithm 1** Curve25519 Montgomery Ladder

\[
\begin{align*}
\text{Input:} & \quad \text{A 255-bit scalar } k \text{ and the } x\text{-coordinate } x_P \text{ of a point } P \text{ on } E. \\
\text{Output:} & \quad (X_{kP}, Z_{kP}) \text{ fulfilling } x_{kP} = X_{kP}/Z_{kP} \\
& \quad t = \lceil \log_2 k + 1 \rceil \\
& \quad X_1 = x_P; X_2 = 1; Z_2 = 0; X_3 = x_P; Z_3 = 1 \\
& \quad \text{for } i \leftarrow t - 1 \text{ downto } 0 \text{ do} \\
& \quad \quad \text{if bit } i \text{ of } k \text{ is } 1 \text{ then} \\
& \quad \quad \quad (X_3, Z_3, X_2, Z_2) \leftarrow \text{ladderstep}(X_1, X_3, Z_3, X_2, Z_2) \\
& \quad \quad \quad \text{else} \\
& \quad \quad \quad (X_2, Z_2, X_3, Z_3) \leftarrow \text{ladderstep}(X_1, X_2, Z_2, X_3, Z_3) \\
& \quad \quad \text{end if} \\
& \quad \text{end for} \\
& \quad \text{return } (X_2, Z_2)
\end{align*}
\]

The biggest difference between the two Curve25519 implementations of Bernstein et al. presented in [11,12] is the representation of elements of $\mathbb{F}_{2^{255} - 19}$. Both implementations have the core parts, the Montgomery ladder step, in fully inlined, hand-optimized assembly. These core portions are what we target for verification in this paper.

3 $\mathbb{F}_{2^{255} - 19}$ arithmetic in AMD64 assembly

Arithmetic in $\mathbb{F}_{2^{255} - 19}$ means addition, subtraction, multiplication and squaring of 255-bit integers modulo the prime $p = 2^{255} − 19$. No mainstream computer architecture offers arithmetic instructions for 255-bit integers directly, so operations on such large integers must be constructed from instructions that work on smaller data types. The AMD64 architecture has instructions to add and subtract (with and without carry/borrow) 64-bit integers, and the MUL instruction returns the 128-bit product of two 64-bit integers, always in general-purpose registers $\text{rdx}$ (higher half) and $\text{rax}$ (lower half).

Section 3 of [11] describes two different approaches to implement $\mathbb{F}_p$ arithmetic in AMD64 assembly. Both approaches use the 64-bit-integer machine instructions. They are different
Algorithm 2 One “step” of the Curve25519 Montgomery Ladder

```plaintext
function ladderstep(X₁, X₂, Z₂, X₃, Z₃)
    T₁ ← X₂ + Z₂
    T₂ ← X₂ − Z₂
    T₇ ← T₂²
    T₆ ← T₁²
    T₅ ← T₆ − T₇
    T₃ ← X₃ + Z₃
    T₄ ← X₃ − Z₃
    T₉ ← T₃ · T₂
    T₈ ← T₄ · T₁
    X₃ ← (T₈ + T₉)
    Z₃ ← (T₈ − T₉)
    X₃ ← X₃²
    Z₃ ← Z₃²
    Z₃ ← Z₃ · X₁
    X₃ ← T₆ · T₇
    Z₃ ← 121666 · T₅
    Z₃ ← Z₃ + T₇
    Z₃ ← Z₃ · T₅
    return (X₂, Z₂, X₃, Z₃)
end function
```

in the representation of elements of $\mathbb{F}_p$, i.e., they decompose the 255-bit field elements into smaller pieces which fit into 64-bit registers in different ways. We now review these approaches and highlight the differences that are most relevant to verification.

### 3.1 Field arithmetic in radix-2⁶⁴ representation

The obvious representation of an element $X \in \mathbb{F}_p$ (or any 256-bit number) with 64 bit integers is radix $2^{64}$. A 256-bit integer $X$ is represented by 4 64-bit unsigned integers $(x_0, x_1, x_2, x_3)$, where the limbs $x_i \in \{0, \ldots, 2^{64} - 1\}$ and

$$X = \sum_{i=0}^{3} x_i 2^{64i} = x_0 + 2^{64}x_1 + 2^{128}x_2 + 2^{192}x_3.$$

We will focus our description on the most complex $\mathbb{F}_p$ operation in the Montgomery ladder step, which is multiplication. Squaring is like multiplying, except that some partial results are known to be zero. Multiplication in $\mathbb{F}_p$ consists of two steps: multiplication of two 256-bit integers to produce a 512-bit intermediate result $S$, and reduction modulo $2p$ to obtain a 256-bit result $R$. Note that the software does not perform a full reduction modulo $p$, but only requires that the result fits into 256 bits.

**Multiplication of 256-bit integers.** The approach for multiplication in radix-$2^{64}$ chosen by [11] is a simple schoolbook approach. Multiplication of two 256-bit integers $X$ and $Y$ can be seen as a 4-step computation which in each step involves one limb of $Y$ and all limbs of $X$ as follows:

$$A_0 = Xy_0, \ A_1 = 2^{64}Xy_1 + A_0, \ A_2 = 2^{128}Xy_2 + A_1, \ S = A_3 = 2^{192}Xy_2 + A_2.$$  \hspace{1cm} (1)
Each step essentially computes and accumulates the 5-limb partial product \(Xy_i\) with 4 \(64 \times 64\)-bit multiplications and several additions as \((x_0y_i + 2^{64}x_1y_i + 2^{128}x_2y_i + 2^{192}x_3y_i)\). Note that “multiplications by \(2^{64}\)” are free and only determine where to add when summing 128-bit products. For example, the result of \(x_0y_i\) is in two 64-bit registers \(t_0\) and \(t_1\) with \(x_0y_i = 2^{64}t_0 + t_1\), therefore \(t_1\) needs to be added to the lower result register of \(x_1y_i\), which in turn produces a carry bit which must go into the register holding the higher half of \(x_1y_i\). Instructions adding \(A_{i-1}\) into \(A_i\) also produce carry bits that need to be propagated through the higher limbs.

Handling the carry bits, which occur inside the radix-\(2^{64}\) multiplication, incurs significant performance penalties on some microarchitectures as detailed in [11, Section 3]. In Sec. 6 we will explain why integrated multiplication and handling of carry bits also constitutes a major obstacle for formal verification.

**Modular reduction.** The multiplication of the two 256-bit integers \(X\) and \(Y\) produced a 512-bit result in \(S = (s_0, \ldots, s_7)\). Now \(S\) must be reduced modulo \(2p = 2^{256} - 38\). We will use repeatedly that \(2^{256} \equiv 38 \pmod{p}\). The initial step of reduction is to compute

\[
S' = (s_0 + 2^{64}s_1 + 2^{128}s_2 + 2^{192}s_3) + 38 \cdot (s_4 + 2^{64}s_5 + 2^{128}s_6 + 2^{192}s_7)
\]

with a 5-limb result \(S' = (s'_0 + 2^{64}s'_1 + 2^{128}s'_2 + 2^{192}s'_3 + 2^{256}s'_4)\). Note that the highest limb \(s'_4\) of \(S'\) has at most 6 bits. A subsequent step computes

\[
S'' = (s'_0 + 2^{64}s'_1 + 2^{128}s'_2 + 2^{192}s'_3 + 38s'_4)
\]

The value \(S'' = (s''_0 + 2^{64}s''_1 + 2^{128}s''_2 + 2^{192}s''_3 + 2^{256}s''_4)\) may still have 257 bits, i.e., \(s''_4\) is either zero or one. The final 4-limb result \(R\) is obtained as

\[
R = (s''_0 + 2^{64}s''_1 + 2^{128}s''_2 + 2^{192}s''_3) + 38s''_4.
\]

### 3.2 Field arithmetic in radix-\(2^{51}\) representation

Due to the performance penalties in handling carries, [11] proposes to represent elements of \(\mathbb{F}_p\) in radix \(2^{51}\), i.e., \(X \in \mathbb{F}_p\) is represented by 5 limbs \((x_0, \ldots, x_4)\) as

\[
X = \sum_{i=0}^{4} x_i2^{51i} = x_0 + 2^{51}x_1 + 2^{102}x_2 + 2^{153}x_3 + 2^{204}x_4.
\]

Every element of \(\mathbb{F}_p\) can be represented with all \(x_i \in [0, 2^{51} - 1]\); however, inputs, outputs, and intermediate results inside the ladder step have relaxed limb-size restrictions. For example, inputs and outputs of the ladder step have limbs in \([0, 2^{51}+2^{15}]\). Additions are done limbwise, e.g., after the first operation \(T_1 \leftarrow X_2 + Z_2\), the limbs of \(T_1\) have at most 53 bits. Subtractions are done by first adding a multiple of \(p\) guaranteed to exceed the subtrahend limbwise. For example, the subtraction \(T_2 \leftarrow X_2 - Z_2\) is done by first adding \(0xFFFFFFFFFFFFFD\) to the lowest limb of \(X_2\) and \(0xFFFFFFFFFFFFFE\) to the four higher limbs of \(X_2\), and then subtracting corresponding limbs of \(Z_2\). The value added is \(2p\), which does not change the result (as element of \(\mathbb{F}_p\)), yet ensures that all limbs of the result \(T_2\) are positive and have at most 53 bits.

The most complex operation—multiplication—is split in two parts, but these now differ from those of Sec. 3.1. The first step performs multiplication and modular reduction; the second step performs the delayed carries.
Multiply-and-Reduce in radix-2^{51}. To multiply \( X = x_0 + 2^{51} x_1 + 2^{102} x_2 + 2^{153} x_3 + 2^{204} x_4 \) and \( Y = y_0 + 2^{51} y_1 + 2^{102} y_2 + 2^{153} y_3 + 2^{204} y_4 \), start by precomputing \( 19 y_1, 19 y_2, 19 y_3 \) and \( 19 y_4 \), then compute 5 intermediate values \( s_0, \ldots, s_4 \), each in two 64-bit registers, as

\[
\begin{align*}
    s_0 & := x_0 y_0 + x_1 (19 y_1) + x_2 (19 y_2) + x_3 (19 y_3) + x_4 (19 y_4), \\
    s_1 & := x_0 y_1 + x_1 y_0 + x_2 (19 y_4) + x_3 (19 y_3) + x_4 (19 y_2), \\
    s_2 & := x_0 y_2 + x_1 y_1 + x_2 y_0 + x_3 (19 y_4) + x_4 (19 y_3), \\
    s_3 & := x_0 y_3 + x_1 y_2 + x_2 y_1 + x_3 y_0 + x_4 (19 y_4), \\
    s_4 & := x_0 y_4 + x_1 y_3 + x_2 y_2 + x_3 y_1 + x_4 y_0.
\end{align*}
\]

(2)

Note that all partial results in this computation are significantly smaller than 128 bits. For example, when \( 0 \leq x_i, y_i < 2^{54} \) (input limbs are at most 54-bits), \( 0 \leq s_0, s_1, s_2, s_3, s_4 < 95 \cdot 2^{108} < 2^{115} \). Thus, adding of multiplication results can be achieved by two 64-bit addition instructions (one with carry); carry bits from these additions are collected by the “free” bits of the higher register of \( s_i \).

Now \( X \cdot Y = S = s_0 + 2^{51} s_1 + 2^{102} s_2 + 2^{153} s_3 + 2^{204} s_4 \), but the limbs \( s_i \) are still two-register values, much too large to be used in subsequent operations.

Delayed carry. For a 2-register value \( s_i \), let \( s_i^{(l)} \) denote the lower register and \( s_i^{(h)} \) the higher register, i.e., \( s_i = s_i^{(l)} + 2^{32} s_i^{(h)} \). Carrying from \( s_i \) to \( s_{i+1} \) is done as follows: Shift \( s_i^{(h)} \) to the left by 13 and shift the 13 high bits of \( s_i^{(l)} \) into the 13 low bits of the result. This operation is just one \textsc{shld} instruction. Now set the high 13 bits of \( s_i^{(l)} \) to zero (logical \textsc{and} with \( 2^{51} - 1 \)). Now do the same shift-by-13 operation on \( s_{i+1}^{(h)} \) and \( s_{i+1}^{(l)} \), set the high 13 bits of \( s_{i+1}^{(l)} \) to zero, add \( s_i^{(h)} \) to \( s_{i+1}^{(l)} \) and discard \( s_i^{(h)} \). This carry chain is performed from \( s_0 \) through \( s_4 \); then \( s_4^{(h)} \) is multiplied by 19 (using a single-word \textsc{mul} and added to \( s_0^{(l)} \)).

Note: We need \( s_i^{(h)} < 2^{51} \) to begin this operation so as not to lose bits from \( s_i^{(h)} \) in the shift-by-13. Due to the multiply by 19 at the end, \( s_4^{(h)} \) must be even smaller, but the expression for \( s_4 \) does not already have a multiply by 19 in Eq. 2, so we can guarantee no overflows if limbs of \( X \) and \( Y \) are at most 54 bits.

This first step of carrying yields \( X Y = s_0^{(l)} + 2^{51} s_1^{(l)} + 2^{102} s_2^{(l)} + 2^{153} s_3^{(l)} + 2^{204} s_4^{(l)} \), but the values in \( s_1^{(l)} \) may still be too big as subsequent operands.

A second carry copies \( s_0^{(l)} \) to a register \( t \), shifts \( t \) to the right by 51, adds \( t \) to \( s_1^{(l)} \) and discards the upper 13 bits of \( s_0^{(l)} \). Carrying continues this way from \( s_1^{(l)} \) to \( s_2^{(l)} \), from \( s_2^{(l)} \) to \( s_3^{(l)} \), and from from \( s_3^{(l)} \) to \( s_4^{(l)} \). Finally \( s_4^{(l)} \) is reduced in the same way; except that 19 \( t \) is added to \( s_0^{(l)} \). This produces the final result

\[
R = (s_0^{(l)} + 2^{51} s_1^{(l)} + 2^{102} s_2^{(l)} + 2^{153} s_3^{(l)} + 2^{204} s_4^{(l)}).
\]

4 Tools Used to Verify Curve25519

4.1 Portable assembly: \texttt{qhasm}

The software we are verifying has not been written directly in AMD64 assembly, but in the portable assembly language \texttt{qhasm} developed by Bernstein [8]. The aim of \texttt{qhasm} is to reduce
development time of assembly software by offering a unified syntax across different architectures and by assisting the assembly programmer with register allocation. Most importantly for us, one line in \texttt{qhasm} translates to exactly one assembly instruction. Also, \texttt{qhasm} guarantees that “register variables” are indeed kept in registers. Spilling to memory has to be done explicitly by the programmer.

Verifying \texttt{qhasm} code. The Curve25519 software we verified is publicly available as part of the SUPERCOP benchmarking framework \cite{9}, but does not include the \texttt{qhasm} source files, which we obtained from the authors. Our verification works on \texttt{qhasm} level. The obvious disadvantage is that we rely on the correctness of \texttt{qhasm} translation. The advantage of this approach is that we can easily adapt our approach to assembly software for other architectures.

4.2 The Boolector SMT solver

\texttt{Boolector} is an efficient SMT solver supporting the bit vector and array theory \cite{17}. In cryptographic software, arithmetic in large finite fields requires hundreds of significant bits. Standard algorithms for linear (integer) arithmetic do not apply. \texttt{Boolector} reduces queries to instances of the Boolean satisfiability problem by bit blasting and is hence more suitable for our purposes.

Theory of arrays is also essential to the formalization of \texttt{qhasm} programs. In low-level programming languages such as \texttt{qhasm}, memory and pointers are indispensable. On the other hand, \texttt{qhasm} programs are finite. Sizes of program variables (including pointers) must be declared. Subsequently, each variable is of finite domain and, more importantly, the memory is finite. Since formal models of \texttt{qhasm} programs are necessarily finite, they are expressible in theories of bit vectors and arrays. \texttt{Boolector} therefore fits perfectly in this application.

4.3 The Coq proof assistant

The Coq proof assistant has been developed in INRIA for more than twenty years \cite{15}. The tool is based on a higher-order logic called the Calculus of Inductive Construction and has lots of libraries for various theories. In contrast to model-theoretic tools such as \texttt{Boolector}, proof assistants are optimized for symbolic reasoning. For instance, the algebraic equation \((x + y)^2 = x^2 + 2xy + y^2\) can be verified by the Coq tactic \texttt{ring} instantaneously.

In this work, we use the Coq standard library \texttt{ZArith} to formalize the congruence relation modulo \(2^{255} - 19\). For non-linear modulo relations in \(F_{2^{255} - 19}\), \texttt{Boolector} may fail to verify in a handful of cases. We verify them with our formalization and simple rewrite tactics in Coq.

5 Methodology

We aim to verify the Montgomery ladder step of the record-holding implementation of Curve25519 in \cite{11,12}. A ladder step (Alg. 2) consists of 18 field arithmetic operations. Considering the complexity of a \(F_p\) multiplication (Sec. 3), the correctness of manually optimized \texttt{qhasm} implementation for the Montgomery ladder step is by no mean clear. Due to space limit, we only detail the verification of \(F_p\) multiplication. Other field arithmetic and the Montgomery ladder step (Alg. 2) itself are handled similarly.

We will use Hoare triples to specify properties about \texttt{qhasm} implementations. Let \(Q, Q'\) be predicate logic formulae, and \(P\) a \texttt{qhasm} program fragment. Informally, a \texttt{Hoare triple}
In order to automate this process, we have defined a simple assertion language to specify pre- and post-conditions. We only use quantifier-free pre- and post-conditions. The typewriter and Fraktur fonts are used to denote program and logical variables respectively in pre- and post-conditions.

Let $P$ be a qhasm implementation for $\mathbb{F}_p$ multiplication. Note that $P$ is loop-free. When the pre- and post-conditions are quantifier-free, it is straightforward to translate a Hoare triple $(Q)P(Q')$ to a Boolector specification. We first convert the qhasm fragment $P$ into static single assignment form $P_{SSA}$, and then check whether the quantifier-free formula $Q \land P_{SSA} \land \neg Q'$ in the bit-vector theory is satisfiable. If not, we have established $\models (Q)P(Q')$. In order to automate this process, we have defined a simple assertion language to specify pre- and post-conditions in qhasm. Moreover, we have build a converter that translates annotated qhasm fragments into Boolector specifications.

### 5.1 $\mathbb{F}_p$ multiplication in the radix-$2^{64}$ representation

Let $P64$ denote the qhasm program for $\mathbb{F}_p$ multiplication in the radix-$2^{64}$ representation. The inputs $X = x_0 + 2^{64}x_1 + 2^{128}x_2 + 2^{192}x_3$ and $Y = y_0 + 2^{64}y_1 + 2^{128}y_2 + 2^{192}y_3$ are stored in memory pointed to by the qhasm variables $x_p$ and $y_p$ respectively. qhasm uses a C-like syntax. Pointer dereferences, pointer arithmetic, and type coercion are allowed in qhasm expressions. Thus, the limbs $x_i$ and $y_i$ correspond to the qhasm expressions $*(\text{uint64 } *)(x_p + 8i)$ and $*(\text{uint64 } *)(y_p + 8i)$ respectively for every $i \in [0, 3]$. We introduce logical variables $x_i$ and $y_i$ to record the limbs $x_i$ and $y_i$ respectively. Consider

$$Q_{64_{xy,eqns}} := \sum_{i=0}^{3} x_i 64 \equiv (\text{uint64 } *)(x_p + 8i) \land \sum_{i=0}^{3} y_i 64 \equiv (\text{uint64 } *)(y_p + 8i)$$

The operator $\equiv$ denotes the $n$-bit equality in the theory of bit-vectors. The formula $Q_{64_{xy,eqns}}$ states that the values of the logical variables $x_i$ and $y_i$ are equal to the limbs $x_i$ and $y_i$ of the initial inputs respectively.

In $P64$, the outcome is stored in memory pointed to by the qhasm variable $r_p$. That is, the limb $r_i$ of $R = r_0 + 2^{64}r_1 + 2^{128}r_2 + 2^{192}r_3$ corresponds to the qhasm expressions $*(\text{uint64 } *)(r_p + 8i)$ for every $i \in [0, 3]$. Define

$$Q_{64_{r,eqns}} := \sum_{i=0}^{3} r_i 64 \equiv (\text{uint64 } *)(r_p + 8i)$$

$$Q_{64_{prod}} := (\sum_{i=0}^{3} x_i 2^{64i}) \times (\sum_{i=0}^{3} y_i 2^{64i}) \equiv \sum_{i=0}^{3} r_i 2^{64i} \pmod{p}$$

The operator $\equiv$ denotes the $n$-bit signed modulo operator in the bit-vector theory. The formula $Q_{64_{r,eqns}}$ introduces the logical variable $r_i$ equal to the coefficient $r_i$ for $0 \leq i \leq 3$. The formula $Q_{64_{prod}}$ specifies that the outcome $R$ is indeed the product of $X$ and $Y$ in field arithmetic.

Consider the top-level Hoare triple $(Q_{64_{xy,eqns}})P64(Q_{64_{r,eqns}} \land Q_{64_{prod}})$. We are concerned about the outcomes of the qhasm fragment $P64$ from states where logical variables $x_i$ and $y_i$ are equal to limbs of the inputs pointed to by the program variables $x_p$ and $y_p$ respectively. During the execution of the qhasm program $P64$, program variables may change their values. Logical variables, on the other hand, remain unchanged. The logical variables $x_i, y_i$ in the pre-condition $Q_{64_{xy,eqns}}$ effectively memorize the input limbs before the execution of $P64$. The post-condition $Q_{64_{r,eqns}} \land Q_{64_{prod}}$ furthermore specifies that the outcome pointed
to by the program variable \( r_p \) is the product of the inputs stored in \( r_i \) and \( y_i \). In other words, the top-level Hoare triple specifies that the \texttt{qhasm} fragment \( P64 \) is \( \mathbb{F}_p \) multiplication in the radix-2\(^{64} \) representation.

The top-level Hoare triple contains complicated arithmetic operations over hundreds of 64-bit vectors. It is perhaps not unexpected that naive verification fails. In order to verify the \texttt{qhasm} implementation of \( \mathbb{F}_p \) multiplication, we exploit the compositional property of proofs for sequential programs. Recall the proof rule for compositions in Hoare logic, where \( Q' \) is a mid-condition:

\[
\vdash (Q) P_0 \quad \vdash (Q') P_1 \quad \vdash (Q) P_0; P_1 (Q')
\]

Composition

Applying the proof rule, it suffices to find a mid-condition for the top-level Hoare triple. Recall that \( \mathbb{F}_p \) multiplication can be divided into two phases: multiply and reduce (Sec. 3.1). It is but natural to verify each phase separately. More precisely, we introduce logical variables to memorize values of program variables at start and end of each phase. The computation of each phase is thus specified by arithmetic relations between logical variables.

Multiply in the radix-2\(^{64} \) representation Let \( P64_M \) and \( P64_R \) denote the \texttt{qhasm} fragments for multiply and reduce respectively. The multiply fragment \( P64_M \) computes the 512-bit value \( S = (s_0, \ldots, s_7) \) in (1) stored in the memory pointed to by the \texttt{qhasm} variable \( sp \). Thus each 64-bit value \( s_i \) corresponds to the \texttt{qhasm} expression \( \ast (\text{uint } 64 \ast) (sp + 8i) \) for every \( i \in [0, 3] \). Define

\[
Q64_{s,\text{eqns}} := \bigwedge_{i=0}^{7} s_i = \ast (\text{uint } 64 \ast) (sp + 8i)
\]

\[
Q64_{\text{mult}} := A_0 = 512 \cdot x_0 + A_1 2^{64} x_1 + A_0 \land A_2 5122^{128} x_2 + A_1 \land
\]

\[
A_3 = 2^{192} x_3 + A_2 \land A = \sum_{i=0}^{7} r_i 2^{64i} \land \sum_{i=0}^{7} s_i 2^{64i} 512 = A_3
\]

For clarity, we introduce the logical variable \( X \) for the input \( X = x_0 + 2^{64} x_1 + 2^{128} x_2 + 2^{192} x_3 \) in \( Q64_{\text{mult}} \). Consider the Hoare triple \( \langle Q64_{x,y,\text{eqns}} \rangle P64_M \langle Q64_{x,y,\text{eqns}} \land Q64_{\text{mult}} \rangle \) The pre-condition \( Q64_{x,y,\text{eqns}} \) memorizes the limbs of the inputs \( X \) and \( Y \) in logical variables \( r_i \)'s and \( y_i \)'s. The formula \( Q64_{s,\text{eqns}} \) records the limbs \( s_i \)'s after the \texttt{qhasm} fragment \( P64_M \) in logical variables \( s_i \)'s. \( Q64_{\text{mult}} \) ensures that the limbs \( s_i \)'s are computed according to (1). In other words, the Hoare triple specifies the multiply phase of \( \mathbb{F}_p \) multiplication in the radix-2\(^{64} \) representation.

Reduce in the radix-2\(^{64} \) representation Following the reduction phase in Sec. 3.1, we introduce logical variables \( s'_i \) and \( s''_i \) for the limbs \( s'_i \) and \( s''_i \) respectively for every \( i \in [0, 4] \). The formulae \( Q64_{s'_i,\text{red}}, Q64_{s''_i,\text{red}}, Q64_{r,\text{red}} \) are defined for the three reduction steps. The formulae \( Q64_{s'_i,\text{bds}}, Q64_{s''_i,\text{bds}}, \) and \( Q64_{r,\text{bds}} \) moreover give upper bounds.
Proposition 1. Assume

\[ Q_{64s_{-}red} := \sum_{i=0}^{4} s_i 2^{64i} = s_0 + 2^{64}s_1 + 2^{128}s_2 + 2^{192}s_3 + 38(s_4 + 2^{64}s_5 + 2^{128}s_6 + 2^{192}s_7) \]

\[ Q_{64s'_{-}red} := \sum_{i=0}^{4} s_i' 2^{64i} = 38s_4' + \sum_{i=0}^{3} s_i 2^{64i} \]

\[ Q_{64s''_{-}red} := \sum_{i=0}^{3} t_i 2^{64i} = 38s_4'' + \sum_{i=0}^{3} s_i 2^{64i} \]

Consider the following Hoare triple

\[ (Q_{64s_{-}red} \land Q_{64s'_{-}red} \land Q_{64s''_{-}red} \land Q_{64s_{-}bds}) \]

The pre-condition \( Q_{64s_{-}bds} \) assumes that variables \( s_i \)'s are obtained from the multiply phase. Recall the formula \( Q_{64r_{-}eqns} \) defined at the beginning of this subsection. The post-condition states that outcome \( r_i \)'s are obtained by the reduce phase. Note that the logical variable \( s_i'' \) for the limb \( s_i'' \) is at most 1. We are using BOOLECTOR to verify this fact in the reduction phase.

**Proposition 1. Assume**

1. \( \models (Q_{64s_{eqns}} \land Q_{64r_{eqns}} \land Q_{64s_{bds}}) \);
2. \( \models (Q_{64s_{bds}} \land Q_{64s'_{bds}} \land Q_{64s''_{bds}} \land Q_{64r_{bds}} \land Q_{64r_{eqns}}) \).

Then \( \models (Q_{64s_{eqns}} \land Q_{64r_{bds}}) \).

Note that the Hoare triples in the proposition do not establish \( Q_{64s_{bds}} \) directly. Indeed, we need to show

\[ Q_{64s_{bds}} \land Q_{64s'_{bds}} \land Q_{64s''_{bds}} \land Q_{64r_{bds}} \land Q_{64r_{eqns}} \implies Q_{64s_{bds}} \]

in the proof of Proposition 1. Observe that the statement involves modular operations in the bit-vector theory. Although the statement is expressible in a quantifier-free formula in the theory of bit-vectors, the SMT solver BOOLECTOR could not verify it. We therefore use the proof assistant Coq to formally prove the statement. With simple facts about modular arithmetic such as \( 38 \equiv 2^{56} \) (mod \( p \)), our formal Coq proof needs less than 800 lines.

### 5.2 \( \mathbb{F}_p \) multiplication in the radix-2^{51} representation

Let \( P51 \) denote the qhasm fragment for \( \mathbb{F}_p \) multiplication in radix-2^{51} representation. The inputs \( X = x_0 + 2^{51}x_1 + 2^{102}x_2 + 2^{153}x_3 + 2^{204}x_4, Y = y_0 + 2^{51}y_1 + 2^{102}y_2 + 2^{153}y_3 + 2^{204}y_4 \), and outcome \( R = r_0 + 2^{51}r_1 + 2^{102}r_2 + 2^{153}r_3 + 2^{204}r_4 \) are stored in memory pointed to by the qhasm variables \( x_0, y_0, \) and \( r_0 \) respectively. We thus introduce logical variables \( t_i, n_i, \) and \( r_i \) to memoize the values of the qhasm expressions *(uint64 *)[0] xp + 8i, *(uint64 *)[0] yp + 8i, \) and *(uint64 *)[0] rp + 8i respectively for every \( i \in [0, 4] \).

The formulae \( Q_{51s_{eqns}}, Q_{51r_{eqns}}, Q_{51prod} \) are defined similarly as in the radix-2^{64} representation. Observe that each limb in the radix-2^{51} representation has only 51 significant bits.
The formulae $Q_{51,xy,bds}$ and $Q_{51,r,bds}$ specify that the inputs and outcome are in the radix-$2^{51}$ representation.

\[
Q_{51,xy,bds} := \prod_{i=0}^{4} (0 \leq r_i < 2^{51}) \land \prod_{i=0}^{4} (0 \leq q_i < 2^{51}) \quad Q_{51,r,bds} := \prod_{i=0}^{4} (0 \leq r_i < 2^{51})
\]

In the top-level Hoare triple $(\langle Q_{51,xy,equ} \land Q_{51,xy,bds} \rangle P_{51} (\langle Q_{51,r,equ} \land Q_{51,r,bds} \land Q_{51,prod} \rangle))$, the pre-condition $Q_{51,xy,equ} \land Q_{51,xy,bds}$ assumes that the inputs $X$ and $Y$ are in the radix-$2^{51}$ representation. The post-condition $Q_{51,r,equ} \land Q_{51,r,bds} \land Q_{51,prod}$ specifies that the outcome is the product of $X$ and $Y$ in the radix-$2^{51}$ representation. The top-level Hoare triple hence specifies that the \texttt{qhasm} fragment $P_{51}$ is $\mathbb{F}_p$ multiplication in the radix-$2^{51}$ representation.

Similar to the case in the radix-$2^{64}$ representation, the top-level Hoare triple should be decomposed before verification. Recall that $\mathbb{F}_p$ multiplication in the radix-$2^{64}$ representation has two phases: multiply-and-reduce and delayed carry (Sec. 3.2). We therefore verify each phase separately.

Let $P_{51,MR}$ and $P_{51,D}$ denote the \texttt{qhasm} fragment for multiply-and-reduce and delayed carry respectively. In the multiply-and-reduce phase, the \texttt{qhasm} fragment $P_{51,MR}$ computes $s_i$’s in (2). Since each $s_i$ has 128 significant bits, $P_{51,MR}$ actually stores each $s_i$ in a pair of 64-bit \texttt{qhasm} variables $s_i, l$ and $s_i, h$. We will use the \texttt{qhasm} expression $u.v$ to denote $u \times 2^{64} + v$. Define

\[
Q_{51,s,equ} := \prod_{i=0}^{128} (s_i = 0) \\
Q_{51,mult_red} := Q_{51,s,equ} \land \begin{cases} 
0 &= (f_0 \eta_0 + 19(f_1 \eta_4 + f_2 \eta_3 + f_3 \eta_2 + f_4 \eta_1)) \\
1 &= (f_0 \eta_1 + f_2 \eta_0 + 19(f_1 \eta_4 + f_3 \eta_2 + f_4 \eta_3)) \\
2 &= (f_0 \eta_2 + f_1 \eta_1 + f_3 \eta_0 + 19(f_4 \eta_3 + f_4 \eta_3)) \\
3 &= (f_0 \eta_3 + f_2 \eta_0 + f_3 \eta_1 + f_3 \eta_0 + 19(f_4 \eta_4)) \\
4 &= (f_0 \eta_4 + f_2 \eta_0 + f_3 \eta_1 + f_3 \eta_0 + f_4 \eta_0)
\end{cases}
\]

$Q_{51,s,equ}$ states that the logical variable $s_i$ is equal to the \texttt{qhasm} expression $s_i, h, s_i, l$ for every $i \in \{0, 4\}$. $Q_{51,mult_red}$ specifies that $s_i$ are computed correctly for every $i \in [0, 4]$. Nonetheless, we find the condition $Q_{51,mult_red}$ is too weak to prove the correctness of the multiply-and-reduce phase. If $s_i, h, s_i, l$ indeed had 128 significant bits, overflow could occur during bitwise operations in multiply-and-reduce. To verify multiplication, we estimate tighter upper bounds for $s_i$’s.

Recall that $s_i$’s are sums of products of $x_i$’s and $y_j$’s which are bounded by $2^{51}$. A simple computation gives us better upper bounds for $s_i$’s. Define

\[
Q_{51,s,bds} := \begin{cases} 
0 &\leq s_0 \leq 4 \cdot 2^{102} + 3 \cdot 2^{102} \land 0 \leq s_1 \leq 2 \cdot 2^{102} + 3 \cdot 19 \cdot 2^{102} \\
0 &\leq s_2 \leq 3 \cdot 2^{102} + 2 \cdot 19 \cdot 2^{102} \land 0 \leq s_3 \leq 4 \cdot 2^{102} + 19 \cdot 2^{102} \\
0 &\leq s_4 \leq 5 \cdot 2^{102}
\end{cases}
\]

Consider the Hoare triple $(\langle Q_{51,xy,equ} \land Q_{51,xy,bds} \rangle P_{51,MR} (\langle Q_{51,s,equ} \land Q_{51,s,bds} \land Q_{51,mult_red} \rangle))$, in addition to checking whether \texttt{qhasm} variables $s_i, h$’s and $s_i, l$’s are computed correctly, the \texttt{qhasm} fragment $P_{51,MR}$ for multiply-reduce is required to meet the upper bounds in $Q_{51,s,bds}$. The mid-condition $Q_{51,s,bds} \land Q_{51,mult_red}$ enables the verification of the \texttt{qhasm} fragment $P_{51,D}$ for the delayed carry phase.
The \texttt{qhasm} fragment $P51_D$ for delayed carry performs carrying on 128-bit expressions $s_i, t_i$ to obtain the product of the inputs $X$ and $Y$. The product must also be in the radix-2\textsuperscript{51} representation. Define

$$Q51_{\text{delayed\_carry}} := \sum_{i=0}^{4} s_i 2^{51i} \equiv \sum_{i=0}^{4} t_i 2^{51i} \pmod{p}.$$ 

The Hoare triple $([Q51_{\text{eqns}} \land Q51_{\text{bds}} \land Q51_{\text{mult\_red}}] P51_D [Q51_{\text{delayed\_carry}} \land Q51_{\text{r\_bds}}])$ verifies that the \texttt{qhasm} fragment $P51_D$ computes a number $\sum_{i=0}^{4} t_i 2^{51i}$ in the radix-2\textsuperscript{51} representation, and it is congruent to $\sum_{i=0}^{4} s_i 2^{51i}$ modulo $p$.

**Proposition 2.** Assume

1. $\models ([Q51_{xy\_eqns} \land Q51_{xy\_bds}] P51_{MR} [Q51_{xy\_eqns} \land Q51_{xy\_bds} \land Q51_{\text{mult\_red}}]);$
2. $\models ([Q51_{s\_eqns} \land Q51_{s\_bds} \land Q51_{\text{mult\_red}}] P51_D [Q51_{\text{delayed\_carry}} \land Q51_{r\_bds}]);$

Then $\models ([Q51_{xy\_eqns} \land Q51_{xy\_bds}] P51_{MR}; P51_D [Q51_{\text{prod}} \land Q51_{r\_bds}]).$

The Hoare triples in the proposition do not establish $Q51_{\text{prod}}$ directly. Again, we formally show $\vdash [Q51_{xy\_bds} \land Q51_{\text{mult\_red}} \land Q51_{\text{delayed\_carry}}] \implies Q51_{\text{prod}}$ in the proof assistant Coq. Our Coq proof contains less than 600 lines.

The verification of the Montgomery ladder step (Alg. 2) is carried out after all field arithmetic operations are verified separately. We replace fragments for field arithmetic by their corresponding pre- and post-conditions. Alg. 2 is then converted to the static single assignment form. We then assert the static single assignments as the post-condition of Alg. 2. Using \textsc{Boolector}, we verify that the post-condition holds, and that the post-condition of every field operation implies the pre-condition of the following field operation in Alg. 2. The record-holding implementation for the Montgomery ladder step in the radix-2\textsuperscript{51} and radix-2\textsuperscript{64} representations are formally verified.

### 6 Results and Discussion

In this section, we present results and findings during the verification process. A summary of the experimental results is in Table 1. The columns are the number of limbs, the number of mid-conditions used, and the verification time used in \textsc{Boolector}. We run \textsc{Boolector} 1.6.0 on a Linux machine with 3.07-GHz CPU and 32-GB memory. We did not set a timeout and thus a verification task can run until it is killed by the operating system.

We formally verified the ladder step in Algorithm 2 in both radix-2\textsuperscript{64} and radix-2\textsuperscript{51}. The pre and postconditions of each operators are obtained from the verification of the corresponding \texttt{qhasm} code fe25519r64*/fe25519*. We are able to reproduce a known bug in an old version of $F_{2^{255}-19}$ multiplication (fe25519r64\_mul-1). A counterexample can be found in seconds with a pair of precondition and postcondition for the reduction phase. The rows mul25519-p2-1 and mul25519-p2-2 are the results of verifying the delayed carry phase of mul25519, a 3-phase implementation of a multiplication. The result shows that if we add an additional mid-condition to the delayed carry phase of mul25519, the verification time of the delayed carry phase can be reduced from 2723 minutes to 263 minutes.
The delayed carry phase of \( r = x \times y \) (mod \( 2^{51} - 19 \)) is thus simplified. The reason why some congruence relations is verified in the radix-2 representation is because we are able to divide \( P \) further into smaller fragments. A few extra carry bits can not only reduce the time for execution but also verification.

We found the following heuristics are quite useful to accelerate verification. We cannot verify many of the cases without them. First, we split conjuncted postconditions, i.e., translate \((Q_0 \land P(Q_1 \land Q_2))\) to \((Q_0 \land P(Q_1))\) and \((Q_0 \land P(Q_2))\). This reduces the verification time of the multiply phase of \texttt{mul25519} from one day to one minute. Second, we delay bit-width extension. For example, consider a formula \( a \overset{256}{=} b * c \) where \( a \) has 256 bits and \( b, c \) have 64 bits. Instead of extending \( b \) and \( c \) to 256 bits before the multiplication, we first extend \( b \) and \( c \) to 128 bits, compute the multiplication, and then extend the result to 256 bits. Third, we over-approximate \textsc{Boolector} specifications automatically by reducing logical variables and weakening preconditions. The validity of an over-approximated specification guaranteed the validity of the original one, but not vice versa. To be more specific, given a specification \( [Q_0 \land Q_1 \land \cdots \land Q_n] \land P(Q_1) \), our translator automatically removes logical variables that do not appear in \( Q_1 \); it removes \( Q_0 \) if some variable in \( Q_0 \) neither appears in \( Q_1 \) nor gets updated in \( P \). We cannot verify \texttt{fe25519r64_sq} and \texttt{fe25519_sq} without this heuristic.

7 Future work

There are several avenues for future work. One interesting topic could be to develop verification approaches for ensuring that the an assembly implementation is resistant against side-channel attacks. Formal techniques in measuring worst case execution time (WCET) might be a starting point for this line of research.

Currently, we need to manually provide mid-conditions for verifications. Although the verification steps between preconditions and postconditions are done automatically, it would be even better if we can increase the degree of automation further by investigating techniques.
for automatic insertion of mid-conditions. We think the tools for automatic assertion insertion could be relevant [21]. The tool obtain assertions based on given templates of assertions and by synthesizing them dynamically from observed executions traces.

Our translator currently can produce BOOLECTOR specifications from annotated qhasm files. Recall that some properties that cannot be proved in BOOLECTOR are proved in Coq. It would be good if the translator can produce both BOOLECTOR specifications and Coq proof obligations from an annotated qhasm file, which makes the qhasm file more self-contained. Moreover, tactics of Coq may be developed to solve some specific problems, for example, modular congruence, to reduce human work.

References


15


30. Tor project: Anonymity online. [https://www.torproject.org/](https://www.torproject.org/).


33. Tor project: Anonymity online. [https://www.torproject.org/](https://www.torproject.org/).