The Complete Cost of Cofactor $h = 1$

Peter Schwabe(✉) and Amber Sprenkels(✉)

Digital Security Group, Radboud University,
P.O. Box 9010, 6500 Nijmegen, GL, The Netherlands
peter@cryptojedi.org, amber@electricdusk.com

Abstract. This paper presents optimized software for constant-time variable-base scalar multiplication on prime-order Weierstraß curves using the complete addition and doubling formulas presented by Renes, Costello, and Batina in 2016. Our software targets three different microarchitectures: Intel Sandy Bridge, Intel Haswell, and ARM Cortex-M4. We use a 255-bit elliptic curve over $\mathbb{F}_{2^{255}-19}$ that was proposed by Barreto in 2017. The reason for choosing this curve in our software is that it allows most meaningful comparison of our results with optimized software for Curve25519. The goal of this comparison is to get an understanding of the cost of using cofactor-one curves with complete formulas when compared to widely used Montgomery (or twisted Edwards) curves that inherently have a non-trivial cofactor.

Keywords: Elliptic curve cryptography · SIMD · Curve25519 · Scalar multiplication · Prime-field arithmetic · Cofactor security

1 Introduction

Since its invention in 1985, independently by Koblitz [34] and Miller [38], elliptic-curve cryptography (ECC) has widely been accepted as the state of the art in asymmetric cryptography. This success story is mainly due to the fact that attacks have not substantially improved: with a choice of elliptic curve that was in the 80s already considered conservative, the best attack for solving the elliptic-curve discrete-logarithm problem is still the generic Pollard-rho algorithm (with minor speedups, for example by exploiting the efficiently computable negation map in elliptic-curve groups [14]).

One of the main developments since the first generation of elliptic-curve cryptography has been in the choice of curves. Until 2006, the widely accepted choice of elliptic curve was a prime-order Weierstraß curve. In 2006, Bernstein proposed the Montgomery curve "Curve25519"[1] as an alternative offering multiple security and performance features. Most importantly—at least in the context of this paper—it featured highly efficient *complete formulas*, which enable fast arithmetic without any checks for special cases. These complete formulas are

[1] In the 2006 paper, Curve25519 referred to the ECDH key-exchange protocol; Bernstein later recommended to use X25519 for the protocol and Curve25519 for the underlying curve [7].

© Springer Nature Switzerland AG 2019
F. Hao et al. (Eds.): INDOCRYPT 2019, LNCS 11898, pp. 375–397, 2019.
https://doi.org/10.1007/978-3-030-35423-7_19

somewhat limited, because they only cover differential addition. This is not a problem for the X25519 elliptic-curve Diffie-Hellman key exchange presented in [6], but it makes implementation of more advanced protocols (like signature schemes) somewhat inconvenient.

This issue was addressed through a sequence of three papers. In [23], Edwards introduced a new description of elliptic curves; in [13], Bernstein and Lange presented very efficient complete formulas for group arithmetic on these curves (and introduced the name "Edwards curves"); and in [8], Bernstein, Birkner, Joye, Lange, and Peters generalized the concept to twisted Edwards curves and showed that this class of elliptic curves is birationally equivalent to Montgomery curves. The twisted Edwards form of Curve25519 was subsequently used in [9,10] for the construction of the Ed25519 digital-signature scheme.

The simplicity and efficiency of X25519 key exchange and Ed25519 signatures resulted in quick adoption in a variety of applications, such as SSH, the Signal protocol, and the Tor anonymity project. Both schemes are also used in TLS 1.3.

Complete Addition or Prime Order. Unfortunately, the advantages of Montgomery and twisted Edwards curves—most notably the very efficient complete addition formulas— have to be weighed against a disadvantage: the group of points cannot have prime order as it always has a cofactor of a multiple of 4. Consequently, a somewhat simplified view on choosing curves for cryptographic applications is that we have to choose between either efficient complete formulas through Montgomery or (twisted) Edwards curves, or prime-order groups through Weierstraß curves.

The design of X25519 and Ed25519 carefully takes the non-trivial cofactor into account. However in more involved protocols, a non-trivial cofactor may complicate the protocol's design, potentially leading to security issues. In the last years, we saw at least two examples of protocols deployed in real-world applications that had catastrophic vulnerabilities because they did not carefully handle the cofactor.

The first example was a vulnerability in the Monero cryptocurrency, that allowed for arbitrary double spending [36]. Monero requires that "key images"—which bind transactions to their sender's public key—should be non-malleable, i.e. for a transaction to be valid, its public key must be unique. Unfortunately, due to the cofactor, an attacker could construct different key images that were bound to the same public key, therefore allowing arbitrary double-spending. This issue was mitigated by checking the order of the key image, which involves a full scalar multiplication, ironically diminishing the performance that Curve25519 was meant to provide.

The second example was recently discovered by Cremers and Jackson, who found more vulnerabilities in protocols caused by the non-trivial cofactor [22]. These vulnerabilities allowed attackers to bypass the authentication properties of the Secure Scuttlebutt Gossip protocol and Tendermint's secure handshake.

Why not both? In 2015, Hamburg presented the "Decaf" technique [28], which removes the cofactor of twisted Edwards curves through a clever encoding. He later refined the technique to "Ristretto" (see [1]), which is now proposed in

the crypto forum research group (CFRG) of IETF for standardization [46]. The Decaf and Ristretto encodings come at some computational cost and also added complexity of the implementation, but it eliminates the burden of handling the cofactor in protocol design.

However, there is another, much more obvious, approach to complete addition in prime-order elliptic-curve groups. It is long known that complete addition formulas also exist for Weierstraß curves [18], but those formulas were long regarded as too inefficient to offer an acceptable tradeoff. The situation changed in 2016, when Renes, Costello, and Batina revisited the approach from [18] and presented much more efficient complete addition formulas for Weierstraß curves [42]. Unfortunately, these formulas are still considerably less efficient than the incomplete addition formulas that possibly require handling of special cases. The performance gap is even larger compared to the complete addition formulas for twisted Edwards curves, and the complete differential additional used in the scalar-multiplication ladder on Montgomery curves.

Contributions of this Paper. If we assume the usage of complete addition formulas for both—twisted Edwards or Montgomery curves on one hand and Weierstraß curves on the other hand—the choice of curve becomes a tradeoff between performance and protocol simplicity. To fully understand this tradeoff, we need to know how large exactly the performance penalty is for using Weierstraß curves with the complete addition formulas from [42]. The standard approach to understand performance differences is to compare the speed of optimized implementations, ideally on different target architectures. Almost surprisingly however, there are no such optimized implementations of elliptic-curve scalar multiplication using complete formulas on Weierstraß curves. In this paper we present such implementations and answer the question about the actual cost of complete cofactor-1 ECC arithmetic using the formulas from [42].

More specifically, we present highly optimized software targeting three different microarchitectures for variable-basepoint scalar multiplication on a 255-bit Weierstraß curve over the field $\mathbb{F}_{2^{255}-19}$, the same field underlying Curve25519. Choosing a curve over the same field eliminates possible effects that are not due to the choice of curve shape and corresponding addition formulas, but due to differences in speed of the field arithmetic.

The three microarchitectures we are targeting are Intel's 64-bit Haswell generation of processors featuring AVX2 vector instructions; the earlier Intel Ivy Bridge processors that feature AVX vector instructions, but not yet the AVX2 integer-vector instruction set; and the ARM Cortex M4 family of 32-bit embedded microcontrollers. All our implementations follow the "constant-time" approach of avoiding secret branch conditions and secretly indexed memory access.

We compare our results to scalar multiplication from highly optimized X25519 software on the same microarchitectures. Perhaps surprisingly the performance penalty heavily depends on the microarchitecture; it ranges between a factor of only 1.47 on Intel Haswell and a factor of 2.87 on ARM Cortex M4.

Disclaimer. This paper revisits the discussion of the performance of Curve-25519 (and Curve448) relative to the old Weierstraß curves. We see a certain risk that the results in this paper may be misinterpreted one way or another, so we would like to clarify our intentions, and how we see the results of this paper:

- The motivation for this work has been that we saw a potentially interesting missing data point in the context of choosing elliptic curves for cryptographic applications.
- We do not think that any elliptic-curve standardization discussion should be re-opened. No result in this paper suggests that this would be useful and we believe that the choice of Curve25519 and Curve448 by IETF was a very sensible one. All effort that the community can invest in standardization is better placed in, for example, efforts to choose sensible post-quantum primitives.
- We see a rather common misconception of Weierstraß curves not having any (practical) complete addition formulas. For example, the book "Serious Cryptography" by Aumasson describes the ANSSI and Brainpool curves (both prime-order Weierstraß curves) as *"two families of curves that don't support complete addition formulas [...]"* [2, page 231]. In similar spirit, Bernstein and Lange on their "SafeCurves" website [12], dismiss Weierstraß curves entirely as a viable option for cryptographic applications based on the ground that complete addition is so much less efficient than incomplete addition formulas (and even less efficient than complete addition on twisted Edwards curves). In our opinion, a sensibly chosen Weierstraß curve using the complete addition formulas from [42] or [45] may well be the *safer* choice for protocols and applications that can live with the performance penalty.
- While we think that it is always preferable to use complete addition formulas for implementing Weierstraß-curve arithmetic, we would like to articulate that by no means we recommend the use of the Renes-Costello-Batina formulas above the use of Curve25519 with Ristretto for *new* protocols. Indeed, we support the proposal brought into CRFG by de Valence, Grigg, Tankersley, Valsorda, and Lovecruft [46] for the standardization of the Ristretto encoding.

Related Work. The most relevant related work for this paper can be grouped in two categories: papers presenting optimized implementations of Curve25519 and papers investigating performance of complete group addition on Weierstraß curves.

In the first category, the directly related papers present results for optimized scalar multiplication on Curve25519 targeting the same microarchitectures that we target in this paper. To the best of our knowledge, the current speed record for X25510 on Intel Sandy Bridge and Ivy Bridge processors is held by the "Sandy2x" software by Chou [19]. The speed record for the Intel Haswell microarchitecture is held by the software by Oliveira, López, Hışıl, Faz-Hernández, Rodríguez-Henríquez presented in [40]. This paper also presents even higher speeds for the Intel Skylake microarchitecture; that software makes use of the `MULX` and `ADCX/ADOX` instructions that are not available on Haswell. Finally, the speed record for scalar multiplication on Curve25519 on Cortex-M4 is held

by software presented by Haase and Labrique in [27]. We provide a comparison of our results with the results from those papers in Sect. 4.

In the second category we are aware of only three results: In [42], Renes Costello, and Batina provide benchmarks of scalar multiplication on various NIST-P curves [32] in OpenSSL [41] using their complete formulas and compare them with the "standard" incomplete formulas used by default in OpenSSL. This comparison shows a performance penalty of a factor of about 1.4. However, the figures in [42, Table 2] strongly suggest that the comparison did not use the optimized implementation of scalar multiplication on the NIST curves that would need to be enabled with the configure option `enable-ec_nistp_64_gcc_128` when building OpenSSL. In [37], Costa Massolino, Renes, and Batina present an FPGA implementation of scalar multiplication on arbitrary Weierstraß curves over prime-order fields using the complete formulas from [42]. They claim that their results are "competitive with current literature", but also state that "it is not straightforward to do a well-founded comparison among works in the literature". This is because hardware implementations have a much larger design space, not only with tradeoffs between area and speed, but also flexibility with regards to the supported elliptic curves (e.g., through different curve shapes or support for specialized or generic field arithmetic). Finally, in [45] Susella and Montrasio estimate performance of different scalar-multiplication approaches. They report estimates in terms of multiplications per scalar bit, assuming that multiplication costs as much as squaring and multiplication by (small) constants, and that addition costs 10% of a multiplication. In this metric the ladder from [45] is very slightly cheaper at 19.1 than scalar multiplication using the formulas from [42] at 19.33. However, if we understand correctly, the estimates for [42] are computed without taking into account *signed* fixed-window scalar multiplication. In this metric, the Montgomery ladder used in X25519 software would come at a cost of 10.8.

Notation. We use abbreviations **M** to refer to the cost of a finite-field multiplication, use **S** to refer to the cost of a squaring, **a** to refer to the cost of an addition, and $\mathbf{m_c}$ to refer to the cost of multiplication by a constant c.

Availability of Software. We place all software related to this paper into the public domain (to the maximum extent possible, using the Creative Commons CC0 waiver). The code packages are published through the public archive at https://doi.org/10.5281/zenodo.7494621.

Organization of this Paper. In Sect. 2 we give a brief review of the mathematical background on elliptic curves and in particular motivate our choice of curve, which we call Curve13318. In Sect. 3 we provide details of our implementations of constant-time variable-basepoint scalar multiplication on Curve13318 for Intel Sandy Bridge, Intel Haswell, and ARM Cortex M4. Section 4 presents the performance results and compares to state-of-the-art implementations of scalar multiplication on Curve25519. We conclude the paper and give an overview of possible future work in Sect. 5.

2 Preliminaries

2.1 Weierstraß, Montgomery, and Twisted Edwards Curves

The typical way to introduce elliptic curves over a field $\mathbb{F}$ with large characteristic is through the *short Weierstraß equation*

$$E_W : y^2 = x^3 + ax + b,$$

where $a, b \in \mathbb{F}$. As long as the discriminant $\delta = -16(4a^3 + 27b^2)$ is nonzero, this equation describes an elliptic curve and any elliptic curve over a field $\mathbb{F}$ with characteristic not equal to two or three can be described through such an equation. For cryptography we typically choose a field of large prime order p; the relevant group in the cryptographic setup is the group of $\mathbb{F}_p$-rational points $E(\mathbb{F}_p)$. Whenever we talk about "the order of an elliptic curve" in this paper we mean the order of this group. The typical way to use Weierstraß curves in cryptography is to pick curve parameter $a = -3$ for somewhat more efficient arithmetic and to represent a point $P = (x, y)$ in Jacobian coordinates $(X : Y : Z)$ with $(x, y) = (X/Z^2, Y/Z^3)$. Point addition is using the formulas from [13] (improving on [20]) and uses $11\mathbf{M} + 5\mathbf{S} + 9\mathbf{a}$. Most efficient doubling uses formulas from [4] that use $3\mathbf{M} + 5\mathbf{S} + 8\mathbf{a}$. Alternatively, one can use a ladder with differential additions, for example using the approach from [33] that costs $6\mathbf{M} + 6\mathbf{S} + 20.5\mathbf{a}$ per ladder step.

Using a ladder for scalar multiplication is also what Montgomery proposed in [39] for a different class of elliptic curves, so-called *Montgomery curves*. These are described through an equation of the form

$$E_M : by^2 = x^3 + ax^2 + x,$$

again with $a, b \in \mathbb{F}$. The "ladder step" consisting of one differential addition and one doubling costs $5\mathbf{M} + 4\mathbf{S} + 8\mathbf{a}$. The formulas were shown to be complete by Bernstein in the Curve25519 paper [6]. One peculiarity of the formulas is that they only involve the x-coordinate of a point. For Diffie-Hellman protocols this has the advantage of free point compression and decompression, but for signatures this involves extra effort to recover the y-coordinate.

The most efficient complete formulas for full addition (and doubling) are on *twisted Edwards* curves [8], i.e., curves with equation

$$E_{tE} : x^2 + y^2 = 1 + dx^2y^2.$$

For the special case of $a = -1$, the formulas from [29] need only $8\mathbf{M} + 8\mathbf{a}$ for addition and $4\mathbf{M} + 4\mathbf{S} + 6\mathbf{a}$ for doubling. If -1 is a square in $\mathbb{F}_p$ then the formulas are complete. Every twisted Edwards curves is birationally equivalent to a Montgomery curve [8, Thm. 3.2] and in the case of Curve25519 both shapes are used in protocols: the Montgomery shape and corresponding ladder for X25519 key exchange and the twisted Edwards shape for Ed25519 signatures.

2.2 Curve13318

The goal of this paper is to investigate the performance of complete addition and doubling on a Weierstraß curve and compare it to the performance of Curve25519. Many aspects contribute to the performance of elliptic-curve arithmetic and as we are mainly interested in the impact of formulas implementing the group law, we decided to choose a curve that is as similar to Curve25519 as possible, except that it is in Weierstraß form and has prime order. This means that in particular, we want a curve that

- is defined over the field $\mathbb{F}_p$ with $p = 2^{255} - 19$;
- is twist secure (for a definition, see [6] or [12]);
- has parameter $a = -3$ to support common speedups of the group law; and
- has small parameter b.

A curve with precisely these properties was proposed in May 2017 by Barreto on Twitter [3]. Specifically, he proposed the curve with equation

$$E : y^2 = x^3 - 3x + 13318,$$

defined over $\mathbb{F}_{2^{255}-19}$. In a follow-up tweet Barreto clarified that the selection criteria for this curve were "all old SafeCurves properties (with recent improvements) plus prime order". Barreto did not name this curve; we will in the following refer to it as Curve13318. This name at the same time points to the curve parameter b and its intended similarities to Curve25519. The order of the group of $\mathbb{F}_p$-rational points on Curve13318 is

$$N = \ell = 2^{255} + 325610659388873400306201440571661405155.$$

2.3 The Renes-Costello-Batina Formulas

In 2016, Renes, Costello, and Batina published a set of formulas for doubling and addition on short Weierstraß curves [42], based on previous work by Bosma and Lenstra [18]. The formulas are complete for all elliptic curves defined over a field with characteristic not equal to 2 or 3. Together with the formulas published by Susella and Montrasio in 2017 [45], the Renes-Costello-Batina formulas are the only set of addition formulas for prime-order Weierstraß curves that is proven to be complete.

Because we implement variable-basepoint scalar multiplication on a curve with $a = -3$, we will use the algorithms for addition and doubling from [42, Section 3.2]. The relevant complete formulas for (projective) point addition are

$$\begin{aligned}
X_3 &= (X_1Y_2 + X_2Y_1)\,(Y_1Y_2 + 3(X_1Z_2 + X_2Z_1 - bZ_1Z_2)) \\
&\quad - 3(Y_1Z_2 + Y_2Z_1)\,(b(X_1Z_2 + X_2Z_1) - X_1X_2 - 3Z_1Z_2)\,, \\
Y_3 &= 3(3X_1X_2 - 3Z_1Z_2)\,(b(X_1Z_2 + X_2Z_1) - X_1X_2 - 3Z_1Z_2) \\
&\quad + (Y_1Y_2 - 3(X_1Z_2 + X_2Z_1 - bZ_1Z_2))\,(Y_1Y_2 + 3(X_1Z_2 + X_2Z_1 - bZ_1Z_2))\,, \\
Z_3 &= (Y_1Z_2 + Y_2Z_1)\,(Y_1Y_2 - 3(X_1Z_2 + X_2Z_1 - bZ_1Z_2)) \\
&\quad + (X_1Y_2 + X_2Y_1)(3X_1X_2 - 3Z_1Z_2).
\end{aligned}$$

In [42], the formula for addition is implemented through 43 distinct operations, specifically $12\mathbf{M}+2\mathbf{m_b}+29\mathbf{a}$. The algorithm used to compute the addition (ADD) is listed in Algorithm 1.

Algorithm 1 Renes-Costello-Batina formula for $a = -3$. Used for exception-free addition on Curve13318.

procedure ADD($(X_1 : Y_1 : Z_1)$, $(X_2 : Y_2 : Z_2)$)

$v_1 \leftarrow X_1 \cdot X_2$
$v_2 \leftarrow Y_1 \cdot Y_2$
$v_3 \leftarrow Z_1 \cdot Z_2$
$v_4 \leftarrow X_1 + Y_1$
$v_5 \leftarrow X_2 + Y_2$
$v_6 \leftarrow v_4 \cdot v_5$
$v_8 \leftarrow v_6 - v_7$
$v_9 \leftarrow Y_1 + Z_1$
$v_{10} \leftarrow Y_2 + Z_2$
$v_{11} \leftarrow v_9 \cdot v_{10}$
$v_{13} \leftarrow v_{11} - v_{12}$
$v_{14} \leftarrow X_1 + Z_1$
$v_{15} \leftarrow X_2 + Z_2$
$v_{16} \leftarrow v_{14} \cdot v_{15}$
$v_{17} \leftarrow v_1 + v_3$
$v_{18} \leftarrow v_{16} - v_{17}$
$v_{19} \leftarrow b \cdot v_3$
$v_{20} \leftarrow v_{19} - v_{18}$
$v_{21} \leftarrow v_{20} + v_{20}$
$v_{22} \leftarrow v_{20} + v_{21}$
$v_{23} \leftarrow v_2 - v_{22}$
$v_{24} \leftarrow v_2 + v_{22}$
$v_{25} \leftarrow b \cdot v_{18}$
$v_{26} \leftarrow v_3 + v_3$
$v_{27} \leftarrow v_{26} + v_3$
$v_{28} \leftarrow v_{25} - v_{27}$
$v_{29} \leftarrow v_{28} - v_1$
$v_{30} \leftarrow v_{29} + v_{29}$
$v_{31} \leftarrow v_{30} + v_{29}$
$v_{33} \leftarrow v_1 + v_1$
$v_{33} \leftarrow v_{32} + v_1$
$v_{34} \leftarrow v_{33} - v_{27}$
$v_{35} \leftarrow v_{13} \cdot v_{31}$
$v_{36} \leftarrow v_{31} \cdot v_{34}$
$v_{37} \leftarrow v_{23} \cdot v_{24}$
$v_{38} \leftarrow v_{36} + v_{37}$
$v_{39} \leftarrow v_8 \cdot v_{24}$
$v_{40} \leftarrow v_{39} - v_{35}$
$v_{41} \leftarrow v_{13} \cdot v_{23}$
$v_{42} \leftarrow v_8 \cdot v_{33}$
$v_{43} \leftarrow v_{41} + v_{42}$

$X_3 \leftarrow v_{40}$
$Y_3 \leftarrow v_{38}$
$Z_3 \leftarrow v_{43}$

return $(X_3 : Y_3 : Z_3)$

Correspondingly, the complete formulas for doubling are

$$\begin{aligned} X_3 &= 2XY(Y^2 + 3(2XZ - bZ^2)) - 6XZ(2bXZ - X^2 - 3Z^2), \\ Y_3 &= (Y^2 - 3(2XZ - bZ^2))(Y^2 + 3(2XZ - bZ^2)) \\ &\quad + 3(3X^2 - 3Z^2)(2bXZ - X^2 - 3Z^2), \\ Z_3 &= 8Y^3Z. \end{aligned}$$

The cost of the doubling formulas is $8\mathbf{M} + 3\mathbf{S} + 2\mathbf{m_b} + 21\mathbf{a}$. The algorithm for doubling (DOUBLE) is listed in Algorithm 2.

We can reduce the cost of the doubling algorithm by erasing (some of) the multiplications v_1, v_4, v_6, v_{28}, using the rule that $2\alpha\beta = (\alpha+\beta)^2 - \alpha^2 - \beta^2$. By applying this rule, we trade $1\mathbf{M} + 1\mathbf{a}$ for $1\mathbf{S} + 3\mathbf{a}$. As we will describe in Sect. 3, this trick is beneficial only on the *Haswell* platform,

3 Implementation

In order to get a comprehensive benchmark for the performance of complete arithmetic on Curve13318, we optimized variable-basepoint scalar multiplication

Algorithm 2 Renes-Costello-Batina formula for $a = -3$. Used for exception-free doubling on Curve13318.

procedure DOUBLE($(X : Y : Z)$)

$v_1 \leftarrow X \cdot X$
$v_2 \leftarrow Y \cdot Y$
$v_3 \leftarrow Z \cdot Z$
$v_4 \leftarrow X \cdot Y$
$v_5 \leftarrow v_4 + v_4$
$v_6 \leftarrow X \cdot Z$
$v_7 \leftarrow v_6 + v_6$
$v_8 \leftarrow b \cdot v_3$
$v_9 \leftarrow v_8 - v_7$
$v_{10} \leftarrow v_9 + v_9$
$v_{11} \leftarrow v_{10} - v_9$
$v_{12} \leftarrow v_2 - v_{11}$
$v_{13} \leftarrow v_2 + v_{11}$
$v_{14} \leftarrow v_{12} \cdot v_{13}$
$v_{15} \leftarrow v_5 \cdot v_{12}$
$v_{16} \leftarrow v_3 + v_3$
$v_{17} \leftarrow v_3 + v_{16}$
$v_{18} \leftarrow b \cdot v_7$
$v_{19} \leftarrow v_{18} - v_{17}$
$v_{20} \leftarrow v_1 - v_{19}$
$v_{21} \leftarrow v_{20} + v_{20}$
$v_{22} \leftarrow v_{20} + v_{21}$
$v_{23} \leftarrow v_1 + v_1$
$v_{24} \leftarrow v_{23} + v_1$
$v_{25} \leftarrow v_{24} - v_{17}$
$v_{26} \leftarrow v_{22} \cdot v_{25}$
$v_{27} \leftarrow v_{14} + v_{26}$
$v_{28} \leftarrow Y \cdot Z$
$v_{29} \leftarrow v_{28} + v_{28}$
$v_{30} \leftarrow v_{22} \cdot v_{29}$
$v_{31} \leftarrow v_{15} - v_{30}$
$v_{32} \leftarrow v_2 \cdot v_{34}$
$v_{33} \leftarrow v_{32} + v_{32}$
$v_{34} \leftarrow v_{33} + v_{33}$

$X_3 \leftarrow v_{31}$
$Y_3 \leftarrow v_{27}$
$Z_3 \leftarrow v_{34}$

return $(X_3 : Y_3 : Z_3)$

on the Intel *Sandy Bridge* and *Haswell* microarchitectures, as well as the ARM *Cortex M4* processor.

The high-level structure of the scalar multiplication is shared among all three implementations. First—before operating on the key k—we validate the input point P, by checking whether P satisfies $y_P^2 = x_P^3 - 3x_P + 13318$. Because we have not defined any encoding for the neutral element $\mathcal{O}$, this check will implicitly validate that $P \neq \mathcal{O}$.

For the scalar-multiplication core, we use a left-to-right signed-window double-and-add algorithm, with $w = 5$. This algorithm is listed in Algorithm 3. The subroutine RECODESIGNEDWINDOW$_5$ computes a vector of coefficients $k' = (k'_0, \ldots, k'_{50})$, such that $k = k'_0 + 32k'_1 + \ldots, + 2^{250}k'_{50}$ and $k'_i \in \{-16, \ldots, 15\}$.

The table lookup is implemented in a traditional scanning fashion: selecting the required value using a bitwise AND operation. Where we use an unsigned representation, we compute the conditional negation of Y by negating Y and selecting the correct result using bitwise operations. When using floating points, we use a single XOR operation to conditionally flip the sign bit. These operations are—as well as the rest of the code—implemented in constant-time.

At the end of the double-and-add algorithm, we end up with a representation of $R = [k]P$ in projective coordinates. We compute the affine representation of x_R and y_R by computing the inverse of Z_R. Like most implementations of Curve25519 scalar multiplication, we use Fermat's little theorem and raise Z_R to the power $2^{255} - 21$ to obtain Z_R^{-1}. We chose not to exploit the optimization described in [16], because previous implementations have not had the

Algorithm 3 Signed double-and-add describe the used functions

1: **function** DoubleAndAdd(k, P)
2: $\mathbf{T} \leftarrow (\mathcal{O}, P, \ldots, [16]P)$ ▷ Precompute $([2]P, \ldots, [16]P)$
3: $k' \leftarrow$ RecodeSignedWindow$_5(k)$
4: $R \leftarrow \mathcal{O}$
5: **for** i **from** 50 **down to** 0 **do**
6: $R \leftarrow [32]R$ ▷ 5 point doublings
7: $Q \leftarrow T_{|k'_i|}$ ▷ Constant-time lookup from **T**
8: $Q \leftarrow (-1)^{k'_i} Q$ ▷ Constant-time conditional negation
9: $R \leftarrow R + Q$ ▷ Point addition
10: **return** R ▷ $R = (X_R : Y_R : Z_R)$

opportunity to implement this technique; exploiting this invention would give us an unfair advantage.

In the following subsections we describe the architecture-specific optimizations of field arithmetic required to implement the Renes-Costello-Batina formulas and in particular our vectorization strategy on Intel processors.

3.1 Sandy Bridge

The first implementation we present is based on the *Sandy Bridge* microarchitecture. Sandy Bridge is Intel's first microarchitecture featuring *Advanced Vector Extensions* (AVX). In addition to 2×-parallel 64-bit integer arithmetic, AVX supports 4×-parallel double-precision floating-point arithmetic. Because the multiplications and squarings in the Renes-Costello-Batina formulas can be conveniently grouped in batches of 4, we will be using the 4×-parallel floating-point arithmetic on 256-bit `ymm` vector registers.

Representation of Prime-Field Elements. Using doubles with 53-bit mantissa, we can emulate integer registers of 53 bits. To guarantee that no rounding errors occur in the underlying floating-point arithmetic, we use carry chains[2] to reduce the amount of bits in each register before performing operations that might overflow. Building on this approach, [6] recommends—but does not implement—*radix*-$2^{21.25}$ redundant representation, based on the arithmetic described in [5].

We use precisely this representation and represent a field element f through 12 signed double-precision floating-point values $f_0, \ldots, f_{11}$. For every f_i, its base b_i is defined by $b_i = 2^{\lceil 21.25i \rceil}$. Doubles already store their base in the exponent, which is large enough for our purposes. Therefore, we do not have to consider the base when evaluating f's value. Indeed, the value is computed by just computing the sum of the limbs:

$$f = \sum_{i=0}^{11} f_i$$

[2] Also called "coefficient reduction".

Coefficient Reduction. The Intel architecture supports no native modulo operation on floating points. Instead we extract a limb f_i's top bits by subsequently adding and subtracting a large constant $c_i = 3 \cdot 2^{51} b_{i+1}$, forcing the processor to discard the lower mantissa bits.

In code, each carry step needs 5 instructions to perform this routine. In Listing 1, the 4×-vectorized carry step from f_0 to f_1 is shown. To reduce f_{12} back to f_0, we multiply by $19 \cdot 2^{-255}$, which is implemented using a regular `vmulpd ymmX, [rel .reduceconstant]` instruction.

Listing 1 Single carry step for radix $2^{21.25}$ from limb f_0 to limb f_1.

```
1  ; Inputs:
2  ;    - ymm0: f0
3  ;    - [rel .precisionloss0]: times 4 dq 0x3p73 (c0 = 3 · 2^51 · 2^22)
4  ; Outputs:
5  ;    - ymm0: f0
6  ;    - ymm1: f1
7  vmovapd ymm14, yword [rel .precisionloss0]   ; load c0
8  vaddpd ymm15, ymm0, ymm14                     ; z' ← round(f0 + c0)
9  vsubpd ymm15, ymm15, ymm14                    ; t ← round(z' − c0)
10 vaddpd ymm1, ymm1, ymm15                      ; f1 ← round(f1 + t)
11 vsubpd ymm0, ymm0, ymm15                      ; f0 ← round(f0 − t)
```

All micro-operations (µops) corresponding to the arithmetic instructions in Listing 1 execute on port 1 of Sandy Bridge's back end. Furthermore, every `v{add,sub}pd` instruction has a latency of 3 cycles (cc). Consequently, the latency of one carry step is the sum of the latencies of instructions 2–4, i.e. the latency is $3 + 3 + 3 = 9$cc. Still, the reciprocal throughput is only 4cc.

In a sequential carry chain the back end is stalled most of the time due to data hazards. We expect a single carry chain to use $9 \cdot 14 = 126$cc or 31.5cc per lane. Even in a twice interleaved carry chain, the latency is still 63cc, while the reciprocal throughput is still only 56cc. In other words, the twice interleaved case still suffers from data hazards.

To overcome this, we implement a triple interleaved carry chain, as displayed in Fig. 1. In this case, the latency is reduced to 45cc, while the reciprocal throughput is 60cc. Conversely the bottleneck is not the latency, but the reciprocal throughput of the carry chain, of which the lower bound is 15cc per lane.

$$
\begin{aligned}
&f_0 \to f_1 \to f_2 \to f_3 \to f_4 \to f_5,\\
&f_4 \to f_5 \to f_6 \to f_7 \to f_8 \to f_9,\\
&f_8 \to f_9 \to f_{10} \to f_{11} \to f_0 \to f_1
\end{aligned}
$$

Fig. 1. Triple interleaved 12-limb carry chain.

Multiplication. For radix-$2^{21.25}$, we use basic 4× parallel Karatsuba multiplication [31], using inspiration from [30]. An inconvenience introduced by implementing Karatsuba using floating points, is that the shift-by-128-bit operations cannot be optimized out. Instead, we have to explicitly multiply some limbs by $2^{\pm 128}$. This costs 23 extra multiplication ops (implemented using 12 `vmulpd`s, and 11 `vandpd`s). Still, the Karatsuba implementation, which contains 131 `vmulpd` instructions, was measured to be 8% faster than the schoolbook method (which contains 155 `vmulpd` instructions).

Vectorization Strategy. We group the multiplications from both algorithms in three batches each, which have been chosen such that the complexity of the operations in-between the multiplications minimized. The resulting algorithms are given in Algorithms 4 and 5.

In particular, we cannot optimize the squaring operations in Double using the $2\alpha\beta = (\alpha + \beta)^2 - \alpha^2 - \beta^2$ rule, because $\alpha + \beta$ has too little headroom to be squared without doing an additional carry chain.

Because we cannot perform shift operations on floating-point values, and because the reciprocal throughput of `vmulpd` and `v{add,sub}pd` are both 1cc, we replace all chained additions by multiplications. This substitutes 8**a** for 4**m** in Add, and 10**a** for 5**m** in Double.

Last, we found that shuffling the `ymm` registers turns out to be relatively weak and expensive. That is because Sandy Bridge has no arbitrary shuffle instruction (like the `vpermq` instruction in AVX2). To shuffle every value in a `ymm` register into the correct lane, we would need at least two µops on port 5. Then it is cheaper to put all the values in the first lane, and accept that most of the additions and subtractions are not batched.

3.2 Haswell

The more recent *Haswell* microarchitecture from Intel supports *Advanced Vector Extensions 2* (AVX2). Haswell's AVX2 is more powerful than its predecessor. First, because AVX2 allows for 4× parallel 64-bit integer arithmetic; and second, because addition and subtraction operations—using the `vp{add,sub}q` instructions—have a reciprocal throughput of only 0.5cc. Together with the other instructions in AVX2, Haswell lends itself for efficient 4× parallel 64-bit integer arithmetic.

Algorithm 4 Algorithm for point addition for Curve13318 as implemented on the Sandy Bridge microarchitecture. A rule (——) denotes a "dead" value, i.e. one that has no meaning and is unused. RED executes a coefficient-reduction chain.

procedure ADD(X_1, Y_1, Z_1, X_2, Y_2, Z_2)

$v_{14} \leftarrow X_1 + Z_1$	$v_4 \leftarrow X_1 + Y_1$	$v_4 \leftarrow X_1 + Y_1$	$v_9 \leftarrow Y_1 + Z_1$
$v_{15} \leftarrow X_2 + Z_2$	$v_5 \leftarrow X_2 + Y_2$	$v_5 \leftarrow X_2 + Y_2$	$v_{10} \leftarrow Y_2 + Z_2$
$v_{16} \leftarrow v_{14} \cdot v_{15}$	$v_1 \leftarrow X_1 \cdot X_2$	$v_2 \leftarrow Y_1 \cdot Y_2$	$v_3 \leftarrow Z_1 \cdot Z_2$
$v_{16} \leftarrow \text{RED}(v_{16})$	$v_1 \leftarrow \text{RED}(v_1)$	$v_2 \leftarrow \text{RED}(v_2)$	$v_3 \leftarrow \text{RED}(v_3)$
$v_7 \leftarrow v_2 + v_1$	$v_{12} \leftarrow v_2 + v_3$	———	———
$v_{17} \leftarrow v_1 + v_3$	———	———	———
$v_{18} \leftarrow v_{16} - v_{17}$	———	———	———
$v_{19} \leftarrow b \cdot v_3$	———	———	———
$v_{20} \leftarrow v_{19} - v_{18}$	———	———	———
$v_{25} \leftarrow b \cdot v_{18}$	———	———	———
$v_{27} \leftarrow 3 \cdot v_3$	———	———	———
$v_{28} \leftarrow v_{25} - v_{27}$	———	———	———
$v_{29} \leftarrow v_{28} - v_1$	$v_{v_1 - v_3} \leftarrow v_1 - v_3$	———	———
$v_{22} \leftarrow 3 \cdot v_{20}$	———	$v_{31} \leftarrow 3 \cdot v_{29}$	$v_{34} \leftarrow 3 \cdot v_{v_1 - v_3}$
$v_{22} \leftarrow \text{RED}(v_{22})$	———	$v_{31} \leftarrow \text{RED}(v_{31})$	$v_{34} \leftarrow \text{RED}(v_{34})$
$v_{23} \leftarrow v_2 - v_{22}$	———	———	———
$v_{24} \leftarrow v_2 + v_{22}$	———	———	———
$v_{37} \leftarrow v_{23} \cdot v_{24}$	$v_{36} \leftarrow v_{31} \cdot v_{34}$	$v_6 \leftarrow v_4 \cdot v_5$	$v_{11} \leftarrow v_9 \cdot v_{10}$
$v_{37} \leftarrow \text{RED}(v_{37})$	$v_{36} \leftarrow \text{RED}(v_{36})$	$v_6 \leftarrow \text{RED}(v_6)$	$v_{11} \leftarrow \text{RED}(v_{11})$
$v_{37} \leftarrow v_{37} - 0$	$v_{36} \leftarrow v_{36} - 0$	$v_8 \leftarrow v_6 - v_7$	$v_{13} \leftarrow v_{11} - v_{12}$
$v_{38} \leftarrow v_{36} + v_{37}$	———	———	———
$v_{39} \leftarrow v_{24} \cdot v_8$	$v_{42} \leftarrow v_{33} \cdot v_8$	$v_{41} \leftarrow v_{23} \cdot v_{13}$	$v_{35} \leftarrow v_{31} \cdot v_{13}$
$v_{39} \leftarrow \text{RED}(v_{39})$	$v_{42} \leftarrow \text{RED}(v_{42})$	$v_{41} \leftarrow \text{RED}(v_{41})$	$v_{35} \leftarrow \text{RED}(v_{35})$
$v_{43} \leftarrow v_{41} + v_{42}$	———	———	———
$v_{40} \leftarrow v_{39} - v_{35}$	———	———	———

$X_3 \leftarrow v_{40}$
$Y_3 \leftarrow v_{38}$
$Z_3 \leftarrow v_{43}$

Representation of Prime-Field Elements. We use the *radix*-$2^{25.5}$ redundant representation, which was introduced in [15]. The representation stores an integer f into 10 *unsigned*[3] 64-bit limbs, with each base $b_i = 2^{\lceil 25.5i \rceil}$. Then the value of f is given by

[3] When we use *signed* limbs, we need—for the coefficient reduction—an instruction that shifts packed quadwords to the right while shifting in sign bits. Such an arithmetic shift operation—which would be called `vpsraq`—has never been implemented for the Haswell microarchitecture. Indeed, the first occurrence of the `vpsraq`-instruction was in AVX-512, in the Knight's Landing and Skylake-X microarchitectures.

Algorithm 5 Algorithm for point doubling for Curve13318 as implemented on the Sandy Bridge microarchitecture. A rule (——) denotes a "dead" value, i.e. one that has no meaning and is unused. RED executes a coefficient-reduction chain.

procedure DOUBLE(X, Y, Z)

$Y \leftarrow Y + 0$	$v_{2X} \leftarrow X + X$	——	——
$v_1 \leftarrow X \cdot X$	$v_6 \leftarrow X \cdot Z$	$v_3 \leftarrow Z \cdot Z$	$v_{28} \leftarrow Y \cdot Z$
$v_1 \leftarrow \text{RED}(v_1)$	$v_6 \leftarrow \text{RED}(v_6)$	$v_3 \leftarrow \text{RED}(v_3)$	$v_{28} \leftarrow \text{RED}(v_{28})$
$v_{24} \leftarrow 3 \cdot v_1$	$v_{18} \leftarrow 2b \cdot v_6$	$v_8' \leftarrow -\frac{b}{2} \cdot v_3$	$v_{17} \leftarrow 3 \cdot v_3$
$v_{25} \leftarrow v_{24} - v_{17}$	$v_{19} \leftarrow v_{18} - v_{17}$	——	——
$v_{20} \leftarrow v_1 - v_{19}$	——	——	——
$v_{22} \leftarrow -3 \cdot v_{20}$	——	——	——
$v_9 \leftarrow v_8' + v_6$	——	——	——
$v_{11} \leftarrow -6 \cdot v_9$	$v_{34} \leftarrow 8 \cdot v_{28}$	——	——
$v_{11} \leftarrow \text{RED}(v_{11})$	$v_{34} \leftarrow \text{RED}(v_{34})$	$v_{22} \leftarrow \text{RED}(v_{22})$	$v_{25} \leftarrow \text{RED}(v_{25})$
$v_{29} \leftarrow v_{28} + v_{28}$	——	——	——
$v_{30} \leftarrow v_{22} \cdot v_{29}$	$v_{26} \leftarrow v_{22} \cdot v_{25}$	$v_2 \leftarrow Y \cdot Y$	$v_5 \leftarrow v_{2X} \cdot Y$
$v_{30} \leftarrow \text{RED}(v_{30})$	$v_{26} \leftarrow \text{RED}(v_{26})$	$v_2 \leftarrow \text{RED}(v_2)$	$v_5 \leftarrow \text{RED}(v_5)$
$v_{12} \leftarrow v_2 - v_{11}$	——	——	——
$v_{13} \leftarrow v_2 + v_{11}$	——	——	——
$v_{32} \leftarrow v_2 \cdot v_{34}$	$v_{15} \leftarrow v_5 \cdot v_{12}$	$v_{14} \leftarrow v_{12} \cdot v_{13}$	——
$v_{32} \leftarrow \text{RED}(v_{32})$	$v_{15} \leftarrow \text{RED}(v_{15})$	$v_{14} \leftarrow \text{RED}(v_{14})$	——
$v_{31} \leftarrow v_{15} - v_{30}$	——	——	——
$v_{27} \leftarrow v_{14} + v_{26}$	——	——	——

$X_3 \leftarrow v_{31}$
$Y_3 \leftarrow v_{27}$
$Z_3 \leftarrow v_{34}$

$$f = \sum_{i=0}^{9} b_i f_i.$$

Coefficient Reduction. For coefficient reduction in radix $2^{25.5}$, we use the carry chain described by Sandy2x [19], adapted to AVX2. It is shown in Listing 2.

One carry step uses only 3 µops, each of which can execute on a separate port. Consequently, the reciprocal throughput of a single carry step is 1cc, while the latency of a carry step is 2cc. Therefore, it is optimal to implement the coefficient reduction using a twice interleaved carry chain, as visualized in Fig. 2.

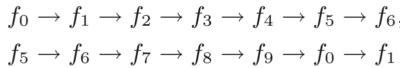

Fig. 2. Twice interleaved 10-limb carry chain.

Listing 2 Single carry step for radix $2^{25.5}$.

```
; Inputs:
;    - ymm0: f0
;    - [rel .MASK26]: times 4 dq 0x3FFFFFF
; Outputs:
;    - ymm0: f0
;    - ymm1: f1
vpsrlq ymm15, ymm0, 26                  ; t ← ⌊2^-26 f0⌋
vpaddq ymm1, ymm1, ymm15                ; f1 ← f1 + t
vpand ymm0, ymm0, yword [rel .MASK26]   ; f0 ← f0 mod 2^26
```

Multiplication and Squaring. For the multiplication of numbers in radix $2^{25.5}$, we adapt the multiplication routine from Sandy2x for AVX2. The multiplication routine uses the schoolbook method of multiplication. It contains 109 `vpmuludq` instructions. Of these 109 instructions, 9 `vpmuludq`s are used to precompute $19 \cdot \{g_1, g_2, \ldots, g_9\} \equiv 2^{255} \cdot \{g_1, g_2, \ldots, g_9\}$, where g is the second input operand. These values will be used for the limbs in the result that wrap around the field modulus, as is described in the Sandy2x paper ([19]).

In addition to the multiplication routine, we implement an optimized routine for squaring operations, based on the multiplication routine, that we will use in the next section.

Furthermore, we looked at the possibility of using Karatsuba multiplication, instead of using the schoolbook method. In this endeavor, we chose the Karatsuba *base* $B = 2^{153}$, i.e. we split the inputs into one part of 6 limbs, and one part of 4 limbs. We execute the Karatsuba algorithm to obtain a 19-limb value h_u, which is the *uncarried* result. To carry the high limbs from h_u onto the lower limbs, we multiply the high limbs by 19 using 3 `vpaddq`s and 1 `vpsllq` per limb; then we accumulate the results onto the lower limbs, yielding our carried product h.

In the Karatsuba routine, the port pressure is better divided, with 97 µops on port 0 and 149 µops on ports 1 and 5, relative to the schoolbook method, with 109 µops on port 0 and 90 on ports 1 and 5. However, the Karatsuba multiplication routine performs considerably worse than the Schoolbook method. Presumably, the CPU's front end cannot keep up with the added bulk of instructions.

Why not `mulx`? In [40], Oliveira et al. make use of the Bit Manipulation Instruction Set 2 (BMI2) extension in Haswell. BMI2 introduces the instruction `mulx`, which allows for unsigned multiply with arbitrary destination registers (instead of always storing the result in `rdx:rax`). Using this with a packed radix-2^{64} representation, field multiplication and squaring can be sped up quite a lot. However, experiments showed that the penalty introduced by more expensive additions/subtractions beat the performance gain achieved by using `mulx`.

Vectorization Strategy. Similar as in the implementation for Sandy Bridge, we batched all multiplications in the ADD algorithm into three different batches. The complete vectorization strategy is given in Algorithms 6 and 7.

In the implementation of DOUBLE, we applied the squaring trick described in Sect. 2.3, and rewrite $v_7 = 2XZ = (X+Z)^2 - X^2 - Z^2$. After replacing the multiplication v_6 by a squaring, we can replace the first of the three multiplication batches in DOUBLE with a batched squaring operation, that computes the values $\{(X+Z)^2, v_1, v_2, v_3\}$.

We realize that in both algorithms we can reduce the amount of shuffle operations needed, by unpacking the values from their SIMD registers after the first batched operations, and using the general-purpose instructions for many cheap operations, i.e. additions, subtractions, triplings[4], and multiplies with b. This way, we eagerly compute the core of the algorithm, leaving only two batched multiplications and a few additions. After repacking the values into the `ymm`-bank, we execute the remainder of the algorithm, including the two other multiplication batches.

3.3 ARM Cortex M4

Field Arithmetic. For the ARM Cortex M4, we reused the finite field arithmetic from Haase and Labrique [27]. For field elements, they use a packed representation in radix 2^{32}. We refer to their paper for the details of the field arithmetic, which can be summarized as cleverly exploiting the magnificent powers of the `umlal` and `umaal` instructions.

One function we added was `fe25519_mul_u32_asm`, used for multiplication with small constants. It was based on Fujii's code [25, Listing 3.2], which was in turn based on [43].

Application of Formulas. On top of the field arithmetic, we implemented the ADD and DOUBLE algorithms using function calls to the underlying field operations. Because—compared to multiplications—field additions are relatively expensive, there is no benefit in using the $2\alpha\beta = (\alpha+\beta)^2 - \alpha^2 - \beta^2$ trick. However, multiply-with-small-constant operations are relatively cheap, so we replaced any chained additions (like the $v_b \leftarrow v_a + v_a;$ then $v_a \leftarrow v_b + v_a$ pattern) by multiplications (i.e. $v_a \leftarrow 3v_a$). No other modifications were introduced. Even the order of the operations has been kept to the original.

[4] Using `lea r64, [2*r64 + r64]` instructions.

Algorithm 6 Algorithm for point addition for Curve13318 as implemented on the Haswell microarchitecture. A rule (——) denotes a "dead" value, i.e. one that has no meaning and is unused. RED executes a coefficient-reduction chain. The additions/subtractions with large constants ($2^{32}p$, $4p$ and $2^{37}p$) are to ensure that all the values are in the positive domain after subtraction.

procedure ADD(X_1, Y_1, Z_1, X_2, Y_2, Z_2)

$Y_1 \leftarrow \text{RED}(Y_1)$	$Y_2 \leftarrow \text{RED}(Y_2)$	$Y_1 \leftarrow \text{RED}(Y_1)$	$Y_2 \leftarrow \text{RED}(Y_2)$
$v_{14} \leftarrow X_1 + Z_1$	$v_4 \leftarrow X_1 + Y_1$	$v_4 \leftarrow X_1 + Y_1$	$v_9 \leftarrow Y_1 + Z_1$
$v_{15} \leftarrow X_2 + Z_2$	$v_5 \leftarrow X_2 + Y_2$	$v_5 \leftarrow X_2 + Y_2$	$v_{10} \leftarrow Y_2 + Z_2$
$v_{16} \leftarrow v_{14} \cdot v_{15}$	$v_1 \leftarrow X_1 \cdot Y_2$	$v_2 \leftarrow Y_1 \cdot Z_2$	$v_3 \leftarrow Z_1 \cdot Z_2$
$v_{16} \leftarrow \text{RED}(v_{16})$	$v_1 \leftarrow \text{RED}(v_1)$	$v_2 \leftarrow \text{RED}(v_2)$	$v_3 \leftarrow \text{RED}(v_3)$
$v_{17} \leftarrow v_1 + v_3$			
$v_7 \leftarrow v_1 + v_2$			
$v_{12} \leftarrow v_2 + v_3$			
$v_{18} \leftarrow v_{16} - v_{17}$			
$v_{19} \leftarrow b \cdot v_3$			
$v_{25} \leftarrow b \cdot v_{18}$			
$v_{20} \leftarrow v_{18} - v_{19}$			
$v_{22} \leftarrow 3 \cdot v_{20}$			
$v_{24} \leftarrow v_{22} + v_2$			
$v_{23} \leftarrow v_2 - v_{22}$			
$v_{27} \leftarrow 3 \cdot v_3$			
$v_{v_{25}-v_1} \leftarrow v_{25} - v_1$			
$v_{29} \leftarrow v_{v_{25}-v_1} - v_{27}$			
$v_{31} \leftarrow 3 \cdot v_{29}$			
$v_{33} \leftarrow 3 \cdot v_1$			
$v_{34} \leftarrow v_{33} - v_{27}$			
$v_{34} \leftarrow v_{34} + 2^{32}p$	$v_{24} \leftarrow v_{24} + 2^{32}p$	$v_{31} \leftarrow v_{31} + 2^{32}p$	$v_{23} \leftarrow v_{23} + 2^{32}p$
$v_{34} \leftarrow \text{RED}(v_{34})$	$v_{24} \leftarrow \text{RED}(v_{24})$	$v_{31} \leftarrow \text{RED}(v_{31})$	$v_{23} \leftarrow \text{RED}(v_{23})$
$v_{36} \leftarrow v_{34} \cdot v_{31}$	$v_{37} \leftarrow v_{24} \cdot v_{23}$	$v_6 \leftarrow v_4 \cdot v_5$	$v_{11} \leftarrow v_9 \cdot v_{10}$
$v_{36} \leftarrow \text{RED}(v_{36})$	$v_{37} \leftarrow \text{RED}(v_{37})$	$v_6 \leftarrow \text{RED}(v_6)$	$v_{11} \leftarrow \text{RED}(v_{11})$
$v_{38} \leftarrow v_{36} + v_{37}$			
——	——	$v_7 \leftarrow v_7 - 4p$	$v_{12} \leftarrow v_{12} - 4p$
——	——	$v_8 \leftarrow v_6 - v_7$	$v_{13} \leftarrow v_{11} - v_{12}$
$v_{42} \leftarrow v_8 \cdot v_{34}$	$v_{39} \leftarrow v_8 \cdot v_{24}$	$v_{35} \leftarrow v_{13} \cdot v_{31}$	$v_{41} \leftarrow v_{13} \cdot v_{23}$
——	$v_{39} \leftarrow v_{39} + 2^{37}p$	——	——
——	$v_{40} \leftarrow v_{39} - v_{35}$	——	——
$v_{43} \leftarrow v_{42} + v_{41}$	——	——	——
$v_{43} \leftarrow \text{RED}(v_{43})$	$v_{40} \leftarrow \text{RED}(v_{40})$	——	——

$X_3 \leftarrow v_{40}$

$Y_3 \leftarrow v_{38}$

$Z_3 \leftarrow v_{43}$

4 Performance Results

The complete scalar multiplication algorithm was tested and benchmarked on Intel Core i7-2600 (Sandy Bridge), Intel Core i5-3210 (Ivy Bridge), Intel Core i7-4770 (Haswell), and the ARM STM32F407 (Cortex-M4). On the Intel processors, all measurements were done with Turbo Boost disabled, all Hyper-Threading cores shut down, and with the CPU clocked at the maximum nominal frequency. The STM32F407 device was run with its default settings, as listed in the datasheet [44] (i.e. clocked from the 16MHz internal RC-oscillator). We list the benchmarking results in Table 1. As expected, none of our implementations exceed the performance of Curve25519.

Algorithm 7 Algorithm for point doubling for Curve13318 as implemented on the Haswell microarchitecture. A rule (———) denotes a "dead" value, i.e. one that has no meaning and is unused. RED executes a coefficient-reduction chain. The additions/subtractions with large constants ($2^{32}p$, $4p$ and $2^{37}p$) are to ensure that all the values are in the positive domain after subtraction.

procedure DOUBLE(X, Y, Z)

$v_{X+Z} \leftarrow X + Z$	$v_{X+Z} \leftarrow X + Z$	$v_{X+Z} \leftarrow X + Z$	$v_{X+Z} \leftarrow X + Z$
$v_{2Y} \leftarrow Y + Y$	$v_{2Y} \leftarrow Y + Y$	$v_{2Y} \leftarrow Y + Y$	$v_{2Y} \leftarrow Y + Y$
$v_{(X+Z)^2} \leftarrow v_{X+Z}^2$	$v_1 \leftarrow X^2$	$v_2 \leftarrow Y^2$	$v_3 \leftarrow Z^2$
$v_{(X+Z)^2} \leftarrow \textsc{Red}(v_{(X+Z)^2})$	$v_1 \leftarrow \textsc{Red}(v_1)$	$v_2 \leftarrow \textsc{Red}(v_2)$	$v_3 \leftarrow \textsc{Red}(v_3)$
$v_{Z^2+2XZ} \leftarrow v_{(X+Z)^2} - v_1$			
$v_7 \leftarrow v_{Z^2+2XZ} - v_3$			
$v_{18} \leftarrow b \cdot v_7$			
$v_8 \leftarrow b \cdot v_3$			
$v_{17} \leftarrow 3 \cdot v_3$			
$v_{19} \leftarrow v_{18} - v_{17}$			
$v_9 \leftarrow v_8 - v_7$			
$v_{24} \leftarrow 3 \cdot v_1$			
$v_{11} \leftarrow 3 \cdot v_9$			
$v_{20} \leftarrow v_{19} - v_1$			
$v_{22} \leftarrow 3 \cdot v_{20}$			
$v_{12} \leftarrow v_2 - v_{11}$			
$v_{13} \leftarrow v_2 + v_{11}$			
$v_{25} \leftarrow v_{24} - v_{17}$			
$v_{4v_2} \leftarrow 4 \cdot v_2$			
$v_{22} \leftarrow v_{22} + 2^{32}p$	$v_{12} \leftarrow v_{12} + 2^{32}p$	$v_{25} \leftarrow v_{25} + 2^{32}p$	$v_{13} \leftarrow v_{13} + 2^{32}p$
$v_{22} \leftarrow \textsc{Red}(v_{22})$	$v_{12} \leftarrow \textsc{Red}(v_{12})$	$v_{25} \leftarrow \textsc{Red}(v_{25})$	$v_{13} \leftarrow \textsc{Red}(v_{13})$
$v_{26} \leftarrow v_{22} \cdot v_{25}$	$v_{14} \leftarrow v_{12} \cdot v_{13}$	$v_{28} \leftarrow v_{2Y} \cdot Z$	$v_4 \leftarrow v_{2Y} \cdot X$
$v_{26} \leftarrow \textsc{Red}(v_{26})$	$v_{14} \leftarrow \textsc{Red}(v_{14})$	$v_{28} \leftarrow \textsc{Red}(v_{28})$	$v_4 \leftarrow \textsc{Red}(v_4)$
$v_{27} \leftarrow v_{26} + v_{14}$	———	———	———
$v_{30} \leftarrow v_{28} \cdot v_{22}$	$v_{15} \leftarrow v_4 \cdot v_{12}$	$v_{34} \leftarrow v_{4v_2} \cdot v_{28}$	———
$v_{30} \leftarrow v_{30} - 2^{37}p$	———	———	———
$v_{31} \leftarrow v_{15} - v_{30}$	———	———	———
$v_{31} \leftarrow \textsc{Red}(v_{31})$	$v_{34} \leftarrow \textsc{Red}(v_{34})$	———	———

$X_3 \leftarrow v_{31}$
$Y_3 \leftarrow v_{27}$
$Z_3 \leftarrow v_{34}$

Table 1. Measured cycle counts of the variable-basepoint scalar-multiplication routines on the Sandy Bridge (SB), Ivy Bridge (IB), Haswell (H) and Cortex M4 (M4) architectures.

Implementation	SB	IB	H	M4
Chou16 [19]	159 128[a]	156 995[a]	155 823[b]	–
Faz-Hernández-Lopez15 [24]	–	–	≈ 156 500[c]	–
OLHF18 [40]	–	–	138 963[a]	–
Fujii-Aranha19 [26]	–	–	–	907 240[a]
Haase-Labrique19 [27]	–	–	–	625 358[a]
Curve13318 (**this work**)	389 546[b]	382 966[b]	204 643[b]	1 797 451[b]
Ed25519 verify	221 988[d]	206 080[d]	184 052[d]	–
slowdown	2.45×	2.44×	1.47×	2.87×

[a] As reported in the respective publication.
[b] From own measurements.
[c] As reported in [24]. This publication expressed their benchmarks in kcc. As such, this value has been padded with zeros.
[d] Cycle counts reported on Bernstein and Lange's eBACS website [11]; included for the sake of completeness. The SB, IB and H measurements were selected from the tables for the `h6sandy`, `manny613` and `genji202` machines respectively. At the moment of writing, it is unclear to the authors which implementations were used to construct these cycle counts.

It can immediately be seen that the slowdown factor is dependent on the platform. In particular, the Haswell implementation of scalar multiplication on Curve13318 performs, also relatively speaking, much better than the others. The source of this is seems to be that Algorithms 1 and 2 lend themselves for very efficient 4-way parallelization, which is not supported by Curve25519's ladder algorithm. Through AVX2, 4-way parallelization is very powerful on Haswell, whereas on the other platforms it is not, at least not to the same extent. This makes it possible to write a Haswell implementation that is significantly faster than the others.

The Cost of Completeness

Another question we might be able to answer is if the factor-1.4 penalty claimed in [42]—for complete formulas vs. incomplete formulas—is realistic also for optimized implementations.

In [17], Bos, Costello, Longa, and Naehrig present performance results for scalar multiplication on a prime-order Weierstraß curve over $\mathbb{F}_{2^{256}-189}$ using parameter $a = -3$. The curve is very similar to Curve13318 and the implementation uses non-complete formulas for addition and doubling. The authors report 278 000 cycles for variable-base scalar multiplication on Intel Sandy Bridge. The software in [17] is seriously optimized, and claimed to run in constant time, so these 278 000 cycles are reasonably comparable to our 389 546 cycles with complete formulas. In other words, this comparison affirms the factor-1.4 performance-penalty claim from [42].

5 Future Work and Conclusion

Future work. Of course it might be possible to improve on our results for optimized arithmetic using the Renes-Costello-Batina formulas, but we would be surprised to see such improvements change the big picture and conclusion we draw in this paper. What would be interesting to explore is carefully optimized software for the complete ladder formulas presented in [45]. Our intuition is that in practice they will end up slightly slower than the signed fixed-window scalar multiplication using Renes-Costello-Batina formulas we employed here, but settling this question clearly needs more implementation effort.

Conclusion. The analysis in this paper shows that using prime-order Weierstraß curves with complete addition formulas is between ≈1.5 times and ≈2.9 times slower than using state-of-the-art Montgomery curve arithmetic. In an area where even a 10% improvement in performance is often considered important and worth publication in major venues, this is a pretty heavy price to pay; at least for some applications that are bottlenecked by ECC performance.

However, for applications that primarily aim at simplicity and safety against subgroup attacks, the performance penalty might be acceptable. This point of view is supported, for example, also by the fact that the attempt to standardize the high-performance "Four$\mathbb{Q}$" curve [21] in CFRG [35] was only very short lived. The discussion around this proposal acknowledged that Four$\mathbb{Q}$ offers considerably faster arithmetic than Curve25519, but questioned that there are any applications that really need that performance[5].

In our opinion, for the design of new protocols, the most efficient, simple, and safe choice of elliptic curve remains Curve25519 in twisted Edwards form with the Ristretto encoding to remove the non-trivial cofactor.

References

1. Arcieri, T., de Valence, H., Lovecruft, I.: The Ristretto Group. https://ristretto.group/ristretto.html. Accessed 31 July 2019
2. Aumasson, J.P.: Serious Cryptography: A Practical Introduction to Modern Encryption. No Starch Press, San Francisco (2017)
3. Barreto, P.S.L.M.: Tweet (2017). https://twitter.com/pbarreto/status/869103226276134912
4. Bernstein, D.J.: A software implementation of NIST P-224. In: Talk at the Workshop on Elliptic Curve Cryptography - ECC 2001 (2001). http://cr.yp.to/talks.html#2001.10.29
5. Bernstein, D.J.: Floating-point arithmetic and message authentication (2004). http://cr.yp.to/papers.html#hash127
6. Bernstein, D.J.: Curve25519: new Diffie-Hellman speed records. In: Yung, M., Dodis, Y., Kiayias, A., Malkin, T. (eds.) PKC 2006. LNCS, vol. 3958, pp. 207–228. Springer, Heidelberg (2006). https://doi.org/10.1007/11745853_14. http://cr.yp.to/papers.html#curve25519

[5] For the full discussion, see https://mailarchive.ietf.org/arch/msg/cfrg/sCqu86nFiAw_9beBXVqBM_zES_k.

7. Bernstein, D.J.: 25519 naming. Posting to the CFRG mailing list (2014). https://www.ietf.org/mail-archive/web/cfrg/current/msg04996.html
8. Bernstein, D.J., Birkner, P., Joye, M., Lange, T., Peters, C.: Twisted Edwards curves. In: Vaudenay, S. (ed.) AFRICACRYPT 2008. LNCS, vol. 5023, pp. 389–405. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68164-9_26. http://cr.yp.to/papers.html#twisted
9. Bernstein, D.J., Duif, N., Lange, T., Schwabe, P., Yang, B.-Y.: High-speed high-security signatures. In: Preneel, B., Takagi, T. (eds.) CHES 2011. LNCS, vol. 6917, pp. 124–142. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23951-9_9. See also full version [10]
10. Bernstein, D.J., Duif, N., Lange, T., Schwabe, P., Yang, B.Y.: High-speed high-security signatures. J. Cryptogr. Eng. **2**(2), 77–89 (2012). http://cryptojedi.org/papers/#ed25519. See also short version [9]
11. Bernstein, D.J., Lange, T.: eBACS: ECRYPT Benchmarking of Cryptographic Systems. https://bench.cr.yp.to/results-sign.html. Accessed 03 Oct 2019
12. Bernstein, D.J., Lange, T.: SafeCurves: choosing safe curves for elliptic-curve cryptography. https://safecurves.cr.yp.to. Accessed 31 July 2019
13. Bernstein, D.J., Lange, T.: Faster addition and doubling on elliptic curves. In: Kurosawa, K. (ed.) ASIACRYPT 2007. LNCS, vol. 4833, pp. 29–50. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-76900-2_3. https://cr.yp.to/papers.html#newelliptic
14. Bernstein, D.J., Lange, T., Schwabe, P.: On the correct use of the negation map in the Pollard rho method. In: Catalano, D., Fazio, N., Gennaro, R., Nicolosi, A. (eds.) PKC 2011. LNCS, vol. 6571, pp. 128–146. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19379-8_8. https://cryptojedi.org/papers/#negation
15. Bernstein, D.J., Schwabe, P.: NEON crypto. In: Prouff, E., Schaumont, P. (eds.) CHES 2012. LNCS, vol. 7428, pp. 320–339. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33027-8_19. https://cryptojedi.org/papers/#neoncypto
16. Bernstein, D.J., Yang, B.Y.: Fast constant-time GCD computation and modular inversion. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2019**(3), 340–398 (2019). https://tches.iacr.org/index.php/TCHES/article/view/8298
17. Bos, J.W., Costello, C., Longa, P., Naehrig, M.: Selecting elliptic curves for cryptography: an efficiency and security analysis. J. Cryptogr. Eng. **6**(4), 259–286 (2016). https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/selecting.pdf
18. Bosma, W., Lenstra, H.W.: Complete systems of two addition laws for elliptic curves. J. Number Theory **53**(2), 229–240 (1995). http://www.sciencedirect.com/science/article/pii/S0022314X85710888
19. Chou, T.: Sandy2x: new Curve25519 speed records. In: Dunkelman, O., Keliher, L. (eds.) SAC 2015. LNCS, vol. 9566, pp. 145–160. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-31301-6_8. https://www.win.tue.nl/~tchou/papers/sandy2x.pdf
20. Cohen, H., Miyaji, A., Ono, T.: Efficient elliptic curve exponentiation using mixed coordinates. In: Ohta, K., Pei, D. (eds.) ASIACRYPT 1998. LNCS, vol. 1514, pp. 51–65. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-49649-1_6
21. Costello, C., Longa, P.: Four$\mathbb{Q}$: four-dimensional decompositions on a $\mathbb{Q}$-curve over the Mersenne prime. In: Iwata, T., Cheon, J.H. (eds.) ASIACRYPT 2015. LNCS, vol. 9452, pp. 214–235. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48797-6_10. https://eprint.iacr.org/2015/565.pdf

22. Cremers, C., Jackson, D.: Prime, order please! Revisiting small subgroup and invalid curve attacks on protocols using Diffie-Hellman. Cryptology ePrint Archive, Report 2019/526 (2019). https://eprint.iacr.org/2019/526
23. Edwards, H.M.: A normal form for elliptic curves. Bull. (New Series) Am. Math. Soc. **44**(3), 393–422 (2007). https://www.ams.org/journals/bull/2007-44-03/S0273-0979-07-01153-6/S0273-0979-07-01153-6.pdf
24. Faz-Hernández, A., López, J.: Fast implementation of Curve25519 using AVX2. In: Lauter, K., Rodríguez-Henríquez, F. (eds.) LATINCRYPT 2015. LNCS, vol. 9230, pp. 329–345. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22174-8_18
25. Fujii, H.: Efficient Curve25519 implementation for ARM microcontrollers. Master's thesis, Universidade Estadual de Campinas (2018). http://taurus.unicamp.br/bitstream/REPOSIP/332957/1/Fujii_Hayato_M.pdf
26. Fujii, H., Aranha, D.F.: Curve25519 for the Cortex-M4 and beyond. In: Lange, T., Dunkelman, O. (eds.) LATINCRYPT 2017. LNCS, vol. 11368, pp. 109–127. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25283-0_6. http://www.cs.haifa.ac.il/~orrd/LC17/paper39.pdf
27. Haase, B., Labrique, B.: AuCPace: Efficient verifier-based PAKE protocol tailored for the IIoT. IACR Trans. Cryptogr. Hardw. Embed. Syst. 1–48 (2019). https://tches.iacr.org/index.php/TCHES/article/view/7384
28. Hamburg, M.: Decaf: eliminating cofactors through point compression. In: Gennaro, R., Robshaw, M. (eds.) CRYPTO 2015. LNCS, vol. 9215, pp. 705–723. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-47989-6_34. https://www.shiftleft.org/papers/decaf/
29. Hisil, H., Wong, K.K.-H., Carter, G., Dawson, E.: Twisted Edwards curves revisited. In: Pieprzyk, J. (ed.) ASIACRYPT 2008. LNCS, vol. 5350, pp. 326–343. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89255-7_20. http://eprint.iacr.org/2008/522/
30. Hutter, M., Schwabe, P.: Multiprecision multiplication on AVR revisited. J. Cryptogr. Eng. **5**(3), 201–214 (2015). http://cryptojedi.org/papers/#avrmul
31. Karatsuba, A., Ofman, Y.: Multiplication of multidigit numbers on automata. Soviet Physics Doklady **7**, 595–596 (1963). Translated from Doklady Akademii Nauk SSSR, **145**(2), 293–294, July 1962
32. Kerry, C.F., Director, C.R.: FIPS PUB 186–4 federal information processing standards publication digital signature standard (DSS) (2013). http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf
33. Kim, K.H., Choe, J., Kim, S.Y., Kim, N., Hong, S.: Speeding up elliptic curve scalar multiplication without precomputation. Cryptology ePrint Archive, Report 2017/669 (2017). https://eprint.iacr.org/2017/669.pdf
34. Koblitz, N.: Elliptic curve cryptosystems. Math. Comput. **48**, 209–209 (1987). https://www.ams.org/journals/mcom/1987-48-177/S0025-5718-1987-0866109-5/S0025-5718-1987-0866109-5.pdf
35. Ladd, W., Longa, P., Barnes, R.: Curve4Q. IETF CFRG Internet Draft (2017). https://tools.ietf.org/html/draft-ladd-cfrg-4q-00. Accessed 18 Aug 2019
36. luigi1111, Spagni, R. ("fluffypony"): Disclosure of a major bug in CryptoNote based currencies. Post on the Monero website (2017). https://www.getmonero.org/2017/05/17/disclosure-of-a-major-bug-in-cryptonote-based-currencies.html. Accessed 31 Aug 2019

37. Massolino, P.M.C., Renes, J., Batina, L.: Implementing complete formulas on Weierstrass curves in hardware. In: Carlet, C., Hasan, M.A., Saraswat, V. (eds.) SPACE 2016. LNCS, vol. 10076, pp. 89–108. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49445-6_5. https://eprint.iacr.org/2016/1133.pdf
38. Miller, V.S.: Use of elliptic curves in cryptography. In: Williams, H.C. (ed.) CRYPTO 1985. LNCS, vol. 218, pp. 417–426. Springer, Heidelberg (1986). https://doi.org/10.1007/3-540-39799-X_31
39. Montgomery, P.L.: Speeding the Pollard and elliptic curve methods of factorization. Math. Comput. **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
40. Oliveira, T., López, J., Hışıl, H., Faz-Hernández, A., Rodríguez-Henríquez, F.: How to (pre-)compute a ladder. In: Adams, C., Camenisch, J. (eds.) SAC 2017. LNCS, vol. 10719, pp. 172–191. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-72565-9_9. https://eprint.iacr.org/2017/264.pdf
41. OpenSSL: Cryptography and SSL/TLS toolkit. http://www.openssl.org/. Accessed 18 Aug 2019
42. Renes, J., Costello, C., Batina, L.: Complete addition formulas for prime order elliptic curves. In: Fischlin, M., Coron, J.-S. (eds.) EUROCRYPT 2016. LNCS, vol. 9665, pp. 403–428. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49890-3_16. http://eprint.iacr.org/2015/1060
43. Santis, F.D., Sigl, G.: Towards side-channel protected X25519 on ARM Cortex-M4 processors. In: SPEED-B – Software performance enhancement for encryption and decryption, and benchmarking (2016). https://ccccspeed.win.tue.nl/papers/SPEED-B_Final.pdf
44. STMicroelelectronics: RM0090 reference manual (2019). https://www.st.com/content/ccc/resource/technical/document/reference_manual/3d/6d/5a/66/b4/99/40/d4/DM00031020.pdf/files/DM00031020.pdf/jcr:content/translations/en.DM00031020.pdf
45. Susella, R., Montrasio, S.: A compact and exception-free ladder for all short Weierstrass elliptic curves. In: Lemke-Rust, K., Tunstall, M. (eds.) CARDIS 2016. LNCS, vol. 10146, pp. 156–173. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-54669-8_10
46. de Valence, H., Grigg, J., Tankersley, G., Valsorda, F., Lovecruft, I.: The ristretto255 group. IETF CFRG Internet Draft (2019). https://tools.ietf.org/html/draft-hdevalence-cfrg-ristretto-01. Accessed 31 July 2019