Verifying Curve25519 Software

Yu-Fang Chen[1]
yfc@iis.sinica.edu.tw

Chang-Hong Hsu[2]
hsuch@umich.edu

Hsin-Hung Lin[3]
h-lin@ait.kyushu-u.ac.jp

Peter Schwabe[4]
peter@cryptojedi.org

Ming-Hsien Tsai[1]
mhtsai208@gmail.com

Bow-Yaw Wang[1]
bywang@iis.sinica.edu.tw

Bo-Yin Yang[1]
by@crypto.tw

Shang-Yi Yang[1]
ilway25@crypto.tw

[1]Institute of Information Science, Academia Sinica, Taiwan
[2]University of Michigan, Ann Arbor, USA
[3]Faculty of Information Science and Electrical Engineering, Kyushu University, Japan
[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 inline 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 `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 `qhasm` code. The `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[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

Public domain. This work was supported by the Ministry of Science and Technology (MOST), Taiwan, through projects no 102-2221-E-001-017/018, and 103-2221-E-001-019/020-MY3; by the Netherlands Organisation for Scientific Research (NWO) through Veni 2013 project 13114; and by Academia Sinica, through the IIS Specialty Project and BY's Career Advancement Award. Part of this work was done while Schwabe was employed by Academia Sinica. Permanent ID of this document: 55ab8668ce87d857c02a5b2d56d7da38. Date: August 24, 2014.
CCS'14, November 3–7, 2014, Scottsdale, Arizona, USA.
ACM 978-1-4503-2957-6/14/11. http://dx.doi.org/10.1145/2660267.2660370.

[1]This bug is documented, see 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 now 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 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 `qhasm` 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 `qhasm` 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.

[2]Specifically, the scripts http://www.hyperelliptic.org/EFD/g1p/auto-sage/montgom/coordinates.sage and http://www.hyperelliptic.org/EFD/g1p/auto-sage/montgom/xz/ladder/mladd-1987-m.sage

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, Bayrak 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`.

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

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

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 a 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 $k \in Z$ 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 [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 = ⌈log_2 k + 1⌉
X_1 = x_P; X_2 = 1; Z_2 = 0; X_3 = x_P; Z_3 = 1
for i ← t − 1 downto 0 do
    if bit i of k is 1 then
        (X_3, Z_3, X_2, Z_2) ← LADDERSTEP(X_1, X_3, Z_3, X_2, Z_2)
    else
        (X_2, Z_2, X_3, Z_3) ← 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 ← X_2 + Z_2
T_2 ← X_2 − Z_2
T_7 ← T_2^2
T_6 ← T_1^2
T_5 ← T_6 − T_7
T_3 ← X_3 + Z_3
T_4 ← X_3 − Z_3
T_9 ← T_3 · T_2
T_8 ← T_4 · T_1
X_3 ← (T_8 + T_9)
Z_3 ← (T_8 − T_9)
X_3 ← X_3^2
Z_3 ← Z_3^2
Z_3 ← Z_3 · X_1
X_2 ← T_6 · T_7
Z_2 ← 121666 · T_5
Z_2 ← Z_2 + T_7
Z_2 ← Z_2 · T_5
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 $\mathbb{F}_p$ arithmetic in AMD64 assembly. Both approaches use the 64-bit-integer machine instructions. They are different 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 Arithmetic in radix-2^{64} 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 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 $\mathbb{F}_p$ 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. 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 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:

$$\begin{aligned} A_0 &= Xy_0, \\ A_1 &= 2^{64}Xy_1 + A_0, \\ A_2 &= 2^{128}Xy_2 + A_1, \\ S = A_3 &= 2^{192}Xy_3 + A_2. \end{aligned} \tag{1}$$

Each step essentially computes and accumulates the 5-limb partial product Xy_i with 4 64×64-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 [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

$$\begin{aligned} S' \quad = \quad & (s_0 + 2^{64}s_1 + 2^{128}s_2 + 2^{192}s_3) + \\ & 38(s_4 + 2^{64}s_5 + 2^{128}s_6 + 2^{192}s_7) \end{aligned}$$

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 s'_4, the highest limb 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 Arithmetic in radix-2^{51} representation

Due to the performance penalties in handling carries, [13] 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

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

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 $[0, 2^{51} + 2^{15}]$. 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 `0xFFFFFFFFFFFFDA` 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. 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 $19y_1, 19y_2, 19y_3$ and $19y_4$, then compute 5 intermediate values $t_0, \ldots, t_4$, where each $t_i = \left(t_i^{(l)}, t_i^{(h)}\right) := t_i^{(l)} + 2^{64}\ t_i^{(h)}$ is a pair of 64-bit

registers, with

$$
\begin{aligned}
t_0 &:= x_0y_0 + x_1(19y_4) + x_2(19y_3) + x_3(19y_2) + x_4(19y_1),\\
t_1 &:= x_0y_1 + x_1y_0 + x_2(19y_4) + x_3(19y_3) + x_4(19y_2),\\
t_2 &:= x_0y_2 + x_1y_1 + x_2y_0 + x_3(19y_4) + x_4(19y_3),\\
t_3 &:= x_0y_3 + x_1y_2 + x_2y_1 + x_3y_0 + x_4(19y_4),\\
t_4 &:= x_0y_4 + x_1y_3 + x_2y_2 + x_3y_1 + x_4y_0. \qquad (2)
\end{aligned}
$$

All partial results in this computation are significantly smaller than 128 bits. For example, when $0 \le x_i, y_i < 2^{54}$ (input limbs are at most 54-bits), $0 \le t_0, t_1, t_2, t_3, 19t_4 < 95 \cdot 2^{108} < 2^{115}$. 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 $t_i^{(h)}$.

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 $t_i^{(h)}$ 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 $t_{i+1}^{(h)}$ and $t_{i+1}^{(l)}$, set the high 13 bits of $t_{i+1}^{(l)}$ to zero, add $t_i^{(h)}$ to $t_{i+1}^{(l)}$ 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 $t_0^{(l)}$.

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 $t_1^{(l)}$ and discards the upper 13 bits of $t_0^{(l)}$. Carrying continues this way from $t_1^{(l)}$ to $t_2^{(l)}$, from $t_2^{(l)}$ to $t_3^{(l)}$, and from 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 `qhasm` 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 `qhasm` 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 `qhasm`, Hoare logic, Boolector, and Coq.

4.1 Portable assembly: `qhasm`

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

Verifying `qhasm` code. The Curve25519 software we verified is publicly available as part of the SUPERCOP benchmarking framework [10], but does not include the `qhasm` source files, which we obtained from the authors. Our verification works on `qhasm` level. The obvious disadvantage is that we rely on the correctness of `qhasm` 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 `qhasm` 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 iff the program C ends in a state satisfying Q provided that C 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 $\models (\!|P|\!)\, C\, (\!|Q|\!)$ 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 $\models (\!|P|\!)\, C\, (\!|Q|\!)$ iff $\vdash (\!|P|\!)\, C\, (\!|Q|\!)$.

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

$$\frac{\vdash (\!|P|\!)\, C_0\, (\!|R|\!) \quad \vdash (\!|R|\!)\, C_1\, (\!|Q|\!)}{\vdash (\!|P|\!)\, C_0; C_1\, (\!|Q|\!)}\ \text{Composition}$$

With the rule Composition, to prove $\vdash (\!|P|\!)\, C_0; C_1\, (\!|Q|\!)$, 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:

$$\frac{\vdash (\!|P|\!)\, C_0\, (\!|R|\!) \quad R \to S \quad \vdash (\!|S|\!)\, C_1\, (\!|Q|\!)}{\vdash (\!|P|\!)\, C_0; C_1\, (\!|Q|\!)}\ \text{RelaxedComposition}$$

In rule RelaxedComposition, $R \to S$ 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, $f(x, y) = g(z) \wedge z = x + y$ 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 $a \wedge b$ 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 $2^{255} - 19$. For non-linear modular relations in $\mathbb{F}_{2^{255}-19}$, 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 $\mathbb{F}_p$ 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 $\mathbb{F}_p$ 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 post-conditions. The `typewriter` and $\mathfrak{Fraktur}$ fonts are used to denote program and logical variables respectively in pre- and postconditions.

Let P be a `qhasm` implementation for $\mathbb{F}_p$ multiplication. 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 $R \wedge P_{SSA} \wedge \neg R'$ 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, $(\!|\texttt{r} = 0|\!)$ `r+=19*x; r+=19*y` $(\!|\texttt{r} = 19\texttt{x} + 19\texttt{y}|\!)$ is translated to $\texttt{r}_0 = 0 \wedge \texttt{r}_1 = \texttt{r}_0 + 19\texttt{x}_0 \wedge \texttt{r}_2 = \texttt{r}_1 + 19\texttt{y}_0 \wedge \texttt{r}_2 \neq 19\texttt{x}_0 + 19\texttt{y}_0$. 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 $\mathbb{F}_p$ multiplication in radix-2^{64}

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 `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 `*(uint64 *)(xp + 8i)` and `*(uint64 *)(yp + 8i)` respectively for every $i \in [0, 3]$. We introduce logical variables $\mathfrak{x}_i$ and $\mathfrak{y}_i$ to record the limbs x_i and y_i respectively. Consider

$$Q64_{xy\_eqns} := \bigwedge_{i=0}^{3} \mathfrak{x}_i \overset{64}{=} \texttt{*(uint64 *)(xp + 8}i\texttt{)} \wedge \bigwedge_{i=0}^{3} \mathfrak{y}_i \overset{64}{=} \texttt{*(uint64 *)(yp + 8}i\texttt{)}.$$

The operator $\overset{n}{=}$ denotes the n-bit equality in the theory of bit-vectors. The formula $Q64_{xy\_eqns}$ states that the values of the logical variables $\mathfrak{x}_i$ and $\mathfrak{y}_i$ 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 `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 `*(uint64 *)(rp + 8i)` for every $i \in [0, 3]$. Define

$$Q64_{r\_eqns} := \bigwedge_{i=0}^{3} \mathfrak{r}_i \overset{64}{=} \texttt{*(uint64 *)(rp + 8}i\texttt{)},$$

$$Q64_{prod} := (\sum_{i=0}^{3} \mathfrak{x}_i 2^{64i}) \times (\sum_{i=0}^{3} \mathfrak{y}_i 2^{64i}) \overset{512}{\equiv} \sum_{i=0}^{3} \mathfrak{r}_i 2^{64i} \pmod{p}.$$

The operator $\overset{n}{\equiv}$ denotes the n-bit signed modulo operator in the bit-vector theory and the operation $\times$ is an exact product (without any truncation). The formula $Q64_{r\_eqns}$ introduces the logical variable $\mathfrak{r}_i$ equal to the limb r_i for $0 \leq i \leq 3$. The formula $Q64_{prod}$ specifies that the outcome R is indeed the product of X and Y in field arithmetic.

Consider the top-level Hoare triple

$$(\!|Q64_{xy\_eqns}|\!)\, P64\, (\!|Q64_{r\_eqns} \wedge Q64_{prod}|\!)\,.$$

We are concerned about the outcomes of the `qhasm` fragment $P64$ from states where logical variables $\mathfrak{x}_i$ and $\mathfrak{y}_i$ 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 $\mathfrak{x}_i, \mathfrak{y}_i$ in the precondition $Q64_{xy\_eqns}$ effectively memorize the input limbs before the execution of $P64$. The postcondition $Q64_{r\_eqns} \wedge Q64_{prod}$ furthermore specifies that the outcome pointed to by the program variable `rp` is the product of the inputs stored in $\mathfrak{x}_i$ and $\mathfrak{y}_i$. In other words,

the top-level Hoare triple specifies that the `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 `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 $S = (s0, \ldots, s_7)$ 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 `*(uint64 *)(sp + 8i)` for every $i \in [0, 3]$. Define

$$\begin{aligned}
Q64_{s_eqns} &:= \bigwedge_{i=0}^{7} \mathfrak{s}_i \overset{64}{=} \texttt{*(uint64 *)(sp + 8}i\texttt{)}, \\
Q64_{mult} &:= \mathfrak{A}_0 \overset{512}{=} \mathfrak{X}\mathfrak{y}_0 \wedge \mathfrak{A}_1 \overset{512}{=} 2^{64}\mathfrak{X}\mathfrak{y}_1 + \mathfrak{A}_0 \wedge \\
&\quad \mathfrak{A}_2 \overset{512}{=} 2^{128}\mathfrak{X}\mathfrak{y}_2 + \mathfrak{A}_1 \wedge \\
&\quad \mathfrak{A}_3 \overset{512}{=} 2^{192}\mathfrak{X}\mathfrak{y}_3 + \mathfrak{A}_2 \wedge \\
&\quad \mathfrak{X} \overset{512}{=} \sum_{i=0}^{3} \mathfrak{x}_i 2^{64i} \wedge \sum_{i=0}^{7} \mathfrak{s}_i 2^{64i} \overset{512}{=} \mathfrak{A}_3.
\end{aligned}$$

For clarity, we introduce the logical variable $\mathfrak{X}$ for the input $X = x_0 + 2^{64}x_1 + 2^{128}x_2 + 2^{192}x_3$ in $Q64_{mult}$. Consider the Hoare triple $(\!|Q64_{xy\_eqns}|\!)\, P64_M\, (\!|Q64_{s\_eqns} \wedge Q64_{mult}|\!)$ The precondition $Q64_{xy\_eqns}$ memorizes the limbs of the inputs X and Y in logical variables $\mathfrak{x}_i$'s and $\mathfrak{y}_i$'s. The formula $Q64_{s\_eqns}$ records the limbs s_i's after the `qhasm` fragment $P64_M$ in logical variables $\mathfrak{s}_i$'s. $Q64_{mult}$ 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.

Reduction in radix-2^{64} representation. Following the reduction phase in Sec. 3.1, we introduce logical variables $\mathfrak{s}'_i$ and $\mathfrak{s}''_i$ for the limbs s'_i and s''_i respectively for every $i \in [0, 4]$. The formulas $Q64_{s'\_red}$, $Q64_{s''\_red}$, $Q64_{r\_red}$ are defined for the three reduction steps. The formulas $Q64_{s'\_bds}$, $Q64_{s''\_bds}$, and $Q64_{r\_bds}$ moreover give upper bounds.

$$\begin{aligned}
Q64_{s'_red} &:= \sum_{i=0}^{4} \mathfrak{s}'_i 2^{64i} \overset{320}{=} \mathfrak{s}_0 + 2^{64}\mathfrak{s}_1 + 2^{128}\mathfrak{s}_2 + 2^{192}\mathfrak{s}_3 + \\
&\quad 38(\mathfrak{s}_4 + 2^{64}\mathfrak{s}_5 + 2^{128}\mathfrak{s}_6 + 2^{192}\mathfrak{s}_7) \\
Q64_{s'_bds} &:= \bigwedge_{i=0}^{4} 0 \le \mathfrak{s}'_i < 2^{64} \\
Q64_{s''_red} &:= \sum_{i=0}^{4} \mathfrak{s}''_i 2^{64i} \overset{320}{=} 38\mathfrak{s}'_4 + \sum_{i=0}^{3} \mathfrak{s}'_i 2^{64i} \\
Q64_{s''_bds} &:= \bigwedge_{i=0}^{3} 0 \le \mathfrak{s}''_i < 2 \wedge 0 \le \mathfrak{s}''_4 < 2 \\
Q64_{r_red} &:= \sum_{i=0}^{3} \mathfrak{r}_i 2^{64i} \overset{256}{=} 38\mathfrak{s}''_4 + \sum_{i=0}^{3} \mathfrak{s}''_i 2^{64i} \\
Q64_{r_bds} &:= \bigwedge_{i=0}^{3} 0 \le \mathfrak{r}_i < 2^{64}
\end{aligned}$$

Consider the following Hoare triple

$$(\!|Q64_{mult}|\!)\, P64_R \left(\begin{array}{c} Q64_{s'\_red} \wedge Q64_{s'\_bds} \wedge \\ Q64_{s''\_red} \wedge Q64_{s''\_bds} \wedge \\ Q64_{r\_red} \wedge Q64_{r\_bds} \wedge \\ Q64_{r\_eqns} \end{array} \right).$$

The precondition $Q64_{mult}$ assumes that variables $\mathfrak{s}_i$'s are obtained from the multiply phase. Recall the formula $Q64_{r\_eqns}$ 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 $\mathfrak{s}''_4$ for the limb s''_4 is at most 1. We are using Boolector to verify this fact in the reduction phase.

Proposition 1. *Assume*

1. $\models (\!|Q64_{xy\_eqns}|\!)\, P64_M\, (\!|Q64_{s\_eqns} \wedge Q64_{mult}|\!)$;
2. $\models (\!|Q64_{mult}|\!)\, P64_R \left(\begin{array}{c} Q64_{s'\_red} \wedge Q64_{s'\_bds} \wedge \\ Q64_{s''\_red} \wedge Q64_{s''\_bds} \wedge \\ Q64_{r\_red} \wedge Q64_{r\_bds} \wedge \\ Q64_{r\_eqns} \end{array} \right)$.

Then $\models (\!|Q64_{xy\_eqns}|\!)\, P64_M; P64_R\, (\!|Q64_{prod} \wedge Q64_{r\_bds}|\!)$.

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

$$\begin{array}{l} Q64_{mult} \wedge Q64_{s'\_red} \wedge Q64_{s'\_bds} \wedge \\ Q64_{s''\_red} \wedge Q64_{s''\_bds} \wedge Q64_{r\_red} \wedge Q64_{r\_bds} \end{array} \implies Q64_{prod}$$

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 $38 \equiv 2^{256} \pmod{p}$, 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 $X = x_0 + 2^{51}x_1 + 2^{102}x_2 + 2^{153}x_3 + 2^{204}x_4$, $Y = y_0 + 2^{51}y_1 + 2^{102}y_2 + 2^{153}y_3 + 2^{204}y_4$, 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 `xp`, `yp`, and `rp` respectively. We thus introduce logical variables $\mathfrak{x}_i$, $\mathfrak{y}_i$, and $\mathfrak{r}_i$ to memorize the values of the `qhasm` expressions `*(uint64 *)(xp + 8i)`, `*(uint64 *)(yp + 8i)`, and `*(uint64 *)(rp + 8i)` respectively for every $i \in [0, 4]$.

The formulas $Q51_{xy\_eqns}$, $Q51_{r\_eqns}$, $Q51_{prod}$ are defined similarly as in the radix-2^{64} representation. The formulas $Q51_{xy\_bds}$ and $Q51_{r\_bds}$ specify that the inputs and outcome are in the radix-2^{51} representation.

$$\begin{aligned}
Q51_{xy_bds} &:= \bigwedge_{i=0}^{4} 0 \le \mathfrak{x}_i < 2^{51} \wedge \bigwedge_{i=0}^{4} 0 \le \mathfrak{y}_i < 2^{51} \\
Q51_{r_bds} &:= \bigwedge_{i=0}^{4} 0 \le \mathfrak{r}_i < 2^{51}
\end{aligned}$$

In the top-level Hoare triple

$$(\!|Q51_{xy\_eqns} \wedge Q51_{xy\_bds}|\!)\, P51 \left(\begin{array}{c} Q51_{r\_eqns} \wedge Q51_{r\_bds} \wedge \\ Q51_{prod} \end{array} \right),$$

the precondition $Q51_{xy\_eqns} \wedge Q51_{xy\_bds}$ assumes that the inputs X and Y are in the radix-2^{51} representation. The postcondition $Q51_{r\_eqns} \wedge Q51_{r\_bds} \wedge Q51_{prod}$ 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 qhasm fragment $P51$ 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^{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 $\mathtt{s}_i\mathtt{l}$ and $\mathtt{s}_i\mathtt{h}$. We will use the qhasm expression $u.v$ to denote $u \times 2^{64} + v$. Define

$$\begin{aligned} Q51_{s\_eqns} &:= \bigwedge_{i=0}^{4} \mathfrak{s}_i \overset{128}{=} \mathtt{s}_i\mathtt{h}.\mathtt{s}_i\mathtt{l} \\ Q51_{mult\_red} &:= \bigwedge_{i=0}^{4} Q51_{s_i} \end{aligned}$$

where

$$\begin{aligned} Q51_{s_0} &:= \mathfrak{s}_0 \overset{128}{=} (\mathfrak{r}_0\mathfrak{y}_0 + 19(\mathfrak{r}_1\mathfrak{y}_4 + \mathfrak{r}_2\mathfrak{y}_3 + \mathfrak{r}_3\mathfrak{y}_2 + \mathfrak{r}_4\mathfrak{y}_1)) \\ Q51_{s_1} &:= \mathfrak{s}_1 \overset{128}{=} (\mathfrak{r}_0\mathfrak{y}_1 + \mathfrak{r}_1\mathfrak{y}_0 + 19(\mathfrak{r}_2\mathfrak{y}_4 + \mathfrak{r}_3\mathfrak{y}_3 + \mathfrak{r}_4\mathfrak{y}_2)) \\ Q51_{s_2} &:= \mathfrak{s}_2 \overset{128}{=} (\mathfrak{r}_0\mathfrak{y}_2 + \mathfrak{r}_1\mathfrak{y}_1 + \mathfrak{r}_2\mathfrak{y}_0 + 19(\mathfrak{r}_3\mathfrak{y}_4 + \mathfrak{r}_4\mathfrak{y}_3)) \\ Q51_{s_3} &:= \mathfrak{s}_3 \overset{128}{=} (\mathfrak{r}_0\mathfrak{y}_3 + \mathfrak{r}_1\mathfrak{y}_2 + \mathfrak{r}_2\mathfrak{y}_1 + \mathfrak{r}_3\mathfrak{y}_0 + 19\mathfrak{r}_4\mathfrak{y}_4) \\ Q51_{s_4} &:= \mathfrak{s}_4 \overset{128}{=} (\mathfrak{r}_0\mathfrak{y}_4 + \mathfrak{r}_1\mathfrak{y}_3 + \mathfrak{r}_2\mathfrak{y}_2 + \mathfrak{r}_3\mathfrak{y}_1 + \mathfrak{r}_4\mathfrak{y}_0)\,. \end{aligned}$$

$Q51_{s\_eqns}$ states that the logical variable $\mathfrak{s}_i$ is equal to the qhasm expression $\mathtt{s}_i\mathtt{h}.\mathtt{s}_i\mathtt{l}$ for every $i \in [0, 4]$. $Q51_{mult\_red}$ specifies that $\mathfrak{s}_i$ are computed correctly for every $i \in [0, 4]$. Nonetheless, we find the condition $Q51_{mult\_red}$ is too weak to prove the correctness of the multiply-and-reduce phase. If $\mathtt{s}_i\mathtt{h}.\mathtt{s}_i\mathtt{l}$ indeed had 128 significant bits, overflow could occur during bitwise operations in multiply-and-reduce. To verify multiplication, we estimate tighter upper bounds for $\mathfrak{s}_i$'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 $\mathfrak{s}_i$'s. Define

$$\begin{aligned} Q51_{s\_bds} := \; & 0 \le \mathfrak{s}_0 \le 2^{102} + 4 \cdot 19 \cdot 2^{102} \wedge \\ & 0 \le \mathfrak{s}_1 \le 2 \cdot 2^{102} + 3 \cdot 19 \cdot 2^{102} \wedge \\ & 0 \le \mathfrak{s}_2 \le 3 \cdot 2^{102} + 2 \cdot 19 \cdot 2^{102} \wedge \\ & 0 \le \mathfrak{s}_3 \le 4 \cdot 2^{102} + 19 \cdot 2^{102} \wedge \\ & 0 \le \mathfrak{s}_4 \le 5 \cdot 2^{102} \end{aligned}$$

Consider the Hoare triple $(\!|Q51_{xy\_eqns} \wedge Q51_{xy\_bds}|\!)\, P51_{MR}\, (\!|Q51_{s\_eqns} \wedge Q51_{s\_bds} \wedge Q51_{mult\_red}|\!)$, in addition to checking whether qhasm variables $\mathtt{s}_i\mathtt{h}$'s and $\mathtt{s}_i\mathtt{l}$'s are computed correctly, the qhasm fragment $P51_{MR}$ for multiply-reduce is required to meet the upper bounds in $Q51_{s\_bds}$. The midcondition $Q51_{s\_bds} \wedge Q51_{mult\_red}$ 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 $\mathtt{s}_i\mathtt{h}.\mathtt{s}_i\mathtt{l}$'s to obtain the product of the inputs X and Y. The product must also be in the radix-2^{51} representation. Define

$$Q51_{delayed\_carry} := \sum_{i=0}^{4} \mathfrak{s}_i 2^{51i} \overset{512}{\equiv} \sum_{i=0}^{4} \mathfrak{r}_i 2^{51i} \pmod p.$$

The Hoare triple $(\!|Q51_{s\_eqns} \wedge Q51_{s\_bds} \wedge Q51_{mult\_red}|\!)\, P51_D\, (\!|Q51_{delayed\_carry} \wedge Q51_{r\_bds}|\!)$ verifies that the qhasm fragment $P51_D$ computes a number $\sum_{i=0}^{4} \mathfrak{r}_i 2^{51i}$ in the radix-2^{51} representation, and it is congruent to $\sum_{i=0}^{4} \mathfrak{s}_i 2^{51i}$ modulo p.

Proposition 2. *Assume that*

1. $\models \left(\begin{array}{l} Q51_{xy\_eqns} \wedge \\ Q51_{xy\_bds} \end{array} \right) P51_{MR} \left(\begin{array}{l} Q51_{s\_eqns} \wedge \\ Q51_{s\_bds} \wedge \\ Q51_{mult\_red} \end{array} \right)$; *and*
2. $\models \left(\begin{array}{l} Q51_{s\_eqns} \wedge \\ Q51_{s\_bds} \wedge \\ Q51_{mult\_red} \end{array} \right) P51_D \left(\begin{array}{l} Q51_{delayed\_carry} \wedge \\ Q51_{r\_bds} \end{array} \right)$.

Then $\models \left(\begin{array}{l} Q51_{xy\_eqns} \wedge \\ Q51_{xy\_bds} \end{array} \right) P51_{MR}; P51_D \left(\begin{array}{l} Q51_{prod} \wedge \\ Q51_{r\_bds} \end{array} \right)$.

The Hoare triples in Proposition 2 do not establish $Q51_{prod}$ directly. Again, we formally show $\vdash [Q51_{xy\_bds} \wedge Q51_{mult\_red} \wedge Q51_{delayed\_carry}] \implies Q51_{prod}$ in the proof assistant Coq. Our Coq proof contains less than 600 lines.

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^{51} and radix-2^{64} 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^{64} and radix-2^{51}. The pre- and postconditions of each operators are obtained from the verification of the

Table 1: Verification of the `qhasm` code.

File Name	Description		# of *limb*	# of *MC*	Time
radix-2^{64} representation					
fe25519r64_mul-1	$r = x * y \pmod{2^{255}-19}$, a buggy version		4	1	0m8.73s
fe25519r64_add	$r = x + y \pmod{2^{255}-19}$	Operations of Algorithm 2	4	0	0m3.15s
fe25519r64_sub	$r = x - y \pmod{2^{255}-19}$		4	0	0m16.24s
fe25519r64_mul-2	$r = x * y \pmod{2^{255}-19}$, a fixed version of fe25519r64_mul-1		4	19	73m55.16s
fe25519r64_mul121666	$r = x * 121666 \pmod{2^{255}-19}$		4	2	0m2.03s
fe25519r64_sq	$r = x * x \pmod{2^{255}-19}$		4	15	3m16.67s
ladderstepr64	The implementation of Algorithm 2		4	14	0m3.23s
fe19119_mul	$r = x * y \pmod{2^{191}-19}$		3	12	8m43.07s
mul1271	$r = x * y \pmod{2^{127}-1}$		2	1	141m22.06s
radix-2^{51} representation					
fe25519_add	$r = x + y \pmod{2^{255}-19}$	Operations of Algorithm 2	5	0	0m16.35s
fe25519_sub	$r = x - y \pmod{2^{255}-19}$		5	0	3m38.62s
fe25519_mul	$r = x * y \pmod{2^{255}-19}$		5	27	5658m2.15s
fe25519_mul121666	$r = x * 121666 \pmod{2^{255}-19}$		5	5	0m12.75s
fe25519_sq	$r = x * x \pmod{2^{255}-19}$		5	17	463m59.5s
ladderstep	The implementation of Algorithm 2		5	14	1m29.05s
mul25519	$r = x * y \pmod{2^{255}-19}$, a 3-phase implementation		5	3	286m52.75s
mul25519-p2-1	The delayed carry phase of $r = x * y \pmod{2^{255}-19}$		5	1	2723m16.56s
mul25519-p2-2	The delayed carry phase of $r = x * y \pmod{2^{255}-19}$ with two sub-phases		5	2	263m35.46s
muladd25519	$r = x * y + z \pmod{2^{255}-19}$		5	7	1569m11.06s
re15319	$r = x * y \pmod{2^{153}-19}$		3	3	2409m16.89s

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. The result 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^{64} are equalities. Since Boolector can not verify modular congruence relations in the radix-2^{64} representation, we have to establish them in Coq. On the other hand, Boolector successfully verifies the modular congruence relation $Q51_{delay\_carry}$ for the radix-2^{51} representation. Our Coq proof for the radix-2^{51} representation is thus simplified. The reason why some congruence relations is verified in the radix-2^{51} representation is because we are able to divide $P51_D$ 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 $(\!|Q_0|\!)\, P\, (\!|Q_1 \wedge Q_2|\!)$ to $(\!|Q_0|\!)\, P\, (\!|Q_1|\!)$ and $(\!|Q_0|\!)\, P\, (\!|Q_2|\!)$. 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 $a \overset{256}{=} b * c$ 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, 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 $r = 19(x_0y_2 + x_1y_1 + x_2y_0)$ or $r = 19x_0y_2 + (19x_1y_1 + 19x_2y_0)$. If we really 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 $\left(\!\left|Q_0^1 \wedge Q_0^2 \wedge \cdots \wedge Q_0^n\right|\!\right) P\, (\!|Q_1|\!)$, our translator automatically removes logical variables that do not appear in Q_1; it removes Q_0^i if some variable in Q_0^i neither appears in Q_1 nor gets updated in P. For example, given a Hoare triple $\left(\!\left|\mathtt{r} = \mathfrak{r}_1 \wedge \mathfrak{r}_1 = \mathfrak{r}_0 + \mathtt{x} \wedge \mathtt{x} \leq 2^{51}\right|\!\right)$ `r+=y` $(\!|\mathtt{r} = \mathfrak{r}_1 + \mathtt{y}|\!)$, this heuristic produces $(\!|\mathtt{r} = \mathfrak{r}_1|\!)$ `r+=y` $(\!|\mathtt{r} = \mathfrak{r}_1 + \mathtt{y}|\!)$ where (1) $\mathfrak{r}_1 = \mathfrak{r}_0 + \mathtt{x}$ is removed because the logical variable $\mathfrak{r}_0$ does not appear in the postcondition, and (2) $\mathtt{x} \leq 2^{51}$ is removed

because x neither gets updated nor appears in the postcondition. Traditional program slicing may not remove r_0 and x because both variables are related to r_1, which appears in the postcondition. We cannot verify fe25519r64_sq and 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 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 insertion of midconditions. 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 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.

8. REFERENCES

[1] R. Affeldt. On construction of a library of formally verified low-level arithmetic functions. *Innovations in Systems and Software Engineering*, 9(2):59–77, 2013. https://staff.aist.go.jp/reynald.affeldt/documents/arilib-affeldt.pdf.

[2] R. Affeldt and N. Marti. An approach to formal verification of arithmetic functions in assembly. In M. Okada and I. Satoh, editors, *Advances in Computer Science – ASIAN 2006*, volume 4435 of *Lecture Notes in Computer Science*, pages 346–360. Springer-Verlag Berlin Heidelberg, 2007. https://staff.aist.go.jp/reynald.affeldt/documents/affeldt-asian2006.pdf.

[3] R. Affeldt, D. Nowak, and K. Yamada. Certifying assembly with formal security proofs: The case of BBS. *Science of Computer Programming*, 77(10–11):1058–1074, 2012. http://www.sciencedirect.com/science/article/pii/S0167642311001493.

[4] J. B. Almeida, M. Barbosa, G. Barthe, and F. Dupressoir. Certified computer-aided cryptography: efficient provably secure machine code from high-level implementations. In V. Gligor and M. Yung, editors, *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1217–1230. ACM New York, 2013. http://eprint.iacr.org/2013/316/.

[5] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *ACM Symposium on Principles of Programming Languages – (POPL 1988)*, pages 1–11. ACM Press, 1988. http://courses.cs.washington.edu/courses/cse501/04wi/papers/alpern-popl88.pdf.

[6] R. Avanzi, H. Cohen, C. Doche, G. Frey, T. Lange, K. Nguyen, and F. Vercauteren. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. Chapman & Hall/CRC, 2006.

[7] E.-I. Bartzia. Formalisation des courbes elliptiques en COQ. Master's thesis, Université de Vincennes-Saint Denis – Paris VIII, 2011. http://pierre-yves.strub.nu/research/ec/.

[8] A. G. Bayrak, F. Regazzoni, D. Novo, and P. Ienne. Sleuth: Automated verification of software power analysis countermeasures. In G. Bertoni and J.-S. Coron, editors, *Cryptographic Hardware and Embedded Systems – CHES 2013*, volume 8086 of *Lecture Notes in Computer Science*, pages 293–310. Springer-Verlag Berlin Heidelberg, 2013.

[9] D. J. Bernstein. qhasm: tools to help write high-speed software. http://cr.yp.to/qhasm.html.

[10] D. J. Bernstein. Supercop: System for unified performance evaluation related to cryptographic operations and primitives. http://bench.cr.yp.to/supercop.html. Published as part of ECRYPT II VAMPIRE Lab.

[11] D. J. Bernstein. Curve25519: new Diffie-Hellman speed records. In M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, editors, *Public Key Cryptography – PKC 2006*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer-Verlag Berlin Heidelberg, 2006. http://cr.yp.to/papers.html#curve25519.

[12] D. J. Bernstein. Batch binary Edwards. In S. Halevi, editor, *Advances in Cryptology – CRYPTO 2009*, volume 5677 of *Lecture Notes in Computer Science*, pages 317–336. Springer-Verlag Berlin Heidelberg, 2009. http://cr.yp.to/papers.html#bbe.

[13] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang. High-speed high-security signatures. In B. Preneel and T. Takagi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2011*, volume 6917 of *Lecture Notes in Computer Science*, pages 124–142. Springer-Verlag Berlin Heidelberg, 2011. see also full version [14].

[14] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, 2012. http://cryptojedi.org/papers/#ed25519, see also short version [13].

[15] D. J. Bernstein and T. L. (editors). eBACS: ECRYPT Benchmarking of Cryptographic Systems. http://bench.cr.yp.to (accessed May 17, 2014).

[16] D. J. Bernstein and T. L. (editors). Explicit-formulas database. http://www.hyperelliptic.org/EFD/ (accessed May 17, 2014).

[17] D. J. Bernstein, W. Janssen, T. Lange, and P. Schwabe. TweetNaCl: A crypto library in 100 tweets, 2013. http://cryptojedi.org/papers/#tweetnacl.

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

A. Hevia and G. Neven, editors, *Progress in Cryptology – LATINCRYPT 2012*, volume 7533 of *Lecture Notes in Computer Science*, pages 159–176. Springer-Verlag Berlin Heidelberg, 2012. http://cryptojedi.org/papers/#coolnacl.

[19] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions*. EATCS. Springer, 2004.

[20] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and S. Zanella-Béguelin. miTLS: A verified reference TLS implementation, 2014. http://www.mitls.org/wsgi/home.

[21] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P.-Y. Strub. Implementing TLS with verified cryptographic security. In *IEEE Symposium on Security and Privacy 2013*, pages 445–459, 2013. full version: http://www.mitls.org/downloads/miTLS-report.pdf.

[22] B. B. Brumley, M. Barbosa, D. Page, and F. Vercauteren. Practical realisation and elimination of an ECC-related software bug attack. In O. Dunkelman, editor, *Topics in Cryptology – CT-RSA 2012*, volume 7178 of *Lecture Notes in Computer Science*, pages 171–186. Springer-Verlag Berlin Heidelberg, 2012. http://eprint.iacr.org/2011/633.

[23] R. Brummayer and A. Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In S. Kowalewski and A. Philippou, editors, *Tools and Algorithms for the Construction and Analysis of Systems – (TACAS 2009)*, volume 5505 of *Lecture Notes in Computer Science*, pages 174–177. Springer-Verlag Berlin Heidelberg, 2009. http://fmv.jku.at/papers/BrummayerBiere-TACAS09.pdf.

[24] C. Costello, H. Hisil, and B. Smith. Faster compact diffie–hellman: Endomorphisms on the x-line. In P. Q. Nguyen and E. Oswald, editors, *Advances in Cryptology – EUROCRYPT 2014*, volume 8441 of *Lecture Notes in Computer Science*, pages 183–200. Springer-Verlag Berlin Heidelberg, 2014. http://eprint.iacr.org/2013/692/.

[25] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1–3):35–45, 2007. http://pgbovine.net/publications/daikon-invariant-detector_SCP-2007.pdf.

[26] M. Gordon. CryptVer project. http://www.cl.cam.ac.uk/~mjcg/proposals/CryptVer/.

[27] D. Hankerson, A. Menezes, and S. A. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag New York, 2004.

[28] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. http://www.spatial.maine.edu/~worboys/processes/hoare/20axiomatic.pdf.

[29] P. Longa and F. Sica. Four-dimensional Gallant-Lambert-Vanstone scalar multiplication. In X. Wang and K. Sako, editors, *Advances in Cryptology – ASIACRYPT 2012*, volume 7658 of *Lecture Notes in Computer Science*, pages 718–739. Springer-Verlag Berlin Heidelberg, 2012. https://eprint.iacr.org/2011/608.

[30] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In D. H. Won and S. Kim, editors, *Information Security and Cryptology – ICISC 2005*, volume 3935 of *Lecture Notes in Computer Science*, pages 156–168. Springer-Verlag Berlin Heidelberg, 2005. Full version at http://eprint.iacr.org/2005/368/.

[31] P. L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, 1987. http://www.ams.org/journals/mcom/1987-48-177/S0025-5718-1987-0866113-7/S0025-5718-1987-0866113-7.pdf.

[32] M. O. Myreen and M. J. C. Gordon. Hoare logic for realistically modelled machine code. In O. Grumberg and M. Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424 of *Lecture Notes in Computer Science*, pages 568–582. Springer-Verlag Berlin Heidelberg, 2007. http://www.cl.cam.ac.uk/~mom22/mc-hoare-logic.pdf.

[33] T. Oliveira, J. López, D. F. Aranha, and F. Rodríguez-Henríquez. Lambda coordinates for binary elliptic curves. In G. Bertoni and J.-S. Coron, editors, *Cryptographic Hardware and Embedded Systems – CHES 2013*, volume 8086 of *Lecture Notes in Computer Science*, pages 311–330. Springer-Verlag Berlin Heidelberg, 2013. http://eprint.iacr.org/2013/131/20130611:205154.

[34] S. Ragan. Bugcrowd launches funding drive to audit popenssl. News article on CSO, 2014. http://www.csoonline.com/article/2145020/security-industry/bugcrowd-launches-funding-drive-to-audit-openssl.html (accessed May 17, 2014).

[35] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *ACM Symposium on Principles of Programming Languages – (POPL 1988)*, pages 12–27. ACM Press, 1988. http://www.cs.wustl.edu/~cytron/cs531/Resources/Papers/valnum.pdf.

[36] L. Théry and G. Hanrot. Primality proving with elliptic curves. In K. Schneider and J. Brandt, editors, *Theorem Proving in Higher Order Logics*, volume 4732 of *Lecture Notes in Computer Science*, pages 319–333. Springer-Verlag Berlin Heidelberg, 2007. http://hal.inria.fr/docs/00/14/06/58/PDF/paper.pdf.

[37] K. White and M. Green. IsTrueCryptAuditedYet? http://istruecryptauditedyet.com/ (accessed May 17, 2014).