Verifying Curve25519 Software

Yu-Fang Chen\textsuperscript{1}  
yfc@iis.sinica.edu.tw

Chang-Hong Hsu\textsuperscript{2}  
hsuch@umich.edu

Hsin-Hung Lin\textsuperscript{3}  
h-lin@ait.kyushu-u.ac.jp

Peter Schwabe\textsuperscript{4}  
peter@cryptojedi.org

Ming-Hsien Tsai\textsuperscript{1}  
mhtsai208@gmail.com

Bo-Yin Yang\textsuperscript{1}  
by@crypto.tw

Shang-Yi Yang\textsuperscript{1}  
ilway25@crypto.tw

\textsuperscript{1}Institute of Information Science, Academia Sinica, Taiwan  
\textsuperscript{2}University of Michigan, Ann Arbor, USA  
\textsuperscript{3}Faculty of Information Science and Electrical Engineering, Kyushu University, Japan  
\textsuperscript{4}Institute for Computing and Information Sciences, Radboud University Nijmegen, The Netherlands

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 presented by Bernstein et al. at CHES 2011. Two versions for different microarchitectures are available. We successfully verify the core part of the computation, and reproduce detection of a bug in a previously published edition. An SMT solver supporting array and bit-vector theories is used to establish almost all properties. Remaining properties are verified in a proof assistant with simple rewrite tactics. 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.

Keywords

Elliptic-curve cryptography, optimized assembly, Hoare logic, SMT solver, Boolector, Coq.

1. INTRODUCTION

Optimization of cryptographic primitives and protocols for high performance in software is a very active field of research. Papers that report new speed records for, e.g., elliptic-curve cryptography are published at top cryptography venues like Crypto, Eurocrypt, Asiacrypt or CHES. See, for example, \[12, 29, 33, 24\].

One might expect that the software described in these papers is quickly included in cryptographic libraries so that users benefit from the speedups; however this is often not the case. Sometimes the reason is that the software is not (freely) available or that the primitive is incompatible with existing cryptographic infrastructure. However, in other cases the reason is simply that we do not know whether the software is correct. For example, the designers of the Networking and Cryptography library (NaCl) \[18\] announced in 2011 that they would include Ed25519 signatures \[13\] in NaCl. This has not happened until today; the reason is given in \[17\]: “Auditing the NaCl source is a time-consuming job.”

All of the papers listed above that report speed records for elliptic-curve cryptography rely on large portions of in-line hand-optimized assembly code to achieve the speeds. For example, the Ed25519 signature software consists of 4 different implementations that together have 5521 lines of C code and 16184 lines of \texttt{qhasm} code. The two speed-record-setting Curve25519 \[11\] elliptic-curve Diffie-Hellman implementations, which were also presented in \[13\], consist of 342 lines of C and 4064 lines of \texttt{qhasm} code. The \texttt{qhasm} programming language is a high-level assembler introduced by Bernstein \[9\]. What is particularly interesting about this Ed25519 and Curve25519 software from a correctness perspective is not only that it is waiting for inclusion in a widely used cryptographic library but also that the software initially had a bug which is not found by extensive testing\textsuperscript{1}.

In principle, there are three different approaches to ensure the correctness of software:

Testing: Every serious cryptographic library includes extensive test batteries. Software testing has many advantages: it is relatively cheap, it does not conflict with software performance, and it is able to catch a large amount of bugs. The last aspect is amplified by the nature of many cryptographic algorithms, which require that each input bit influences each output bit and typically also most intermediate values. For example it is very hard to imagine AES software which is correct for almost all inputs but fails in some rare cases.

However, some bugs are naturally much harder to test for. One such example is carries (borrows) in modern elliptic curve cryptography that often requires big

---

\textsuperscript{1}This bug is documented, see \url{http://cryptojedi.org/crypto/#ed25519}
integer arithmetic modulo something very close to a power of two. Thus, often one side in a conditional statement would be extremely rare when considered as a statistical event. The bug reported for the Ed25519 and Curve25519 software from [13] falls into this category. Also the bug in the OpenSSL elliptic-curve Diffie-Hellman implementation, which was exploited in [22], falls into this category.

**Auditing:** One way to find bugs that are not found by testing are code audits. Such audits by experts are a widely accepted means to ensure correctness and generally quality of software, but they come at a relatively high cost. For example, a community effort collected more than US$50,000 for a new ongoing audit of TrueCrypt [37]. The cost for a full audit of OpenSSL has recently been estimated to be US$250,000 [34].

Another disadvantage of software auditing is a conflict with performance. Software that is relatively easy (and thus cheap) to audit is concise, has small amounts of code, and is naturally portable. High-speed cryptographic software is written in assembly with optimizations for multiple architectures and micro-architectures. The core development team of NaCl, together with Janssen, recently released TweetNaCl, a re-implementation of NaCl which is optimized for conciseness and audit-ability [17]. The authors state that TweetNaCl “allows correct functionality to be verified by human auditors with reasonable effort”. However, Curve25519 in TweetNaCl takes 2.5 million cycles on an Intel Ivy Bridge processor – more than 10 times more than the assembly implementation presented in [13].

**Verification:** The third direction to ensure correctness of (cryptographic) software is formal verification; this direction offers the strongest guarantees for the correctness of software. There are two major streams of formal verification approaches, namely model checking and theorem proving. In the model checking approach, an abstract model is first built for a program and then automatically and exhaustively explored to see if there is a counterexample of a property. In the theorem-proving approach, a program and its properties have to be first formalized in the meta-logic of a proof assistant and then the proof of the properties are manually deduced within the proof assistant. Theorem proving requires more manual work, but it is capable of proving harder properties.

The current state of the art in formal verification is far away from being able to verify a complete cryptographic library, which is optimized for speed. From a cryptographic-engineering perspective it is obvious that verification should prioritize those portions of code that are expensive to audit and that may contain bugs which are not revealed by testing. The formal verification of precisely this kind of code is the content of this paper.

**Contributions of this paper.** We describe the formal verification of the central hand-optimized assembly routine of each of the two implementations of Curve25519 Diffie-Hellman key-exchange software presented in [13]. This software is still today the speed-record holder for Curve25519; see [15]. To the best of our knowledge, our work is the first to formally verify an inline hand-optimized assembly implementation of a real-world cryptographic protocol.

The correctness of the mathematical formulas in the computation of Curve25519 Diffie-Hellman key exchange is ensured by Sage verification scripts in [16]². So we assume that those formulas are correct and verify that the low-level implementation correctly implements the formulas. Our verification shows that the core routine of one of the two implementations was indeed correct right from the beginning, reproduces detection of the bug in the other and shows that the bug-fixed version of that software is also correct.

We also present a hybrid methodology that integrates compositional reasoning, SMT (Satisfiability Modulo Theories) solvers, and proof assistants. A language is introduced for annotating qasm code with preconditions on inputs and postconditions on intermediate values and outputs. With the annotations, the compositional reasoning in Hoare logic [28] allows us to boil down the verification of a large program to the verification of smaller programs. We then automatically translate this annotated qasm code to several SMT formulas and use an SMT solver to prove that the code matches the conditions specified in the annotations. To achieve better verifiability, we develop heuristics to help the SMT solver. For a small set of algebraic properties such as modular congruence that are hard for SMT solvers, we rely on proof assistants.

**Related work.** Cryptographic software forms the backbone of information security and it is thus widely accepted that correctness of such software is important enough to justify formal verification efforts.

One approach is to re-implement cryptographic protocols in languages and frameworks that allow efficient verification. The most extensive work in this area is miTLS, a “verified reference implementation of the TLS protocol” [20, 21]. This implementation of TLS is written in F# and specified in F7 – the clear focus is on a verifiable (and verified) re-implementation; not on verifying existing high-speed cryptographic software. Note that miTLS relies on (unverified) “cryptographic providers such as .NET or Bouncy Castle” for core cryptographic primitives. Also the CryptVer project [26] aims at re-implementing cryptography such that it can be formally verified. Their approach is to specify cryptographic algorithms in higher-order logic and then implement them by formal deductive compilation.

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.

The theory of elliptic curve has been formalized in [36, 7]. In principle, the mathematical formulas in Curve25519 Diffie-Hellman key-exchange can be verified with the formalization. Low-level machine codes have been formalized in proof assistants [1, 3, 2, 32]. Large integer arithmetic and cryptographic functions can be formally verified semi-automatically. Our approach is very lightweight. Most of the verification is performed by an SMT solver automatically. It hence requires much less human intervention.

Cryptographic software must be more than correct, it must avoid leaks of secret information through side channels. For example, if the execution time of cryptographic software depends on secret data, this can be exploited by an attacker in a so-called timing attack. As a consequence, countermeasures against side-channel attacks have also been formalized. For example, Bayarak et al. [8] use SAT solving for the automated verification of power-analysis countermeasures. Molnar et al. [30] describe a tool for static analysis of control-flow vulnerabilities and their automatic removal.

**Availability of software.** To maximize reusability of our results we placed the tools and software presented in this paper into the public domain. They are available at [http://cryptojedi.org/crypto/#verify25519](http://cryptojedi.org/crypto/#verify25519).

**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 $\mathbb{F}_{2^{255}−19}$ used in [13]. Section 4 gives the necessary background on verification techniques and describes 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

To establish context, we briefly review the basics of elliptic-curve cryptography. For more information see, for example, [6, 27]. Let $\mathbb{F}_q$ be the finite field with $q$ elements. For coefficients $a_1, a_2, a_3, a_4, a_6 \in \mathbb{F}_q$, 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 $\mathbb{F}_q$ (if certain conditions hold, cf. [27], Chapter 3). The set of points $(x, y) \in \mathbb{F}_q \times \mathbb{F}_q$ that fulfill the equation $E$, together with a “point at infinity”, form a group of size $\ell \approx q$, which is usually written additively. Addition under this group law is efficiently computable through a few operations in the field $\mathbb{F}_q$. Given a point $P$ on the curve and a scalar $k \in \mathbb{Z}$ it is easy to do a scalar multiplication $k \cdot P$; the number of group additions required for such a scalar multiplication is linear in the length of $k$ (i.e., logarithmic in $k$).

In contrast, for a sufficiently large finite field $\mathbb{F}_q$, a suitably chosen curve, and random points $P$ and $Q$, computing the discrete logarithm $\log_P Q$, i.e., finding $k \in \mathbb{Z}$ such that $Q = k \cdot P$, is hard. More specifically, for elliptic curves used in cryptography, the best known algorithms take time $O(\sqrt{\ell})$. 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 $\log_Q P$.

Curve25519 is an elliptic-curve Diffie-Hellman key exchange protocol proposed by Bernstein in 2006 [11]. 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

Curve25519 uses a so-called differential-addition chain proposed by Montgomery [31] to multiply a point, identified only by its $x$-coordinate, by a scalar. This computation is highly regular, performs one ladder step per scalar bit, and is relatively easy to protect against timing attacks; 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 largely determined by the performance of arithmetic operations in this field.

**Algorithm 1 Curve25519 Montgomery Ladder**

Input: scalar $k$, and $x$-coordinate $x_P$ of a point $P$ on $E$.

Output: $(X_kP, Z_kP)$ fulfilling $x_kP = X_kP/Z_kP$

$t = \lceil \log_2 k + 1 \rceil$

$X_1 = x_P; X_2 = 1; Z_2 = 0; X_3 = x_P; Z_3 = 1$

for $i \leftarrow t - 1$ downto 0 do

if bit $i$ of $k$ is 1 then

$(X_3, Z_3, X_2, Z_2) \leftarrow$ LADDERSTEP$(X_1, X_3, Z_3, X_2, Z_2)$

else

$(X_2, Z_2, X_3, Z_3) \leftarrow$ LADDERSTEP$(X_1, X_2, Z_2, X_3, Z_3)$

end if

end for

return $(X_2, Z_2)$

**Algorithm 2 Single Curve25519 Montgomery Ladderstep**

function LADDERSTEP$(X_1, X_2, Z_2, X_3, Z_3)$

$T_1 \leftarrow X_2 + Z_2$

$T_2 \leftarrow X_1 - Z_2$

$T_7 \leftarrow T_2^2$

$T_6 \leftarrow T_7^2$

$T_5 \leftarrow T_6 - T_7$

$T_8 \leftarrow T_6 \cdot T_7$

$T_9 \leftarrow X_3 + Z_3$

$T_4 \leftarrow X_1 - Z_3$

$T_2 \leftarrow T_3 \cdot T_2$

$T_9 \leftarrow T_3 \cdot T_2$

$T_6 \leftarrow T_6 \cdot T_1$

return $(X_2, Z_2, X_3, Z_3)$

end function

The biggest difference between the two Curve25519 implementations of Bernstein et al. presented in [13, 14] is the representation of elements of $\mathbb{F}_{2^{255}−19}$. Both implementations have the core part, the Montgomery ladder step, in fully inlined, hand-optimized assembly. These core parts are what we target for verification in this paper.

3. ARITHMETIC IN $\mathbb{F}_{2^{255}−19}$ FOR AMD64

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 rdx (higher half) and rax (lower half).
Section 3 of [13] describes two different approaches to implement \( F_p \) arithmetic in AMD64 assembly. Both approaches use the 64-bit integer machine instructions. They are different in the representation of elements of \( F_p \), 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 Arithmetic in radix-\(2^{64}\) representation

The obvious representation of an element \( X \in F_p \) (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 \( F_p \) operation in the Montgomery ladder step, which is multiplication. Squaring is like multiplying, except that some partial results are known to be the same and computed only once. Addition and subtraction are straight forward and multiplication by a small constant simply foregoes computation of results known to be zero. Multiplication in \( F_p \) 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. Only the very final result of the Curve25519 computation has to be fully reduced modulo \(2^{255} - 19\).

#### Multiplication of 256-bit integers

The approach for multiplication in radix-\(2^{64}\) chosen by [13] is a simple textbook 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:

\[
A_0 = Xy_0,
A_1 = 2^{64}Xy_1 + A_0,
A_2 = 2^{128}Xy_2 + A_1,
S = A_3 = 2^{192}Xy_3 + A_2.
\]

Each step essentially computes and accumulates the 5-limb partial product \( XY \) with 4 64×64-bit multiplications and several additions as \((x_0y_0 + 2^{64}x_1y_1 + 2^{128}x_2y_2 + 2^{192}x_3y_3)\). Note that “multiplications by \(2^{64}\)” are free and only determine where to add when summing 128-bit products. For example, the result of \( x_0y_0 \) is in two 64-bit registers \( t_0 \) and \( t_1 \) with \( x_0y_0 = \frac{t_0}{t_1} \), therefore \( t_1 \) needs to be added to the lower result register of \( x_1y_1 \) which in turn produces a carry bit which must go into the register holding the higher half of \( x_1y_1 \). 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 [13]. Sec. 6 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 \( S = (s_0, \ldots, s_7) \). As \(2^{256} = 2p + 38\), the [13] code repeatedly reduces modulo \( 2p \) to fit the result into 256 bits. The reduction begins by computing

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

with a 5-limb result \( S' = (s'_0 + 2^{64}s'_1 + 2^{128}s'_2 + 2^{192}s'_3 + 2^{256}s'_4) \) 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 Arithmetic in radix-\(2^{51}\) representation

Due to the performance penalties in handling carries, [13] proposes to represent elements of \( F_p \) in radix \(2^{51}\), i.e., \( X \in F_p \) is represented by 5 limbs \((x_0, \ldots, x_4)\) as

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

Every element of \( F_p \) 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 \([0, 2^{51} + 2^{15}]\). For the inputs to the first iteration this is ensured by \( C \) code which is not part of the verification, but which has been extensively tested. The inputs of all other iterations are outputs of the previous iteration, so we verify that this property holds for outputs of the ladder step. 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, all limbs of the inputs of the subtraction \( T_2 \leftarrow X_2 - Z_2 \) are in \([0, 2^{51} + 2^{15}]\) (see above). Subtraction is performed by first adding \(0xF F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F
registers, with
\[ t_0 := x_0y_0 + x_1(19y_1) + x_2(19y_2) + x_3(19y_3) + x_4(19y_4), \]
\[ t_1 := x_0y_1 + x_1y_0 + x_2(19y_1) + x_3(19y_2) + x_4(19y_4), \]
\[ t_2 := x_0y_2 + x_1y_1 + x_2y_0 + x_3(19y_2) + x_4(19y_3), \]
\[ t_3 := x_0y_3 + x_1y_2 + x_2y_1 + x_3y_0 + x_4(19y_4), \]
\[ t_4 := x_0y_4 + x_1y_3 + x_2y_2 + x_3y_1 + x_4y_0. \]
(2)

All partial results in this computation are significantly smaller than 128 bits. For example, when \(0 \leq x, y < 2^{54}\) (input limbs are at most 54-bits), \(0 \leq t_0, t_1, t_2, t_3, 10t_4 < 95, 2^{108} < 2^{115}\). Accumulation of each multiplication result can thus be achieved by two 64-bit adds (one ADD, one ADC carry) where all carries are absorbed in the “free” bits in each \(t_i^{(b)}\).

Now \(X \cdot Y = T = t_0 + 2^{51}t_1 + 2^{102}t_2 + 2^{153}t_3 + 2^{204}t_4\), but the two-register values \(t_i\) are still much too large to be used in subsequent operations and need to be carried.

**Delayed carry.** Carrying from the 2-register value \(t_i\) to \(t_{i+1}\) is done as follows: Shift \(t_i^{(b)}\) to the left by 13 and shift the 13 high bits of \(t_i^{(l)}\) into the 13 low bits of the result (which is achieved in just one SHLD instruction). Now set the high 13 bits of \(t_i^{(l)}\) to zero (logical AND with \(2^{51} - 1\)). Now do the same shift-by-13 operation on \(t_i^{(b)}\) and \(t_i^{(l)}\), set the high 13 bits of \(t_i^{(l)}\) to zero, add \(t_i^{(b)}\) to \(t_i^{(l)}\), and discard \(t_i^{(h)}\). This carry chain is performed from \(t_0\) through \(t_4\); then \(t_4^{(h)}\) is multiplied by 19 (using a single-word MUL) and added to \(t_0\).

Note: To avoid losing bits in the shift-by-13, \(t_0^{(h)}, t_1^{(h)}, t_2^{(h)}, t_3^{(h)}, \) and \(19t_4^{(h)}\) (note the the multiply by 19 at the end) must all be at most 51 bits (that is, < \(2^{51}\) before that carrying begins. This condition is met, guaranteeing no overflows, if limbs of X and Y are at most 54 bits as noted above.

This first step of carrying yields \(XY = t_0^{(l)} + 2^{51}t_1^{(l)} + 2^{102}t_2^{(l)} + 2^{153}t_3^{(l)} + 2^{204}t_4^{(l)}\), but the values in \(t_i^{(l)}\) may still be too big as subsequent operands.

The second round of carries starts by copying \(t_0^{(l)}\) to a register \(t\), shifts \(t\) to the right by 51, adds \(t\) to \(t_0^{(l)}\) and discards the upper 13 bits of \(t_0^{(l)}\). Carrying continues this way from \(t_0^{(l)}\) to \(t_2^{(l)}\), from \(t_2^{(l)}\) to \(t_3^{(l)}\), and from \(t_3^{(l)}\) to \(t_4^{(l)}\). Finally \(t_4^{(l)}\) is reduced in the same way except that 19t is added to \(t_0^{(l)}\). The final result is thus obtained in

\[ R = (t_0^{(l)} + 2^{51}t_1^{(l)} + 2^{102}t_2^{(l)} + 2^{153}t_3^{(l)} + 2^{204}t_4^{(l)}). \]

4. BACKGROUND

Recall that in our approach, a qasm program annotated with input assumptions and expected properties is split into smaller programs with their own input assumptions and required properties. Such splits are based on compositional reasoning in Hoare logic with the help of midconditions. These smaller annotated qasm programs are then translated to SMT formulas and verified by the SMT solver Boolector. For algebraic properties such as modular congruence that are hard for SMT solvers, we rely on the proof assistant Coq. In the following of this section, we describe some background about qasm, Hoare logic, Boolector, and Coq.

4.1 Portable assembly: qasm

The software we are verifying has not been written directly in AMD64 assembly, but in the portable assembly language qasm developed by Bernstein [9]. The aim of qasm 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 qasm translates to exactly one assembly instruction. Also, qasm guarantees that “register variables” are indeed kept in registers. Spilling to memory has to be done explicitly by the programmer.

Verifying qasm code. The Curve25519 software we verified is publicly available as part of the SUPERCOP benchmarking framework [10], but does not include the qasm source files, which we obtained from the authors. Our verification works on qasm level. The obvious disadvantage is that we rely on the correctness of qasm translation. The advantage of this approach is that we can easily adapt our approach to assembly software for other architectures. In the future, we plan to also formally verify the qasm to assembly translation to provide stronger correctness guarantees.

4.2 Hoare Logic

Hoare logic [28] is a system for proving the correctness of imperative sequential programs. It contains axioms and inference rules used to establish a valid Hoare triple \((P)C(Q)\) where \(P\) and \(Q\) are formulas in predicate logic and \(C\) is a program. The Hoare triple \((P)C(Q)\) is valid if the program \(C\) starts in a state satisfying \(P\) that provides that \(Q\) starts in a state satisfying \(P\). The formula \(P\) and \(Q\) are called pre- and postconditions respectively. For example, a program \(C\) that increments the value of a variable \(x\) by 1 can be specified by the Hoare triple \((x = a)C(x = a + 1)\) where \(a\) is a logical variable that captures the value of \(x\) before \(C\). Logical variables must not appear in programs and are used only for reasoning. We write \(\vdash (P)C(Q)\) if the Hoare triple \((P)C(Q)\) is valid and write \(\vdash (P)C(Q)\) if the Hoare triple is proven in Hoare logic. The axioms and inference rules of Hoare logic guarantee that \(\vdash (P)C(Q)\) if \(\vdash (P)C(Q)\).

Among the inference rules of Hoare logic, there is a rule called Composition:

\[ \vdash (P)C_0(R) \vdash (R)C_1(Q) \quad (\text{Composition}) \]

With the rule Composition, to prove \(\vdash (P)C_0(R)\vdash (R)C_1(Q)\), it suffices to prove both \(\vdash (P)C_0(R)\) and \(\vdash (R)C_1(Q)\) where \(R\) is a midcondition. Sometimes we may not just find a single midcondition in the middle. In this case, we can use the following relaxed version of Composition:

\[ \vdash (P)C_0(R)\rightarrow S \vdash (S)C_1(Q) \quad (\text{RelaxedComposition}) \]

In rule RelaxedComposition, \(R \rightarrow S\) is a logic implication.

4.3 The Boolector SMT solver

Boolector is an efficient SMT solver supporting the theories of bit vectors and arrays [23]. To be brief, an instance of the SMT problem can be viewed as an instance of the Boolean satisfiability (SAT) problem where a Boolean variable corresponds to a predicate from some background theory. For example, \(f(x, y) = g(z) \land z = x + y\) is an SMT formula where \(f(x, y) = g(z)\) is a predicate from the theory of equality and uninterpreted functions while \(z = x + y\) is
a predicate from the theory of integers. This SMT formula can be viewed as a Boolean formula \( a \land b \) where (1) \( a \) is true iff \( f(x, y) = g(z) \) holds and (2) \( b \) is true iff \( z = x + y \) holds.

In cryptographic software, arithmetic in large finite fields requires hundreds of significant bits. Standard algorithms for linear (integer) arithmetic do not apply. BOOLECTOR reduces queries to instances of the SAT problem by bit blasting and is hence more suitable for our purposes.

Theory of arrays is also essential to the formalization of qhasm programs. In low-level programming languages such as qhasm, memory and pointers are indispensable. We use theory of arrays to model memory in BOOLECTOR. Each qhasm program is of finite length. 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 qhasm programs are necessarily finite, they are expressible in theories of bit vectors and arrays. BOOLECTOR therefore fits perfectly in this application.

4.4 The Coq proof assistant

The Coq proof assistant has been developed in INRIA for more than twenty years [19]. The tool is based on a higher-order logic called the Calculus of Inductive Construction and has lots of libraries for various theories. Theorems are proven with the help of Coq tactics. In contrast to model-theoretic tools such as 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 ring instantaneously.

In this work, we use the Coq standard library Zarith to formalize the congruence relation modulo \(2^{64} - 19\). For non-linear modular relations in \(F_{2^{65} - 19}\), 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 [13, 14]. A ladder step (Alg. 2) consists of 18 field arithmetic operations. Considering the complexity of \(F_p\) multiplication (Sec. 3), the correctness of manually optimized qhasm implementation for the Montgomery ladder step is by no mean clear. The algorithm itself is just a linearization of projective addition. The implementation however changes the order of instructions for efficiency and no longer a linearization of mathematical operations. The correctness of the qhasm implementation is not clear as illustrated by the early incorrect version, which passed extensive tests.

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 qhasm implementations. We only use quantifier-free pre- and postconditions. The typewriter and \&egrave;rafur fonts are used to denote program and logical variables respectively in pre- and postconditions.

Let \( P \) be a qhasm implementation for \( F_p \). Note that \( P \) is loop-free. When the pre- and postconditions are quantifier-free, it is straightforward to translate a Hoare triple \( (Q) P (Q') \) to a BOOLECTOR specification. This specification is equivalent to the quantifier-free SMT formula \( R \land P_{SSA} \land \neg R' \) where \( P_{SSA} \) is the qhasm fragment \( P \) in static single assignment form [5, 35], and \( R \) and \( R' \) are respectively \( Q \) and \( Q' \) with program variables replaced by their indexed version. For example, \( (x = 0) \land r++(x) = r+(x) \) is translated to \( r_0 = 0 \land x_1 = r_0 + 19x_2 \land x_2 = 1 + 19r_0 \land r_2 = 1 + 19y_0 \land r_1 = 19x_0 + 19y_0 \). We then check whether the quantifier-free formula in the theory of bit-vectors is satisfiable. If not, we establish \( \models (Q) P (Q') \). In order to automate this process, we define a simple assertion language to specify pre- and postconditions in qhasm. Moreover, we build a converter that translates annotated qhasm fragments into BOOLECTOR specifications.

5.1 \( F_p \) multiplication in radix-\( 2^{64} \)

Let \( P64 \) denote the qhasm program for \( F_p \) 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 \( xp \) and \( yp \) 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 \((\text{uint64} \ast)(xp + 8i)\) and \((\text{uint64} \ast)(yp + 8i)\) respectively for every \( i \in \{0, 3\} \). We introduce logical variables \( r_i \) and \( \eta_i \) to record the limbs \( x_i \) and \( y_i \) respectively.

Consider

\[
Q_{64_{\text{eqns}}} \ := \ \bigwedge_{i=0}^{3} \left( r_i = (\text{uint64} \ast)(xp + 8i) \land \eta_i = (\text{uint64} \ast)(yp + 8i) \right).
\]

The operator \( \equiv \) denotes the \( n \)-bit equality in the theory of bit-vectors. The formula \( Q_{64_{\text{eqns}}} \) states that the values of the logical variables \( r_i \) and \( \eta_i \) 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 \( rp \). 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 \(((\text{uint64} \ast)(rp + 8i)) \) for every \( i \in \{0, 3\} \). Define

\[
Q_{64_{\text{eqns}}} \ := \ \bigwedge_{i=0}^{3} \left( t_i = 2^{64i} \right) \times \left( \sum_{i=0}^{3} \eta_i 2^{64i} \right) \equiv \sum_{i=0}^{3} t_i 2^{64i} \pmod{p}.
\]

The operator \( \equiv \) denotes the \( n \)-bit signed module operator in the bit-vector theory and the operation \( \times \) is an exact product (without any truncation). The formula \( Q_{64_{\text{eqns}}} \) introduces the logical variable \( t_i \) equal to the limb \( r_i \) for \( 0 \leq i \leq 3 \). The formula \( Q_{64_{\text{prod}}} \) specifies that the outcome \( R \) is indeed the product of \( X \) and \( Y \) in field arithmetic.

Consider the top-level Hoare triple

\[
(\langle Q_{64_{\text{eqns}}} \rangle P64 \langle Q_{64_{\text{eqns}}} \land Q_{64_{\text{prod}}} \rangle).
\]

We are concerned about the outcomes of the qhasm fragment \( P64 \) from states where logical variables \( r_i \) and \( \eta_i \) are equal to limbs of the inputs pointed to by the program variables \( xp \) and \( yp \) 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 \( r_i \), \( \eta_i \) in the precondition \( Q_{64_{\text{eqns}}} \) effectively memorize the input limbs before the execution of \( P64 \). The postcondition \( Q_{64_{\text{eqns}}} \land Q_{64_{\text{prod}}} \) furthermore specifies that the outcome pointed to by the program variable \( rp \) is the product of the inputs stored in \( r_i \) and \( \eta_i \). In other words,
the top-level Hoare triple specifies that the qhasm fragment
$P64\in\mathbb{F}_{p}$ 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 qhasm implementation of $\mathbb{F}_{p}$ multiplication, we exploit the compositionality of proofs for sequential programs. Applying the rule Composition, it suffices to find a midcondition 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.

**Multiplication in radix-$2^{64}$ representation.** Let $P64_M$ and $P64_R$ denote the 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 qhasm variable $sp$. Thus each 64-bit value $s_i$ corresponds to the qhasm expression $\texttt{*(uint64 *)(sp + 8i)}$ for every $i \in [0, 3]$. Define

$$Q64_{mult} := \sum_{i=0}^{7} s_i Q64_{eqn}^i (sp + 8i),$$

$$Q64_{eqn} := \begin{cases} \mathcal{L}_0 s_0 \mathcal{L}_1 s_1 \mathcal{L}_2 s_2 \mathcal{L}_3 s_3, & \text{if } i \in [0, 3] \\ \mathcal{L}_0 s_0 \mathcal{L}_1 s_1 \mathcal{L}_2 s_2 \mathcal{L}_3 s_3, & \text{if } i \in [4, 7] \end{cases}$$

For clarity, we introduce the logical variable $X$ for the input $X = x_0 + 2^{64}x_1 + 2^{128}x_2 + 2^{256}x_3$ in $Q64_{mult}$. Consider the Hoare triple $\{(Q64_{eqn}) P64_M (Q64_{eqn} \land Q64_{mult})\}$. The precondition $Q64_{eqn}$ memorizes the limbs of the inputs $X$ and $Y$ in logical variables $r_i$’s and $t_i$’s. The formula $Q64_{eqn}$ records the limbs $s_i$’s after the qhasm fragment $P64_M$. Logical variables $s_i$’s in $Q64_{eqn}$ 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.

**Reduction in radix-$2^{64}$ representation.** Following the reduction phase in Sec. 3.1, we introduce logical variables $s_i'$ and $s_i''$ for the limbs $s_i$’s and $s_i$’s respectively for every $i \in [0, 4]$. The formulas $Q64_{tred}$, $Q64_{sred}$, $\texttt{Q64}_{tred}$, and $\texttt{Q64}_{sred}$ are defined for the three reduction steps. The formulas $Q64_{s\_eqn}$, $\texttt{Q64}_{s\_eqn}$, and $\texttt{Q64}_{t\_eqn}$ moreover give upper bounds.

$$Q64_{tred} := \sum_{i=0}^{3} s_i' 2^{64i} \geq 320 = s_0 + 2^{64} s_1 + 2^{128} s_2 + 2^{256} s_3 + 38(4 + 2^{64} s_1 + 2^{128} s_2 + 2^{256} s_3)$$

$$Q64_{s\_eqn} := \begin{cases} 0 \leq s_i < 2^{64} & \text{if } i \in [0, 3] \\ 0 \leq s_i' < 2^{64} & \text{if } i \in [4, 7] \end{cases}$$

$$Q64_{sred} := \begin{cases} 0 \leq s_i < 2^{64} & \text{if } i \in [0, 3] \\ 0 \leq s_i'' < 2 & \text{if } i \in [4, 7] \end{cases}$$

$$Q64_{tred} := \begin{cases} 0 \leq t_i < 2^{64} & \text{if } i \in [0, 3] \\ 0 \leq t_i' < 2^{64} & \text{if } i \in [4, 7] \end{cases}$$

Consider the following Hoare triple

$$\{Q64_{mult}\} P64_R \left(\begin{array}{l} Q64_{t\_red} \land Q64_{s\_eqn} \\ Q64_{s\_red} \land Q64_{t\_eqn} \land Q64_{s\_eqn} \end{array}\right).$$

The precondition $Q64_{mult}$ assumes that variables $s_i$’s are obtained from the multiply phase. Recall the formula $Q64_{t\_eqn}$ defined at the beginning of this subsection. The postcondition states that outcome $r_i$’s are obtained by the reduce phase. Note that the logical variable $s_i''$ for the limb $s_i$ at most 1. We are using Boolector to verify this fact in the reduction phase.

**Proposition 1. Assume**

1. $\{Q64_{eqn} \land Q64_{mult}\}$

2. $\{Q64_{mult}\} P64_R \left(\begin{array}{l} Q64_{s\_red} \land Q64_{t\_eqn} \land Q64_{s\_eqn} \land Q64_{t\_red} \land Q64_{s\_eqn} \land Q64_{t\_red} \end{array}\right).$

Then $\{Q64_{eqn} \land Q64_{mult}\} P64_M \land P64_R \{Q64_{prod} \land Q64_{t\_eqn}\}$.

Note that the Hoare triples in the proposition do not establish $Q64_{prod}$ directly. Indeed, we need to show

$Q64_{mult} \land Q64_{t\_red} \land Q64_{s\_eqn} \land Q64_{s\_red} \land Q64_{t\_red} \land Q64_{s\_eqn} \land Q64_{t\_red} \land Q64_{s\_eqn} \implies Q64_{prod}$

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 $38 \equiv 2^{56} \pmod{p}$, our formal Coq proof needs less than 800 lines.

**5.2 $\mathbb{F}_{p}$ multiplication in radix-$2^{51}$**

Let $P51$ denote the qhasm fragment for $\mathbb{F}_{p}$ multiplication in radix-$2^{51}$ representation. The inputs $X = x_0 + 2^{31}x_1 + 2^{62}x_2 + 2^{125}x_3 + 2^{250}x_4, Y = y_0 + 2^{31}y_1 + 2^{62}y_2 + 2^{125}y_3 + 2^{250}y_4$, and outcome $R = r_0 + 2^{31}r_1 + 2^{62}r_2 + 2^{125}r_3 + 2^{250}r_4$ are stored in memory pointed to by the qhasms variables $xp$, $yp$, and $rp$ respectively. We thus introduce logical variables $r_i$, $\eta$, and $t_i$ to memorize the values of the qhasm expressions $\texttt{*(uint64 *)(xp + 8i)}$, $\texttt{*(uint64 *)(yp + 8i)}$, and $\texttt{*(uint64 *)(rp + 8i)}$ respectively for every $i \in [0, 4]$. The formulas $Q51_{eqn}$, $Q51_{t\_eqn}$, and $Q51_{prod}$ are defined similarly as in the radix-$2^{64}$ representation. The formulas $Q51_{eqn}$, $Q51_{t\_eqn}$, and $Q51_{prod}$ specify that the inputs and outcome are in the radix-$2^{51}$ representation.

$$Q51_{eqn} := \begin{cases} 0 \leq r_i < 2^{51} & \text{if } i \in [0, 3] \\ 0 \leq r_i < 2^{51} & \text{if } i \in [4, 7] \end{cases}$$

$$Q51_{t\_eqn} := \begin{cases} 0 \leq r_i < 2^{51} & \text{if } i \in [0, 3] \\ 0 \leq r_i < 2^{51} & \text{if } i \in [4, 7] \end{cases}$$

In the top-level Hoare triple

$$\{Q51_{eqn} \land Q51_{t\_eqn}\} P51 \left(\begin{array}{l} Q51_{t\_eqn} \land Q51_{t\_eqn} \land Q51_{t\_eqn} \land Q51_{t\_eqn} \end{array}\right),$$

the precondition $Q51_{eqn} \land Q51_{t\_eqn}$ assumes that the inputs $X$ and $Y$ are in the radix-$2^{51}$ representation. The postcondition $Q51_{t\_eqn} \land Q51_{t\_eqn} \land Q51_{t\_eqn}$ 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 qhasm fragment $P51$ is $\mathbb{F}_p$ multiplication in the radix-2$^{51}$ representation.

Similar to the case in the radix-2$^{254}$ representation, the top-level Hoare triple should be decomposed before verification. Recall that $\mathbb{F}_p$ multiplication in the radix-2$^{51}$ representation has two phases: multiply-and-reduce and delayed carry (Sec. 3.2). We therefore verify each phase separately.

Let $P51_{MR}$ and $P51_D$ denote the qhasm fragment for multiply-and-reduce and delayed carry respectively. In the multiply-and-reduce phase, the qhasm fragment $P51_{MR}$ computes $s_i$'s in (2). Since each $s_i$ has 128 significant bits, $P51_{MR}$ actually stores each $s_i$ in a pair of 64-bit qhasm variables $s_i,l$ and $s_i,h$. We will use the qhasm expression $u.v$ to denote $u \times 2^{64} + v$. Define

$$Q51_{\text{eqns}} := \bigwedge_{i=0}^{4} s_i \overset{128}{=} s_i.h.s_i.l$$
$$Q51_{\text{mult-red}} := \bigwedge_{i=0}^{4} Q51_{s_i}$$

where

$$Q51_{s_0} := s_0 \overset{128}{=} (f_0y_0 + 19(f_0y_2 + f_2+y_3 + f_3y_2 + f_4y_1))$$
$$Q51_{s_1} := s_1 \overset{128}{=} (f_0y_1 + f_1y_0 + 19(f_2y_4 + f_3y_3 + f_4y_2))$$
$$Q51_{s_2} := s_2 \overset{128}{=} (f_0y_2 + f_1y_1 + f_2y_0 + 19(f_3y_4 + f_4y_3))$$
$$Q51_{s_3} := s_3 \overset{128}{=} (f_0y_3 + f_1y_2 + f_2y_1 + f_3y_0 + 19f_4y_4)$$
$$Q51_{s_4} := s_4 \overset{128}{=} (f_0y_4 + f_1y_3 + f_2y_2 + f_3y_1 + f_4y_0).$$

$Q51_{\text{eqns}}$ states that the logical variable $s_i$ is equal to the qhasm expression $s_i.h.s_i.l$ for every $i \in [0, 4]$. $Q51_{\text{mult-red}}$ specifies that $s_i$ are computed correctly for every $i \in [0, 4]$. Nonetheless, we find the condition $Q51_{\text{mult-red}}$ is too weak to prove the correctness of the multiply-and-reduce phase. If $s.h.s.l$ 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 $2^{102}$. A simple computation gives us better upper bounds for $s_i$'s. Define

$$Q51_{\text{lds}} := \begin{cases} 0 & \leq s_0 \leq 2^{102} + 4 \cdot 19 \cdot 2^{102} \\ 0 & \leq s_1 \leq 2 \cdot 2^{102} + 3 \cdot 19 \cdot 2^{102} \\ 0 & \leq s_2 \leq 3 \cdot 2^{102} + 2 \cdot 19 \cdot 2^{102} \\ 0 & \leq s_3 \leq 4 \cdot 2^{102} + 19 \cdot 2^{102} \\ 0 & \leq s_4 \leq 5 \cdot 2^{102} \end{cases}$$

Consider the Hoare triple $\langle Q51_{\text{eqns}} \land Q51_{\text{lds}} \rangle \vdash P51_{MR}$ $\langle Q51_{\text{eqns}} \land Q51_{\text{lds}} \land Q51_{\text{mult-red}} \rangle$, in addition to checking whether qhasm variables $s_i.h,s_i.l$'s and $s_i.l$'s are computed correctly, the qhasm fragment $P51_{MR}$ for multiply-reduce is required to meet the upper bounds in $Q51_{s_i,l}$. The midcondition $Q51_{s_i,l} \land Q51_{s_i,r}$ enables the verification of the qhasm fragment $P51_D$ for the delayed carry phase.

The qhasm fragment $P51_{D}$ for delayed carry performs carrying on 128-bit expressions $s_i.h,s_i.l$'s to obtain the product of the inputs $X$ and $Y$. The product must also be in the radix-2$^{51}$ representation. Define

$$Q51_{\text{delayed-carry}} := \bigwedge_{i=0}^{4} s_i \overset{2^{51}}{=} \bigoplus_{i=0}^{4} r_i \overset{2^{51}}{=} (\text{mod } p).$$

The Hoare triple $\langle Q51_{\text{eqns}} \land Q51_{\text{lds}} \land Q51_{\text{mult-red}} \rangle \vdash P51_D$ $\langle Q51_{\text{delayed-carry}} \land Q51_{r_i,\text{ld}} \rangle$ verifies that the qhasm fragment $P51_D$ computes a number $\sum_{i=0}^{4} r_i \overset{2^{51}}{=} $ in the radix-2$^{51}$ representation, and it is congruent to $\sum_{i=0}^{4} s_i \overset{2^{51}}{=} $ modulo $p$. By their corresponding pre- and postconditions with appropriate variable renaming. Alg. 2 is then converted to the static single assignment form. A formula associating variables in Alg. 2 and corresponding variables in the actual implementation is also added. We then assert the static single assignments as the postcondition of Alg. 2. Using Boolector, we verify that the postcondition holds, and that the postcondition of every field operation implies the precondition of the following field operation in Alg. 2. By the rule RelaxedComposition, the record-holding implementation for the Montgomery ladder step in the radix-2$^{254}$ and radix-2$^{51}$ representations are formally verified, that is, the implementation indeed matches Alg. 2.

We did not verify the Montgomery ladder step with a big annotated qhasm program due to an efficiency consideration. For example, multiplication of two variables is performed five times in Alg. 2. The codes for these multiplication operations are essentially identical with one-to-one and onto variable renaming. Thus, if we verify the Montgomery ladder step as a whole, we will waste time on verifying the same code. Note that the same code with different pre- and postconditions is still needed to be verified separately.

## 5.3 Montgomery ladder step

The verification of the Montgomery ladder step (Alg. 2) is carried out after all implementations of used field arithmetic operations are verified separately. We replace fragments for field arithmetic in the Montgomery ladder step by their corresponding pre- and postconditions with appropriate variable renaming. Alg. 2 is then converted to the static single assignment form. A formula associating variables in Alg. 2 and corresponding variables in the actual implementation is also added. We then assert the static single assignments as the postcondition of Alg. 2. Using Boolector, we verify that the postcondition holds, and that the postcondition of every field operation implies the precondition of the following field operation in Alg. 2. By the rule RelaxedComposition, the record-holding implementation for the Montgomery ladder step in the radix-2$^{254}$ and radix-2$^{51}$ representations are formally verified, that is, the implementation indeed matches Alg. 2.

We did not verify the Montgomery ladder step with a big annotated qhasm program due to an efficiency consideration. For example, multiplication of two variables is performed five times in Alg. 2. The codes for these multiplication operations are essentially identical with one-to-one and onto variable renaming. Thus, if we verify the Montgomery ladder step as a whole, we will waste time on verifying the same code. Note that the same code with different pre- and postconditions is still needed to be verified separately.

## 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 midconditions used, and the verification time used in Boolector. We run 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. All the results in Table 1 are sufficient to verify the qhasm code, not the best.

We formally verified the ladder step in Algorithm 2 in both radix-2$^{254}$ and radix-2$^{51}$. The pre- and postconditions of each operators are obtained from the verification of the
corresponding qhasm code fe25519r64*/fe25519*. We are able to reproduce a known bug in an old version of \( \mathbb{F}_{2^{255} - 19} \) multiplication (fe25519r64_mul-1). A counterexample can be found in seconds with a pair of precondition and postcondition for the reduction phase. Verification time of squaring is less than that of multiplication because (1) squaring is simpler than multiplication which requires more low-level multiplication operations, and (2) multiplication is verified without the fourth heuristic to be introduced later in this section, but squaring is verified with the heuristic.

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 multiplication. It shows that if we add an additional midcondition to the delayed carry phase of mul25519, the verification time of the delayed carry phase can be reduced from 2723 minutes to 263 minutes. In general, inserting more midconditions allows lower verification time, with a cost of more manual efforts. Besides mul25519 and qhasm code in the ladder step, we also successfully verified (1) a 3-phase implementation of multiplication with addition (muladd25519), and (2) implementations of multiplication over different finite fields (fe19119_mul, mul1271, and re15319).

Note that all postconditions for the radix-2^4 are equalities. Since BOOLECTOR can not verify modular congruence relations in the radix-2^4 representation, we have to establish them in Coq. On the other hand, BOOLECTOR successfully verifies the modular congruence relation \( Q_5 \) for the radix-2^8 representation. Our Coq proof for the radix-2^4 representation is thus simplified. The reason why some congruence relations is verified in the radix-2^4 representation is because we are able to divide P51D 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 conjunctions of postconditions, i.e., translate \( \langle Q_0 \rangle P \langle Q_1 \land Q_2 \rangle \) to \( \langle Q_0 \rangle P \langle Q_1 \rangle \) and \( \langle Q_0 \rangle P \langle Q_2 \rangle \). This reduces the verification time of the multiply phase of mul25519 from one day to one minute. Second, we delay bit-width extension. For example, consider a formula \( a = b \cdot c \) 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, the sequence of mathematical operations in annotations should match as much as possible the sequence of operations executed in a program. For example, if a program calculates the value of a variable \( r \) by adding \( 19x_0y_2 \) first, then \( 19x_1y_1 \), and finally \( 19x_2y_0 \), the annotation is better written as \( r = (19x_0y_2 + 19x_1y_1) + 19x_2y_0 \) instead of \( r = 19(x_0y_2 + x_1y_1) + x_2y_0 \). If we need to prove \( r = 19(x_0y_2 + x_1y_1) + x_2y_0 \), it can be done in Coq very easily with rewrite tactics given the fact that \( r = (19x_0y_2 + 19x_1y_1) + 19x_2y_0 \). Fourth, we over-approximate BOOLECTOR specifications by automatically reducing logical variables and weakening preconditions such that the specifications become easier to be proven. The validity of an over-approximated specification guarantees the validity of the original one, but not vice versa. This heuristic can be viewed as program slicing with over-approximation. To be more specific, given a specification \( \langle Q_0 \rangle \land \langle Q_2 \rangle P \langle Q_1 \rangle \), 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 \( P \). For example, given a Hoare triple \( (\{ r = t_1 \} \land \{ t_0 = x + x \leq 2^{23} \} \) \) \( x \mapsto y \langle \{ r = t_1 + y \rangle \), this heuristic produces \( \{ r = t_1 \} \) \( x \mapsto y \langle \{ r = t_1 + y \rangle \) where \( 1 \) \( t_1 = t_0 + x \) is removed because the logical variable \( t_0 \) does not appear in the postcondition, and \( 2 \) \( x \leq 2^{23} \) is removed.

<table>
<thead>
<tr>
<th>File Name</th>
<th>Description</th>
<th># of limb</th>
<th># of MC</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>fe25519r64_mul-1</td>
<td>( r = x \cdot y (mod 2^{255} - 19) ), a buggy version</td>
<td>4</td>
<td>1</td>
<td>0m8.73s</td>
</tr>
<tr>
<td>fe25519r64_add</td>
<td>( r = x + y (mod 2^{255} - 19) )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fe25519r64_sub</td>
<td>( r = x - y (mod 2^{255} - 19) )</td>
<td>4</td>
<td>0</td>
<td>0m3.15s</td>
</tr>
<tr>
<td>fe25519r64_mul-2</td>
<td>( r = x \cdot y (mod 2^{255} - 19) ), a fixed version of fe25519r64_mul-1</td>
<td>4</td>
<td>19</td>
<td>73m55.16s</td>
</tr>
<tr>
<td>fc25519r64_mul121666</td>
<td>( r = x \cdot y (mod 2^{255} - 19) )</td>
<td>4</td>
<td>2</td>
<td>0m2.03s</td>
</tr>
<tr>
<td>fe25519r64_mul</td>
<td>( r = x \cdot y (mod 2^{255} - 19) )</td>
<td>4</td>
<td>5</td>
<td>3m16.67s</td>
</tr>
<tr>
<td>ladderstep64</td>
<td>The implementation of Algorithm 2</td>
<td>4</td>
<td>14</td>
<td>0m3.23s</td>
</tr>
<tr>
<td>fe19119_mul</td>
<td>( r = x \cdot y (mod 2^{191} - 19) )</td>
<td>3</td>
<td>12</td>
<td>8m43.07s</td>
</tr>
<tr>
<td>mul1271</td>
<td>( r = x \cdot y (mod 2^{127} - 1) )</td>
<td>2</td>
<td>1</td>
<td>141m22.06s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>File Name</th>
<th>Description</th>
<th># of limb</th>
<th># of MC</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>fe25519r64_add</td>
<td>( r = x + y (mod 2^{255} - 19) )</td>
<td>5</td>
<td>0</td>
<td>0m16.35s</td>
</tr>
<tr>
<td>fe25519r64_sub</td>
<td>( r = x - y (mod 2^{255} - 19) )</td>
<td>5</td>
<td>0</td>
<td>3m38.62s</td>
</tr>
<tr>
<td>fe25519r64_mul</td>
<td>( r = x \cdot y (mod 2^{255} - 19) )</td>
<td>5</td>
<td>27</td>
<td>565m51.15s</td>
</tr>
<tr>
<td>fe25519r64_sq</td>
<td>( r = x \cdot y (mod 2^{255} - 19) )</td>
<td>5</td>
<td>5</td>
<td>0m12.75s</td>
</tr>
<tr>
<td>laddersq9</td>
<td>The implementation of Algorithm 2</td>
<td>5</td>
<td>17</td>
<td>463m59.5s</td>
</tr>
<tr>
<td>mul25519</td>
<td>( r = x \cdot y (mod 2^{255} - 19) ), a 3-phase implementation</td>
<td>5</td>
<td>14</td>
<td>1m29.05s</td>
</tr>
<tr>
<td>mul25519-p2-1</td>
<td>The delayed carry phase of ( r = x \cdot y (mod 2^{255} - 19) )</td>
<td>5</td>
<td>3</td>
<td>286m52.75s</td>
</tr>
<tr>
<td>mul25519-p2-2</td>
<td>The delayed carry phase of ( r = x \cdot y (mod 2^{255} - 19) ) with two sub-phases</td>
<td>5</td>
<td>7</td>
<td>263m35.40s</td>
</tr>
<tr>
<td>muladd25519</td>
<td>( r = x \cdot y + z (mod 2^{255} - 19) )</td>
<td>5</td>
<td>1</td>
<td>272m31.56s</td>
</tr>
<tr>
<td>re15319</td>
<td>( r = x \cdot y (mod 2^{153} - 19) )</td>
<td>3</td>
<td>3</td>
<td>2409m16.89s</td>
</tr>
</tbody>
</table>
because \( x \) neither gets updated nor appears in the postcondition. Traditional program slicing may not remove \( r_x \) and \( x \) because both variables are related to \( r_1 \), which appears in the postcondition. We cannot verify fe25519r64 sq and fe25519r 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 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 midconditions 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 assertion insertion. We think the tools for automatic assertion insertion could be relevant [25]. The tool obtains assertions based on given templates of assertions and by synthesizing them dynamically from observed executions traces.

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

8. REFERENCES


[18] D. J. Bernstein, T. Lange, and P. Schwabe. The security impact of a new cryptographic library. In