Verifying Curve25519 Software

Yu-Fang Chen[1], Chang-Hong Hsu[2], Hsin-Hung Lin[3], Peter Schwabe[4], Ming-Hsien Tsai[1], Bow-Yaw Wang[1], Bo-Yin Yang[1], and Shang-Yi Yang[1] ⋆

[1] Institute of Information Science
Academia Sinica
128 Section 2 Academia Road, Taipei 115-29, Taiwan
yfc@iis.sinica.edu.tw, mhtsai208@gmail.com, bywang@iis.sinica.edu.tw, by@crypto.tw, ilway25@crypto.tw
[2] University of Michigan, Ann Arbor, USA
hsuch@umich.edu
[3] Faculty of Information Science and Electrical Engineering
Kyushu University
744, Motoka, Nishi-ku, Fukuoka-Shi, Fukuoka, 819-0395, Japan
h-lin@ait.kyushu-u.ac.jp
[4] Radboud University Nijmegen
Digital Security Group
PO Box 9010, 6500GL Nijmegen, The Netherlands
peter@cryptojedi.org

Abstract. This paper presents results on formal verification of high-speed cryptographic software. We consider speed-record-setting hand-optimized assembly software for Curve25519 elliptic-curve key exchange in [5]. Two versions for different microarchitectures are available. We successfully verify the core part of computation, and reproduce a bug in a previously published edition. An SMT solver for the bit-vector theory is used to establish almost all properties. Remaining properties are verified in a proof assistant. We also exploit the compositionality of Hoare logic to address the scalability issue. Essential differences between both versions of the software are discussed from a formal-verification perspective.

1 Introduction

XXX: Emphasize importance of correctness of crypto software

There are basically three different approaches to ensure the correctness of cryptographic 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 very nature of many cryptographic algorithms. For example, flipping one bit of either the key or the message block of a block cipher should change each output bit with probability 1/2. An AES implementation which produces correct test vectors on multiple messages of different lengths is very unlikely to produce incorrect test vectors on some other messages. Consequently, the confidence is the correctness of well-tested AES software is very high. However, there are classes of bugs that are very hard to catch with testing.

⋆ This work was supported by the National Science Foundation under grant 1018836 and by the Netherlands Organisation for Scientific Research (NWO) through Veni 2013 project 13114. Permanent ID of this document: 55ab8668ce87d857c02a5b2d56d7da38. Date: 2014.04.28.; it was supported by the Academia Sinica Institute of Information Science under BY's Specialty Project and Career Advancement Award.

Auditing: Code audits by independent experts XXX.

One disadvantage of auditing is monetary cost. For example, the cost for the recent code audit of Truecrypt [**?**] was US$XXX; the cost for the recent audit of cryptocat [**?**] was US$XXX. XXX: comment on OpenSSL's funding.

Another disadvantage of software auditing is a conflict with performance. Cryptographic software which is relatively easy (and therefore cheap) to audit is concise, has small amounts of code, and is naturally portable. High-speed cryptographic software is written in assembly with optimziations for multiple architectures and micro-architectures. The core development team of the Networking and Cryptography library (NaCl) [**?**], together with Janssen, recently released TweetNaCl, a reimplementation of NaCl which is optimized for conciseness [7]. The authors state that TweetNaCl "allows correct functionality to be verified by human auditors with reasonable effort"; however, the Curve25519 elliptic-curve Diffie-Hellman software in TweetNaCl takes 2.5 million cycles on an Intel Ivy Bridge processor – more than 10 times more than the hand-optimized assembly implementation presented in [5].

Finally, also a code audit is not guaranteed to find all bugs in cryptographic software.

Verification: The third direction to ensure correctness of (cryptographic) software is formal verification.

XXX: Something on how these three techniques work together.

Contributions of this paper. In this paper we

Related work. XXX: Verifying compilers vs. verifying code XXX: Verifying correctness vs. verifying side-channel protection

2 Curve25519

Cryptography based on the arithmetic on elliptic curves was proposed independently by Miller and Koblitz [12,13]. We only give a very brief introduction to elliptic-curve cryptography here; please refer to (for example) [1,11] for more information.

Let $\mathbb{F}_q$ be the finite field with q elements. For $a_1, a_2, a_3, a_4, a_6 \in \mathbb{F}_q$, 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 $\mathbb{F}_q$ (with certain conditions, cf. [11, Chapter 3]). The set of points $(x, y) \in \mathbb{F}_q \times \mathbb{F}_q$ together with a "point at infinity" form a group of size $\ell \approx q$, usually written additively. The group law is efficiently computable through a few operations in the field $\mathbb{F}_q$, so given a point P and a scalar $k \in \mathbb{Z}$ it is easy to do a scalar multiplication $k \cdot P$, which requires a number of addition operations that is linear in the length of k (i.e., logarithmic in k).

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

Curve25519 is an elliptic-curve Diffie-Hellman key exchange protocol proposed by Bernstein in 2006 [4]. It is based on arithmetic on the elliptic curve $E : y^2 = x^3 + 486662x^2 + x$ defined over the field $\mathbb{F}_{2^{255}-19}$.

2.1 The Montgomery ladder

Scalar multiplication in Curve25519 uses a so-called differential-addition chain proposed by Montgomery in [14] that uses only the x-coordinate of points. It is highly regular, performs one *ladder step* per scalar bit, and can be run in constant time; the whole loop is often called *Montgomery ladder*. An overview of the structure of the Montgomery ladder and the operations involved in one ladder-step are given respectively in Algs. 1 and 2. The inputs and outputs x_P, X_1, X_2, Z_2, X_3, Z_3, and temporary values T_i are elements in $\mathbb{F}_{2^{255}-19}$. The performance of the computation is is largely determined by the performance of arithmetic operations in this field.

Algorithm 1 Curve25519 Montgomery Ladder

Input: A 255-bit scalar k and the 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 One "step" of the Curve25519 Montgomery Ladder

```
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 [5,6] is the representation of elements of $\mathbb{F}_{2^{255}-19}$. Both implementations have the core parts, the Montgomery ladder step, in fully inlined, hand-optimized assembly. These core portions are what we target for verification in this paper.

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

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

Section 3 of [5] 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 Field 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 the 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.

Multiplication of 256-bit integers. The approach for multiplication in radix-2^{64} chosen by [5] is a simple schoolbook approach. Multiplication of two 256-bit integers X and Y can be seen as a 4-step computation which in each step involves one limb of Y and all limbs of X as follows:

$$A_0 = Xy_0,\ A_1 = 2^{64}Xy_1 + A_0,\ A_2 = 2^{128}Xy_2 + A_1,\ S = A_3 = 2^{192}Xy_2 + A_2. \quad (1)$$

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 [5, Section 3]. In Sec. 6 we will explain why integrated multiplication and handling of carry bits also constitutes a major obstacle for formal verification.

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

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

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

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

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

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

3.2 Field arithmetic in radix-2^{51} representation

Due to the performance penalties in handling carries, [5] 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 $[0, 2^{51} + 2^{15}]$. Additions are done limbwise, e.g., after the first operation $T_1 \leftarrow X_2 + Z_2$, the limbs of T_1 have at most 53 bits. Subtractions are done by first adding a multiple of p guaranteed to exceed the subtrahend limbwise. For example, the subtraction $T_2 \leftarrow X_2 - Z_2$ is done by first adding `0xFFFFFFFFFFFDA` to the lowest limb of X_2 and `0xFFFFFFFFFFFFE` to the four higher limbs of X_2, and then subtracting corresponding limbs of Z_2. The value added is $2p$, which does not change the result (as element of $\mathbb{F}_p$), yet ensures that all limbs of the result T_2 are positive and have at most 53 bits.

The most complex operation—multiplication—is split in two parts, but these now differ from those of Sec. 3.1. The first step performs multiplication and modular reduction; the second step performs the delayed carries.

Multiply-and-Reduce in radix-2^{51}. To multiply $X = x_0+2^{51}x_1+2^{102}x_2+2^{153}x_3+2^{204}x_4$ and $Y = y_0 + 2^{51}y_1 + 2^{102}y_2 + 2^{153}y_3 + 2^{204}y_4$, start by precomputing $19y_1, 19y_2, 19y_3$ and $19y_4$, then compute 5 intermediate values $s_0, \ldots, s_4$, each in two 64-bit registers, as

$$\begin{aligned}
s_0 &:= x_0y_0 + x_1(19y_4) + x_2(19y_3) + x_3(19y_2) + x_4(19y_1),\\
s_1 &:= x_0y_1 + x_1y_0 + x_2(19y_4) + x_3(19y_3) + x_4(19y_2),\\
s_2 &:= x_0y_2 + x_1y_1 + x_2y_0 + x_3(19y_4) + x_4(19y_3),\\
s_3 &:= x_0y_3 + x_1y_2 + x_2y_1 + x_3y_0 + x_4(19y_4),\\
s_4 &:= x_0y_4 + x_1y_3 + x_2y_2 + x_3y_1 + x_4y_0.
\end{aligned} \tag{2}$$

Note that 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), $0 \le s_0, s_1, s_2, s_3, 19s_4 < 95 \cdot 2^{108} < 2^{115}$. Thus, adding of multiplication results can be achieved by two 64-bit addition instructions (one with carry); carry bits from these additions are collected by the "free" bits of the higher register of s_i.

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

Delayed carry. For a 2-register value s_i, let $s_i^{(l)}$ denote the lower register and $s_i^{(h)}$ the higher register, i.e., $s_i = s_i^{(l)} + 2^{64}s_i^{(h)}$. Carrying from s_i to s_{i+1} is done as follows: Shift $s_i^{(h)}$ to the left by 13 and shift the 13 high bits of $s_i^{(l)}$ into the 13 low bits of the result. This operation is just one `SHLD` instruction. Now set the high 13 bits of $s_i^{(l)}$ to zero (logical `AND` with $2^{51}-1$). Now do the same shift-by-13 operation on $s_{i+1}^{(h)}$ and $s_{i+1}^{(l)}$, set the high 13 bits of $s_{i+1}^{(l)}$ to zero, add $s_i^{(h)}$ to $s_{i+1}^{(l)}$ and discard $s_i^{(h)}$. This carry chain is performed from s_0 through s_4; then $s_4^{(h)}$ is multiplied by 19 (using a *single-word* `MUL`) and added to $s_0^{(l)}$.

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

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

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

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

4 Tools Used to Verify Curve25519

4.1 Portable assembly: `qhasm`

The software we are verifying has not been written directly in AMD64 assembly, but in the portable assembly language `qhasm` developed by Bernstein [2]. 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 explicitely by the programmer.

Verifying `qhasm` code. The Curve25519 software we verified is publicly available as part of the SUPERCOP benchmarking framework [3], 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.

4.2 The Boolector SMT solver

Boolector is an efficient SMT solver supporting the bit vector and array theory [9]. 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 Boolean satisfiability 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 langauges such as `qhasm`, memory and pointers are indispensable. On the other hand, `qhasm` programs are finite. Sizes of program variables (including pointers) must be declared. Subsequently, each variable is of finite domain and, more importantly, the memory is finite. Since formal models of `qhasm` programs are necessarily finite, they are expressible in theories of bit vectors and arrays. Boolector therefore fits perfectly in this application.

4.3 The Coq proof assistant

The Coq proof assistant has been developed in INRIA for more than twenty years [8]. The tool is based on a higher-order logic called the Calculus of Inductive Construction and has lots of libraries for various theories. In contrast to model-theoretic tools such as 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 modulo 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 [5,6]. A ladder step (Alg. 2) consists of 18 field arithmetic operations. Considering the complexity of a $\mathbb{F}_p$ multiplication (Sec. 3), the correctness of manually optimized `qhasm` implementation for the Montgomery ladder step is by no mean clear. 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. Let Q, Q' be predicate logic formulae, and P a `qhasm` program fragment. Informally, a *Hoare triple*

$(\!|Q|\!)P(\!|Q'|\!)$ specifies that the `qhasm` fragment P ends in a state satisfying Q' provided that P starts in a state satisfying Q. The formula Q and Q' are called *pre-* and *post-conditions* respectively. We only use quantifier-free pre- and post-conditions. The `typewriter` and $\mathfrak{Frankfurt}$ fonts are used to denote program and logical variables respectively in pre- and post-conditions.

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

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

Let $P64$ denote the `qhasm` program for $\mathbb{F}_p$ multiplication in the radix-2^{64} representation. The inputs $X = x_0 + 2^{64}x_1 + 2^{128}x_2 + 2^{192}x_3$ and $Y = y_0 + 2^{64}y_1 + 2^{128}y_2 + 2^{192}y_3$ are stored in memory pointed to by the `qhasm` variables `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 + 8`i`)` and `*(uint64 *)(yp + 8`i`)` 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 + 8`i`)` 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. The formula $Q64_{r\_eqns}$ introduces the logical variable $\mathfrak{r}_i$ equal to the coefficient 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 pre-condition $Q64_{xy\_eqns}$ effectively memorize the input limbs before the execution of $P64$. The post-condition $Q64_{r\_eqns} \wedge Q64_{prod}$ 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. Recall the proof rule for compositions in Hoare logic, where Q' is a *mid-condition*:

$$\frac{\vdash (\!|Q|\!)P_0(\!|Q'|\!) \quad \vdash (\!|Q'|\!)P_1(\!|Q''|\!)}{\vdash (\!|Q|\!)P_0; P_1(\!|Q''|\!)} \text{ Composition}$$

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

Multiply in the radix-2^{64} representation Let $P64_M$ and $P64_R$ denote the `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 `*(uint 64 *)(sp + 8`i`)` for every $i \in [0, 3]$. Define

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

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 pre-condition $Q64_{xy\_eqns}$ 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.

Reduce in the 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 formulae $Q64_{s'\_red}$, $Q64_{s''\_red}$, $Q64_{r\_red}$ are defined for the three reduction steps. The formulae $Q64_{s'\_bds}$, $Q64_{s''\_bds}$, and $Q64_{r\_bds}$ moreover give upper bounds.

$$Q64_{s'\_red} := \sum_{i=0}^{4} \mathfrak{s}'_i 2^{64i} \overset{320}{=} \mathfrak{s}_0 + 2^{64}\mathfrak{s}_1 + 2^{128}\mathfrak{s}_2 + 2^{192}\mathfrak{s}_3 + 38(\mathfrak{s}_4 + 2^{64}\mathfrak{s}_5 + 2^{128}\mathfrak{s}_6 + 2^{192}\mathfrak{s}_7)$$

$$Q64_{s'\_bds} := \bigwedge_{i=0}^{4} 0 \le \mathfrak{s}'_i < 2^{64}$$

$$Q64_{s''\_red} := \sum_{i=0}^{4} \mathfrak{s}''_i 2^{64i} \overset{320}{=} 38\mathfrak{s}'_4 + \sum_{i=0}^{3} \mathfrak{s}'_i 2^{64i} \qquad Q64_{s''\_bds} := \bigwedge_{i=0}^{3} 0 \le \mathfrak{s}''_i < 2 \wedge 0 \le \mathfrak{s}''_4 < 2$$

$$Q64_{r\_red} := \sum_{i=0}^{3} \mathfrak{r}_i 2^{64i} \overset{256}{=} 38\mathfrak{s}''_4 + \sum_{i=0}^{3} \mathfrak{s}''_i 2^{64i} \qquad Q64_{r\_bds} := \bigwedge_{i=0}^{3} 0 \le \mathfrak{r}_i < 2^{64}$$

Consider the following Hoare triple

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

The pre-condition $Q64_{mult}$ 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 post-condition 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. $\models (\!|Q64_{mult}|\!)P64_R(\!|\begin{array}{c} Q64_{s'\_red} \wedge Q64_{s'\_bds} \wedge Q64_{s''\_red} \wedge Q64_{s''\_bds} \wedge \\ Q64_{r\_red} \wedge Q64_{r\_bds} \wedge Q64_{r\_eqns} \end{array}|\!)$.

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 the radix-2^{51} representation

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

The formulae $Q51_{xy\_eqns}$, $Q51_{r\_eqns}$, $Q51_{prod}$ are defined similarly as in the radix-2^{64} representation. Observe that each limb in the radix-2^{51} representation has only 51 significant bits.

The formulae $Q51_{xy\_bds}$ and $Q51_{r\_bds}$ specify that the inputs and outcome are in the radix-2^{51} representation.

$$Q51_{xy\_bds} := \bigwedge_{i=0}^{4} 0 \le \mathfrak{x}_i < 2^{51} \wedge \bigwedge_{i=0}^{4} 0 \le \mathfrak{y}_i < 2^{51} \qquad Q51_{r\_bds} := \bigwedge_{i=0}^{4} 0 \le \mathfrak{r}_i < 2^{51}$$

In the top-level Hoare triple $(\!|Q51_{xy\_eqns} \wedge Q51_{xy\_bds}|\!)P51(\!|Q51_{r\_eqns} \wedge Q51_{r\_bds} \wedge Q51_{prod}|\!)$, the pre-condition $Q51_{xy\_eqns} \wedge Q51_{xy\_bds}$ assumes that the inputs X and Y are in the radix-2^{51} representation. The post-condition $Q51_{r\_eqns} \wedge Q51_{r\_bds} \wedge Q51_{prod}$ 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

$$\begin{aligned}
Q51_{s_eqns} &:= \bigwedge_{i=0}^{4} \mathfrak{s}_i \overset{128}{=} \mathtt{s}_i\mathtt{h}.\mathtt{s}_i\mathtt{l} \\
Q51_{mult_red} &:= \mathfrak{s}_0 \overset{128}{=} (\mathfrak{x}_0\mathfrak{y}_0 + 19(\mathfrak{x}_1\mathfrak{y}_4 + \mathfrak{x}_2\mathfrak{y}_3 + \mathfrak{x}_3\mathfrak{y}_2 + \mathfrak{x}_4\mathfrak{y}_1)) \wedge \\
&\quad \mathfrak{s}_1 \overset{128}{=} (\mathfrak{x}_0\mathfrak{y}_1 + \mathfrak{x}_1\mathfrak{y}_0 + 19(\mathfrak{x}_2\mathfrak{y}_4 + \mathfrak{x}_3\mathfrak{y}_3 + \mathfrak{x}_4\mathfrak{y}_2)) \wedge \\
&\quad \mathfrak{s}_2 \overset{128}{=} (\mathfrak{x}_0\mathfrak{y}_2 + \mathfrak{x}_1\mathfrak{y}_1 + \mathfrak{x}_2\mathfrak{y}_0 + 19(\mathfrak{x}_3\mathfrak{y}_4 + \mathfrak{x}_4\mathfrak{y}_3)) \wedge \\
&\quad \mathfrak{s}_3 \overset{128}{=} (\mathfrak{x}_0\mathfrak{y}_3 + \mathfrak{x}_1\mathfrak{y}_2 + \mathfrak{x}_2\mathfrak{y}_1 + \mathfrak{x}_3\mathfrak{y}_0 + 19\mathfrak{x}_4\mathfrak{y}_4) \wedge \\
&\quad \mathfrak{s}_4 \overset{128}{=} (\mathfrak{x}_0\mathfrak{y}_4 + \mathfrak{x}_1\mathfrak{y}_3 + \mathfrak{x}_2\mathfrak{y}_2 + \mathfrak{x}_3\mathfrak{y}_1 + \mathfrak{x}_4\mathfrak{y}_0)\,.
\end{aligned}$$

$Q51_{s\_eqns}$ 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 mid-condition $Q51_{s\_bds} \wedge Q51_{mult\_red}$ 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*

1. $\models (\!|Q51_{xy\_eqns} \wedge Q51_{xy\_bds}|\!)P51_{MR}(\!|Q51_{s\_eqns} \wedge Q51_{s\_bds} \wedge Q51_{mult\_red}|\!)$;
2. $\models (\!|Q51_{s\_eqns} \wedge Q51_{s\_bds} \wedge Q51_{mult\_red}|\!)P51_D(\!|Q51_{delayed\_carry} \wedge Q51_{r\_bds}|\!)$.

Then $\models (\!|Q51_{xy\_eqns} \wedge Q51_{xy\_bds}|\!)P51_{MR}; P51_D(\!|Q51_{prod} \wedge Q51_{r\_bds}|\!)$.

The Hoare triples in the proposition 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.

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

6 Results and Discussion

In this section, we present results and findings during the verification process. A summary of the experimental results is in Table 1. The columns are the number of limbs, the number of mid-conditions used, and the verification time used in 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.

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 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. The rows mul25519-p2-1 and mul25519-p2-2 are the results of verifying the delayed carry phase of mul25519, a 3-phase implementation of a multiplication. The result shows that if we add an additional mid-condition to the delayed carry phase of mul25519, the verification time of the delayed carry phase can be reduced from 2723 minutes to 263 minutes.

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	0m8.733s
fe25519r64_add	$r = x + y \pmod{2^{255} - 19}$	Operations of Algorithm 2	4	0	0m3.154s
fe25519r64_sub	$r = x - y \pmod{2^{255} - 19}$		4	0	0m16.236s
fe25519r64_mul-2	$r = x * y \pmod{2^{255} - 19}$, a fixed version of fe25519r64_mul-1		4	19	73m55.159s
fe25519r64_mul121666	$r = x * 121666 \pmod{2^{255} - 19}$		4	2	0m2.034s
fe25519r64_sq	$r = x * x \pmod{2^{255} - 19}$		4	15	3m16.669s
ladderstepr64	The implementation of Algorithm 2		4	14	0m3.227s
fe19119_mul	$r = x * y \pmod{2^{191} - 19}$		3	12	8m43.071s
mul1271	$r = x * y \pmod{2^{127} - 1}$		2	1	141m22.062s
radix-2^{51} representation					
fe25519_add	$r = x + y \pmod{2^{255} - 19}$	Operations of Algorithm 2	5	0	0m16.349s
fe25519_sub	$r = x - y \pmod{2^{255} - 19}$		5	0	3m38.621s
fe25519_mul	$r = x * y \pmod{2^{255} - 19}$		5	27	5658m2.148s
fe25519_mul121666	$r = x * 121666 \pmod{2^{255} - 19}$		5	5	0m12.753s
fe25519_sq	$r = x * x \pmod{2^{255} - 19}$		5	17	463m59.503s
ladderstep	The implementation of Algorithm 2		5	14	1m29.051s
mul25519	$r = x * y \pmod{2^{255} - 19}$, a 3-phase implementation		5	3	286m52.751s
mul25519-p2-1	The delayed carry phase of $r = x * y \pmod{2^{255} - 19}$		5	1	2723m16.556s
mul25519-p2-2	The delayed carry phase of $r = x * y \pmod{2^{255} - 19}$ with two subphases		5	2	263m35.461s
muladd25519	$r = x * y + z \pmod{2^{255} - 19}$		5	7	1569m11.057s
re15319	$r = x * y \pmod{2^{153} - 19}$		3	3	2409m16.890s

Note that all post-conditions 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 conjuncted 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, we over-approximate Boolector specifications automatically by reducing logical variables and weakening preconditions. The validity of an over-approximated specification guaranteed the validity of the original one, but not vice versa. To be more specific, given a specification $(\!|Q_0^1 \wedge Q_0^2 \wedge \cdots \wedge Q_0^n|\!)P(\!|Q_1|\!)$, 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. 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 mid-conditions for verifications. Although the verification steps between preconditions and postconditions are done automatically, it would be even better if we can increase the degree of automation further by investigating techniques

for automatic insertion of mid-conditions. We think the tools for automatic assertion insertion could be relevant [10]. The tool obtain assertions based on given templates of assertions and by synthesizing them dynamically from observed executions traces.

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

References

1. Roberto Avanzi, Henri Cohen, Christophe Doche, Gerhard Frey, Tanja Lange, Kim Nguyen, and Frederik Vercauteren. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. Chapman & Hall/CRC, 2006. 2
2. Daniel J. Bernstein. `qhasm`. `http://cr.yp.to/qhasm.html`. 6
3. Daniel 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. 7
4. Daniel J. Bernstein. Curve25519: new Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal 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`. 2
5. Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. In Bart Preneel and Tsuyoshi 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 [6]. 1, 2, 4, 5, 7, 14
6. Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, 2012. `http://cryptojedi.org/papers/#ed25519`, see also short version [5]. 4, 7, 14
7. Daniel J. Bernstein, Wesley Janssen, Tanja Lange, and Peter Schwabe. TweetNaCl: A crypto library in 100 tweets, 2013. `http://cryptojedi.org/papers/#tweetnacl`. 2
8. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions*. EATCS. Springer, 2004. 7
9. 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, 2009. 7
10. Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1–3):35–45, 2007. 14
11. Darrel Hankerson, Alfred Menezes, and Scott A. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag New York, 2004. 2
12. Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, 1987. `http://www.ams.org/journals/mcom/1987-48-177/S0025-5718-1987-0866109-5/S0025-5718-1987-0866109-5.pdf`. 2
13. Victor S. Miller. Use of elliptic curves in cryptography. In Hugh C. Williams, editor, *Advances in Cryptology – CRYPTO '85: Proceedings*, volume 218 of *Lecture Notes in Computer Science*, pages 417–426. Springer-Verlag Berlin Heidelberg, 1986. 2
14. Peter 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`. 3