Efficient Elliptic Curve Operations On Microcontrollers With Finite Field Extensions

Thomas Pornin

NCC Group, `thomas.pornin@nccgroup.com`

3 January 2020

Abstract. In order to obtain an efficient elliptic curve with 128-bit security and a prime order, we explore the use of finite fields $GF(p^n)$, with p a small modulus (less than 2^{16}) and n a prime. Such finite fields allow for an efficient inversion algorithm due to Itoh and Tsujii, which we can leverage to make computations on an ordinary curve (short Weierstraß equation) in affine coordinates. We describe a very efficient variant of Montgomery reduction for computations modulo p, and choose $p = 9767$ and $n = 19$ to better map the abilities of small microcontrollers of the ARM Cortex-M0+ class. Inversion cost is only six times the cost of multiplication. Our fully constant-time implementation of curve point multiplication runs in about 4.5 million cycles (only 1.29 times slower than the best reported Curve25519 implementations); it also allows for efficient key pair generation (about 1.9 million cycles) and Schnorr signature verification (about 5.6 million cycles). Moreover, we describe variants of the Itoh-Tsujii algorithms that allow fast computations of square roots and cube roots (in less than twenty times the cost of a multiplication), leading to efficient point compression and constant-time hash-to-curve operations with Icart's map.

1 Introduction

This article explores the use of affine coordinates for fast and secure implementation of an elliptic curve defined over a carefully chosen finite field.

Affine Coordinates. In a finite field $GF(p^n)$ with characteristic $p \geq 5$ and extension degree $n \geq 1$, all elliptic curves can be expressed as sets of points $(x, y) \in GF(p^n) \times GF(p^n)$ that fulfill the short Weierstraß equation:

$$y^2 = x^3 + ax + b$$

for two constants a and b in $GF(p^n)$ such that $4a^3 + 27b^2 \neq 0$; an extra point (denoted $\mathbb{O}$), called the "point at infinity" and with no defined coordinates, is adjoined to the curve and serves as neutral element for the group law on the curve. The formulas for point addition are well-known: for points $Q_1 = (x_1, y_1)$ and $Q_2 = (x_2, y_2)$, we have:

$$\begin{aligned} -Q_1 &= (x_1, -y_1) \\ Q_1 + Q_2 &= (x_3, y_3) \\ &= (\lambda^2 - x_1 - x_2, \lambda(x_1 - x_3) - y_1), \text{ where:} \\ \lambda &= \begin{cases} \dfrac{y_2 - y_1}{x_2 - x_1} & \text{if } Q_1 \neq \pm Q_2 \\ \dfrac{3x_1^2 + a}{2y_1} & \text{if } Q_1 = Q_2 \end{cases} \end{aligned}$$

The quantity λ is the slope of the line that contains Q_1 and Q_2. The representation of a point Q as two field elements x and y is called *affine coordinates*.

Fractions and Coordinate Systems. Computing a point addition with these formulas involves a division in $GF(p^n)$. This operation is usually quite expensive, traditionally estimated at about 80 times the cost of a multiplication in the field. For much better performance, it is customary to replace coordinates with fractions, which makes inversion free (the numerator and denominator are swapped) but increases the cost of multiplications (both numerators and denominators must be multiplied), and greatly increases the cost of additions (in all generality, the addition of two fractions involves three multiplications and one addition in the field). The idea is that if a large number of curve point additions are to be computed (e.g. as part of a point multiplication routine), then the whole computation can be done with fractions, and only at the end are divisions needed to obtain the affine result.

The fractions are often expressed as systems of coordinates, such as *projective coordinates*, in which point (x, y) is represented by the triplet $(X{:}Y{:}Z)$ such that $x = X/Z$ and $y = Y/Z$; in essence, these are fractions such that the same denominator is used for x and y. Another popular choice is *Jacobian coordinates* that instead use $x = X/Z^2$ and $y = Y/Z^3$. Vast amounts of research efforts have been invested in finding systems of coordinates and point addition formulas that minimize cost (see [10] for a database of such formulas)[1].

Alternate Curve Types and Cofactors. An additional strategy to optimize performance of elliptic curve operations has been to seek alternate curve equations that, combined with an adequate system of coordinates, yield formulas with fewer multiplications and squarings. In particular, Montgomery curves ($by^2 = x^3 + ax^2 + x$) and twisted Edwards curves ($ax^2 + y^2 = 1 + dx^2y^2$) offer better performance than short Weierstraß curves. However, this comes at a price: these faster curves cannot have a prime order, since they necessarily contain points of order 2. These curves are chosen to have order hm, where m is a large prime, and h is the *cofactor*. For the well-known curves Curve25519 (Montgomery curve) and Edwards25519 (a twisted Edwards curve which is birationally equivalent to Curve25519), the cofactor is $h = 8$.

[1]Coordinate systems can also be interpreted geometrically; e.g. projective coordinates are part of the generic treatment of projective geometry, while Jacobian coordinates can be seen as jumps between isomorphic curves. However, from an implementation performance point of view, what ultimately matters is that most divisions are avoided, and the number of multiplications is minimized.

An unfortunate consequence is that some extra complexity is induced in protocols that use such curves, in order to avoid subtle weaknesses. For instance, in the EdDSA algorithm, as specified in RFC 8032[36] (and following the original definition paper [9], which already contains that assertion), the verification process of a signature ends with the following item:

> *Check the group equation* $8sB = 8R + 8kA'$. *It's sufficient, but not required, to instead check* $sB = R + kA'$.

This means that it is possible to (maliciously) craft a public/private key pair, and a signature on some message, such that some implementations will use the first equation and accept the signature, and others will use the second equation and reject it. This does not contradict the core properties of signature algorithms, but it is sufficient to induce forks in distributed applications that rely on several systems following consensus rules and accepting or rejecting exactly the same data elements.

More severe issues coming from non-trivial cofactors have also been reported (e.g. [41]; see also [20]). In general, most if not all protocols that use elliptic curves can be made safe against such issues by sprinkling some multiplications by the cofactor here and there, but the exact analysis is complex and fraught with subtle details. It follows that, all other things being equal, having a prime order (i.e. cofactor $h = 1$) is a much desirable property.

Prime Order Curve Strategies. From a twisted Edwards curve with order $8m$ (for a big prime m), a group of order m can be obtained with the Ristretto map, designed by M. Hamburg (see [4] for details). Group elements are internally represented by points on the curve, but the map encoding and decoding processes ensure proper filtering and normalization. Compared to operations on the curve itself, the map implies a small but nonnegligible computational overhead.

In [54], Schwabe and Sprenkels explore the overhead implied by the use of a prime-order short Weierstraß curve, through performing benchmarks on such a curve defined over the same field as Curve25519 and Edwards25519. They rely on traditional projective coordinates, along with formulas described in [53]; these formulas are not the fastest available, but they are *complete*, i.e. they produce the correct results on all inputs with no special cases. They obtain, as expected, worse performance than Curve25519 and Edwards25519.

In this paper, we explore an alternate avenue. As explained above, all the efforts on formulas, coordinate systems and curve equations take place under the assumption that field inversions are desperately inefficient and only one or two such inversions may happen over a full curve point multiplication. Here, we instead focus on finding a finite field where operations are efficient, and in particular such that inversions are not especially slow. We will show in the following sections a finite field appropriate for defining secure curves, such that additions, multiplications and inversions are fast. Our chosen field is $GF(9767^{19})$; on our target implementation platform (the ARM Cortex-M0+), multiplications and squarings are of similar performance to the best reported implementations in finite field $GF(2^{255} - 19)$ (the finite field used in Curve25519), while inversion cost is about six times the cost of multiplication (6M). We also obtain fast quadratic residue test (5.9M), square root extraction (17.1M) and cube root extraction (19.8M), allowing for very efficient point compression and hash-to-curve operations.

Yet Another Curve? Elliptic-curve cryptography is already rich with many curves, in fact too many for comfort. Implementing elliptic curve operations generically is possible, but usually yields substantially lower performance than implementations optimized for a specific finite field, and a specific curve equation. Generic curve implementations are also rarely constant-time, i.e. they may leak information on secret elements through timing-based side channels (including cache attacks). Specialized implementations, however, are specific to a single curve; supporting many different curves securely and efficiently thus requires large amounts of code. Consequently, there is a push for the reduction of the number of "standard curves". For instance, in the context of TLS, client and server negotiate elliptic curves with a handshake extension which initially contained no fewer than 25 possible curves, not counting the possibility of sending arbitrary curve equation parameters explicitly[13]. A later revision of the standard reduced that number to just 5, deprecating use of all other curves, as well as the ability to send explicit curve equation parameters[46].

In that respect, defining another curve is counterproductive. The goal of this paper is not to push for immediate adoption and standardisation of our curve; rather, it is an exploration of the concept of keeping to affine coordinates on a field where inversions (and also square roots and cube roots) are efficient. We see this new curve as a starting point for further research, especially beyond the basic curve point multiplication operations; for instance, as will be detailed in this article, fast square and cube roots allow for very efficient hash-to-curve operations, making more attractive protocols that entail many such hashing operations.

We still took care to write our implementations with a clean API, amenable to integration in applications, and with well-defined encoding and decoding rules, for the following reasons:

- Making the effort of writing full implementations guarantees that we did not forget any part that would be required in practice.
- The closer to production-ready structure implementations get, the more meaningful and precise benchmarks become.
- Creations often escape from their creator, and once code has been published, especially with an open-source license, there is no way to prevent it from being reused by anybody. A responsible cryptographic code writer should ensure that any such published code is harmless, i.e. secure enough not to induce catastrophic weaknesses if deployed by the unwary.

Target Platform. In the current article, we focus on low-end platforms, in particular the ARM Cortex-M0+ CPU, a popular microcontroller core because of its compactness and very low energy usage[2]. The techniques we develop are specially meant to map well to what that CPU offers. However, we will see that such optimization does not necessarily forfeit performance on larger systems, especially since, for instance, modern large CPU have SIMD units that can perform many "small" operations (e.g. 16-bit multiplications) in parallel.

Baseline for any talk about performance is Curve25519 (and its twisted Edwards counterpart Edwards25519). Curve25519 was described in [8] and is a Montgomery curve defined over field $GF(2^{255} - 19)$.

[2]The ARM Cortex-M0+ is an improvement over the previous ARM Cortex-M0; however, they do not differ in their timing characteristics for the operations described here. We also consider the variant with the 1-cycle fast multiplier, not the smaller 32-cycle slow multiplier.

It should be noted that there are recent works that claim much better performance than Curve25519 on small systems, in particular the Four$\mathbb{Q}$ curve[19]. In [58], a point multiplication cost under 2 million cycles on an ARM Cortex-M0+ is reported, about 55% of the cost of the best reported Curve25519 implementation on the same CPU. However, this particular curve has some extra structure (a low-degree endomorphism) which speeds up the computation but is slightly controversial, because cryptographers are often wary of extra structures, especially in elliptic curves (historically, many attacks on special curves exploited their extra structure), and because of the unclear intellectual property status of that endomorphism. Research on Four$\mathbb{Q}$ should be closely followed, but it is still too recent to serve as a reliable comparison point.

Article Organization. The article outline is the following:

– Section 2 describes the criteria we used for finding the finite field in which we will define a curve.
– In section 3, we explain how operations in the finite field, in particular inversions and square roots, can be implemented efficiently.
– Section 4 includes the definition our new elliptic curve, Curve9767; we explain how operations are optimized. We provide a generic, constant-time, complete point addition routine, as well as optimized constant-time point multiplication functions, for the three situations commonly encountered in cryptographic protocols (multiplying the conventional generator G, multiplying a dynamically obtained point Q, and a combined double-multiplication $uG + vQ$ used in ECDSA and Schnorr signature verification). Efficient routines for point compression and decompression, and hash-to-curve operations, are also provided.
– Implementation issues, including performance benchmarks, side channel attacks and countermeasures, and ideas for optimizations on other architectures, are detailed in section 5.
– In appendix A, we list a few extra facts and ideas that turned out not to work as well as expected, or have redhibitory flaws, but are still worth exposing because they could lead to useful results in other contexts.

All of our source code (reference C code, ARM assembly, test vectors, and support scripts) is available on:

`https://github.com/pornin/curve9767`

2 Finding The Field

2.1 Field Type

A finite field has cardinal p^n for a prime p (the field characteristic). Most modern elliptic curves use a prime field (i.e. $n = 1$). Here, we focus on extension fields ($n > 1$). We furthermore investigate only characteristics $p \geq 5$ (when $p = 2$ or $p = 3$, curve equations are different; moreover, an ordinary curve over $GF(2^n)$ must have an even order, and hence cannot have cofactor $h = 1$). Use of extension fields for defining elliptic curves with efficient implementations has been described under the name "optimal extension fields"[44,6]; however, we diverge from OEF in some aspects, explained below.

When $n > 1$, $GF(p^n)$ is defined by first considering the polynomial ring $GF(p)[z]$ (in all of this article, z denotes the symbolic variable for polynomials). We then make a quotient ring by computing operations on polynomials modulo a given monic polynomial $\mathcal{M}$ of degree n. When $\mathcal{M}$ is irreducible over $GF(p)$, this defines a field of cardinal p^n. Two finite fields with the same cardinal are isomorphic to each other, and the isomorphisms can be efficiently computed in both directions; therefore, the choice of the exact field (i.e. of the modulus $\mathcal{M}$) is irrelevant to security, and we are free to use a modulus that favours efficient implementation.

For fast inversion and square roots, as will be explained in section 3, we prefer to have $\mathcal{M} = z^n - c$ for some constant $c \in GF(p)$. Since $\mathcal{M}$ must be irreducible, c cannot be 0, 1 or -1.

2.2 Extension Degree

The extension degree n can be any integer. However, if the degree is very small, or is composite with a very small divisor, then attacks on elliptic curves based on Weil descent may apply. Weil descent is a generic process through which a discrete logarithm problem (DLP) on an algebraic curve over a field K is transformed into a DLP on another curve of higher degree, over a subfield K' of K. The latter problem may be easier to solve.

Gaudry, Hess and Smart[29] have applied that idea to elliptic curves over $GF(2^n)$; the GHS attack solves DLP on ordinary elliptic curves faster than generic attacks provided that the degree n is composite and with small factors. Application of the GHS attack to curve fields with odd characteristic is non-trivial[5]. Gaudry[28] has shown that if $n = 0 \bmod 4$ (i.e. $GF(p^n)$ is a quartic extension) then there exists an algorithm that solves DLP in asymptotic time $O(p^{3n/8})$, i.e. faster than the generic attack (Pollard's Rho algorithm, $O(p^{n/2})$). Conversely, Diem[21] showed that if the n is prime and not lower than 11, then the GHS attack cannot work.

It should be noted that the GHS attack is not necessarily the only way to leverage Weil descent in order to break DLP on elliptic curves. Moreover, most known results are asymptotic in nature, and the extent of their applicability to practical cases (with curve order sizes of about 256 bits) is not fully known at this point. However, it seems that using a prime extension degree $n \geq 11$ provides adequate protection against Weil descent attacks. We will use this criterion for our field selection. In that respect, we diverge from OEF, which typically use smaller and/or composite degrees.

2.3 Delayed Modular Reduction

Our main target system is the ARM Cortex-M0+. That CPU has a very fast multiplier on 32-bit operands, working in a single clock cycle. However, it returns only the low 32 bits of the result, i.e. it computes modulo 2^{32}. There is no opcode yielding the upper 32 bits of the product. Since we will need to perform multiplications in $GF(p)$, this seems to limit the range of p to values of at most 16 bits[3].

[3] This is not entirely true, as will be explained in section A.1.

Moreover, computations in $GF(p)$ also involve reduction modulo p, which will be substantially more expensive than the product itself. It is thus advantageous to mutualize modular reductions. Consider the product $w = uv$ where u and v are elements of $GF(p^n)$. Elements of $GF(p^n)$ are polynomials of degree less than n, and we denote the coefficients of u as (u_i) (for $i = 0$ to $n - 1$). The modulus for the definition of $GF(p^n)$ is $\mathcal{M} = z^n - c$. We then have the following:

$$\begin{aligned}
w_0 &= u_0v_0 + c(u_1v_{n-1} + u_2v_{n-2} + u_3v_{n-3} + \cdots + u_{n-1}v_1)\\
w_1 &= u_0v_1 + u_1v_0 + c(u_2v_{n-1} + u_3v_{n-2} + \cdots + u_{n-1}v_2)\\
w_2 &= u_0v_2 + u_1v_1 + u_2v_0 + c(u_3v_{n-1} + \cdots + u_{n-1}v_3)\\
&\ldots
\end{aligned}$$

There is a quadratic[4] number of products in $GF(p)$; however, we can also compute each w_i over plain integers, and making only one final reduction for each w_i, thereby lowering the number of reductions to n. This strategy is valid as long as the intermediate values (each w_i before reduction) fits in a machine word, i.e. 32 bits. We see that the largest potential pre-reduction value is for the computation of w_0; if each u_i and v_i is an integer less than p, then the intermediate value may range up to $(1 + c(n - 1))(p - 1)^2$. We will therefore look for a prime p and degree n such that $(1 + c(n - 1))(p - 1)^2 < 2^{32}$.

2.4 Fast Modular Reduction

Even if delaying modular reduction allows us to perform only n such operations for a product of two elements in $GF(p^n)$, they still constitute a nonnegligible cost; thus, reduction modulo p should be made as efficient as possible. In the OEF analysis[6], moduli very close to a power of 2 are favoured; however, this restricts the number of candidate moduli. In order to have a larger range of potential values for p, we instead use Montgomery reduction[45].

Let $s \in \mathbb{N}$ such that $p < 2^s$. We define $R = 2^s$, and $f = -1/p \bmod 2^s$. For an integer $x \in \mathbb{N}$, we can compute $x/R \bmod p$ as follows:

1. Let $t = xf \bmod 2^s$.
2. Let $t' = x + tp \bmod 2^s$.
3. Return $t'/2^s$.

Indeed, we can see that $x + tp = 0 \bmod 2^s$; hence, the division by 2^s in the third step is exact. Since p is relatively prime to 2^s, it follows that the result is correct. Morever, if $x < p^2$, then the result t_2 is less than $2p$ and can be reduced down to the $0..p-1$ range with a single conditional subtraction.

Montgomery multiplication is a plain product, followed by a Montgomery reduction; the Montgomery multiplication of x and y computes $xy/R \bmod p$. It is convenient to use values in *Montgomery representation*, i.e. value x is stored as $xR \bmod p$. Additions and subtractions are unchanged ($xR + yR = (x + y)R$), and the Montgomery product of xR with yR is $(xR)(yR)/R = (xy)R$, i.e. the Montgomery representation of xy. We can keep all values in that representation, converting back to integers only for encoding purposes.

[4]As will be explained in section 3.5, this can be done in a sub-quadratic number of products, but still vastly larger than n.

Traditionally, s is chosen to be close to the minimal value, since we perform computations modulo 2^s. On an ARM Cortex-M0+, using $s = 16$ for a modulus p less than 2^{16} would lead to a reduction using about 15 clock cycles. However, we can do much better, since the multiplication opcode actually works over 32-bit inputs.

Suppose that a value $x \in \mathbb{N}$ must be reduced modulo p. Suppose moreover that $0 < x < 2^{32}$. We apply Montgomery reduction with $s = 32$; but modulus p is smaller than 2^{16}. This has the following consequences:

- Value t is computed with a single `mul` opcode. It is a 32-bit value; we can split it into a low and high halves, i.e. $t = t_0 + 2^{16}t_1$.
- We then have: $t' = x + t_0p + 2^{16}t_1p$. Since t_1 and p are both lower than 2^{16}, the value t_1p will fit on 32 bits, and we can also split it into a low and high halves: $t_1p = t_2 + 2^{16}t_3$.
- This implies that $t' = x + t_0p + 2^{16}t_2 + 2^{32}t_3$. But we know that t' is a multiple of 2^{32}. Therefore, the three values x, t_0p and $2^{16}t_2$ add up to a value V whose low 32 bits are zero.

Suppose that $0 < x < 2^{32} + 2^{16} - (2^{16} - 1)p$. In that case:

$$0 < x + t_0p + 2^{16}t_2 < (2^{32} + 2^{16} - (2^{16} - 1)p) + (2^{16} - 1)p + 2^{16}(2^{16} - 1) = 2^{33}$$

Since value V is a multiple of 2^{32}, greater than 0 and lower than 2^{33}, it follows that V must be equal to 2^{32}. Therefore, the result of the reduction is necessarily equal to $t_3 + 1$. We do not have to compute other intermediate values at all! Moreover, t_3 is the high half of t_1p, with $t_1 < 2^{16}$; this implies that $t_3 < p$.

This leads to the following algorithm:

- We represent elements of $GF(p)$ with Montgomery representation in the $1..p$ range: value a is stored as $aR \bmod p$, and if $a = 0$, then we store the value as p, not 0.
- We perform additions and plain integer multiplications, resulting in a value x that must be reduced. That value fits on a 32-bit word. Since we started with non-zero integers and only performed additions and multiplications (not subtractions), we have $x > 0$. We assume that $x < 2^{32} + 2^{16} - (2^{16} - 1)p$.
- To perform a Montgomery reduction of value x, we apply the following steps:
  1. $t = xf \bmod p$ (where $f = -1/p \bmod 2^{32}$ is precomputed).
  2. $t_1 = \lfloor t/2^{16} \rfloor$ (a "right shift" operation).
  3. $t_3 = \lfloor (t_1p)/2^{16} \rfloor$ (a multiplication by the constant p, followed by a right shift).
  4. Reduced value is $t_3 + 1$, and is already in the $1..p$ range; no conditional subtraction is needed.

This implementation of Montgomery reduction uses only 5 cycles on an ARM Cortex-M0+; it requires two constants but no extra scratch register:

```
muls    r0, r7     @ r0 <- r0 * r7
lsrs    r0, #16    @ r0 <- r0 >> 16
muls    r0, r6     @ r0 <- r0 * r6
lsrs    r0, #16    @ r0 <- r0 >> 16
adds    r0, #1     @ r0 <- r0 + 1
```

In this code, registers `r6` and `r7` must have been loaded with the constants p and $-1/p \bmod 2^{32}$, respectively. Since they are not modified, they can be reused for further Montgomery reductions with no reloading cost.

For this algorithm to work properly in our case (implementation of multiplications in $GF(p^n)$), we need the pre-reduction values to fit in the acceptable range for Montgomery reduction. In the previous section, we assumed that polynomial coefficients were integers strictly less than p, but our representation now allows the value p itself (which stands for zero). Moreover, the range for fast Montgomery reduction is somewhat smaller than 2^{32}. We will therefore require the following:

$$(1 + c(n-1))p^2 + (2^{16} - 1)p < 2^{32} + 2^{16}$$

2.5 Field Selection Criteria

We need a field $GF(p^n)$ of a sufficient size to achieve a given security level. The order of a curve defined over a field of cardinal q is close to q (by Hasse's theorem, it differs from $q + 1$ by at most $2\sqrt{q}$). Since Pollard's Rho algorithm solves DLP in a group of order q in time $O(\sqrt{q})$, we need a 256-bit q in order to achieve the traditional 128-bit security level.

The choice of "128 bits" is not very rational. In general, we want a security level which is such that attacks are not practically feasible, and on top of that some "security margin", an ill-defined notion. "128" is a power of two, i.e. a nice number for somebody who thinks in binary; this makes it psychologically powerful. However, in practice, some deviations are allowed. For instance, Curve25519 uses a 255-bit field and has cofactor $h = 8$, leading to a group order close to 2^{252}. This would technically make it a 126-bit curve, two bits short of the target 128-bit level. Curve25519 is still widely accepted to offer "128-bit security" for the official reason that the level is really about equivalence to AES-128 against brute force attacks and each step in Pollard's Rho algorithm will involve substantially more work than one AES encryption; and, officiously, ditching Curve25519 because of a failure to reach a totally arbitrary level by only two bits would be too inconvenient.

For the same reasons, in our own field selection process, we will be content with any field that has cardinal close to 2^{250} or greater.

Taking all criteria listed so far, we end up with the following list:

- $p < 2^{16}$, and is prime.
- $n \geq 11$, and is prime.
- Polynomial $z^n - c$ is irreducible over $GF(p)$ for some constant c (this requires that $n|p-1$).
- $p^n \geq 2^{250}$.
- $(1 + c(n-1))p^2 + (2^{16} - 1)p < 2^{32} + 2^{16}$.

We want to minimize the degree n, since that parameter is what will drive performance. Therefore, for each potential n value, we want to find the largest possible p that satisfies the criteria above. Note that since n divides $p - 1$, the criteria imply that $n^3 < 2^{31}$, i.e. $n < 1291$.

We enumerated all primes n from 11 to 1289, and obtained the optimal values listed on table 1.

All of these solutions use polynomial $z^n - 2$ (i.e. $c = 2$ yields the best results). The list is exhaustive in the following sense: for any line in the table, corresponding to a solution (n, c, p), there is no triplet (n', c', p') that fulfills the criteria and such that $n' \leq n$ and $p'^{n'} > p^n$. With

degree n	modulus p	field size $\log_2(p^n)$
11	12739	150.007
13	11831	175.894
17	10337	226.704
19	9767	251.820
23	8971	302.014
29	7541	373.536
31	7193	397.184
37	6883	471.706
41	6397	518.370
43	6709	546.611
47	6299	593.183
53	6043	665.736
59	5783	737.359

Table 1: Optimal field degrees and base field modulus. All solutions use $M = z^n - 2$.

larger degrees, we can get to increasingly larger fields, up to $n = 761$ for a 8045.83-bit field (with $p = 1523$).

For our target goal of "128-bit security", the best choice appears to be $n = 19$ and $p = 9767$: this yields a field size (and thus, a curve order) of about 251.82 bits, very close to the 252 bits of Curve25519.

3 Efficient Field Operations

3.1 Platform Details

The ARM Cortex-M0+ is a small, low-power core that implements the ARMv6-M architecture. This follows the "Thumb" instruction set, in which almost all instructions are encoded over 16 bits; this instruction set is much more limited than what is offered by larger cores such as the ARM Cortex-M3 and M4, that use the ARMv7-M architecture. The following points are most relevant to implementation:

– There are 16 registers (`r0` to `r15`); however, the program counter (`r15`) and the stack pointer (`r13`) cannot practically be used to store any state values. Register `r9` is reserved (e.g. to support position-independent code, or thread-local storage) and is best left untouched. There are thus 13 usable registers.
– Very few operations can use the "high" registers (`r8` to `r15`): only simple copies (`mov`) and additions (`add`). Moreover, the additions are of the two-operand kind: one of the source operands is the destination; thus, that operand is consumed.
– A few operations on the "low" registers (`r0` to `r7`) can have an output distinct from the operands, e.g. additions (`adds`) and subtractions (`subs`). Multiplications (`muls`), however, are two-operand: when a product is computed, one of the source values is consumed. If both source operands must be retained for further computations, then an extra copy will be needed.

– All computation opcodes execute in 1 cycle each, with no special latency (the result of an opcode can be used as source in the next one with no penalty). However, memory accesses take 2 cycles, both for reading and for writing. The `ldm` and `stm` opcodes can respectively read and write several words in a faster way ($1+N$ cycles for N 32-bit words), furthermore incrementing accordingly the register used as pointer for that access. The destination or source registers must be among the low registers, and are used in ascending order.
– Unaligned accesses are not tolerated (they trigger CPU exceptions). 8-bit and 16-bit accesses can be performed, but cannot use addressing based on the stack pointer, contrary to 32-bit accesses.

Since arithmetic operations are fast, but memory accesses are slow, and the number of available registers is limited, most of the computation time will not be spent in actual computations, but when moving data. Optimization efforts consist mostly in finding the algorithmic data flow that will minimize the number of memory accesses, and will allow the use of the relatively faster `ldm` and `stm` opcodes with two or more words per opcode.

Performance of any routine written in assembly can be obtained in two ways:

– by measuring it on a test microcontroller that has a precise cycle counter;
– by painstakingly counting instructions manually.

We applied both methods to our code, and they match perfectly. The test system is an Atmel (now Microchip) SAM D20 Xplained Pro board, using an ATSAMD20J18 microcontroller[43]. That microcontroller can be configured to run on several clock sources; moreover, it also has some internal counters that can also be configured to use these clock sources. By using the same source (the internal 8 MHz oscillator) for both, an accurate cycle counter is obtained.

It shall be noted that while the SAM D20 board can run at up to 48 MHz, the Flash element that stores the code cannot provide 1-cycle access time at high frequencies; extra wait states are generated, that slow down execution. By running tests at 8 MHz, we can avoid any wait state. This is the usual way of providing benchmark values, and all figures in this article assume zero-wait state RAM and ROM accesses[5].

Counting instructions manually is a valid method, since the timing rules are simple (no hidden penalty or optimizations). In our code, we obtain the exact same cycle counts as what the measures show. This allows making most of the optimization work while working only with a non-accurate software simulator. In practice, development was done mostly against an embedded Linux libc (libc and compiler were obtained through the Buildroot project[16]) and executed with QEMU[52] in user-only mode (no full system emulation). This combination provides a great ease of debugging, but tests must still be ultimately performed on actual hardware, because QEMU does not trap on unaligned accesses. Tests on hardware use the cryptographic routines alone, with no libc, and only minimal boot code to configure the clocks and serial line (for measure reporting).

The exact definition of the performance of a software routine is subject to some semi-arbitrary choices: a routine must receive parameters, and returns results. The callee must preserve some register values, as per the used ABI; the caller must then save all values that are not

[5]It is also possible to copy code into RAM and then executed it from RAM, allowing high-frequency execution with no wait state; however, RAM is normally a scarce resource on microcontrollers, thus making that trick rarely worth it.

in preserved registers, but that must still be retained. Which of these saving costs should be accounted as part of the routine cost is a matter of definition.

In our implementations, we found that for most "expensive" routines (e.g. multiplication of two elements), callers usually do not have many values to retain, and it is wasteful to force the callee to save registers that the caller will not need. Indeed, saving registers `r4` to `r8`, `r10` and `r11`, as per the standard ARM ABI, and restoring them on exit, requires 22 cycles in addition to the normal function entry (saving of the link register `r12` on the stack) and exit (restoring of that value into the program counter `pc`). Our internal routines (which are not callable from C code) therefore use a modified ABI in which these registers are not saved. For such routines, the figures reported below include all opcodes that constitute the function body (including the initial "`push { lr }`" and the final "`pop { pc }`") but not the cost of the `bl` opcode that calls the routine (3 cycles) nor any value-saving costs on the caller side.

3.2 Baseline Performance

In [22], an implementation of Curve25519 point multiplication (with the Montgomery ladder) is reported, with the following performance:

– field multiplication: 1 469 cycles
– field squaring: 1 032 cycles
– curve point multiplication: 3 589 850 cycles

More recently, Haase and Labrique[31] reported slight improvements on the same operations:

– field multiplication: 1 478 cycles
– field squaring: 998 cycles
– curve point multiplication: 3 474 201 cycles

In [47], Nishinaga and Mambo claim a faster field multiplication, at only 1 350 cycles; however, the performance of the complete curve point multiplication routine is worse than above, at 4 209 843 cycles. Chances are that while they have a faster multiplication routine, they do not have a dedicated routine for squarings, making the latter substantially slower than they could. Moreover, they report substantially longer times for the ARM Cortex-M0 when compared with the M0+ (about +23% cycles), which is not consistent with known instruction timings: the M0 and M0+ should differ only in rare corner cases, such as the cost of a taken conditional branch (this costs one extra cycle on the M0). It is possible that their test platform for the ARM Cortex-M0 was used at a frequency that induced extra wait states when reading from ROM/Flash. Since their code was not published, such hypotheses cannot be verified. We consequently disregard these figures in our evaluation.

We did not find any published benchmark for curve Edwards25519 on an ARM Cortex-M0 or M0+. We can make some rough estimates, based on the figures for Curve25519. The Montgomery ladder implementation needs 5 multiplications and 4 squarings per multiplier bit (using the formulas in [38] and ignoring the multiplication by the constant `a24` which is much faster than a normal multiplication because the constant is a small integer). Since the result is obtained as a fraction, and extra inversion is needed, normally implemented with a modular exponentiation for a cost of about one squaring per modulus bit. The total cost per bit is then 5 multiplications and 5 squarings (denoted: "5M+5S"), plus some lighter operations (field additions and subtractions, conditional swap). We note that with the figures listed above, the multiplications and squarings account for about 90% of the total cost.

For Edwards25519, a typical point multiplication will need about 251 doublings, and some point additions. A microcontroller usually has limited RAM, thereby preventing use of large windows; a 4-bit window, storing 8 precomputed points in extended coordinates, will need 1 kB of temporary RAM space, and imply one point addition every 4 point doublings[6]. Using the formulas in [36], we find that:

- point doubling uses 4M+4S;
- point addition uses 8M (ignoring the multiplication by the curve constant d);
- decoding an input point (e.g. a Diffie-Hellman public key) from its encoded (compressed) format requires a combined inversion and square root, normally done with a modular exponentiation (1S per modulus bit);
- obtaining the final point in affine coordinates implies an inversion, hence an extra 1S per modulus bit.

This brings the total cost at 6M+6S per bit, i.e. about 20% more expensive than the Montgomery ladder, but more versatile: since curve Edwards25519 offers generic complete point addition formulas, it supports many cases beyond plain Diffie-Hellman, e.g. optimizing point multiplication for a conventional generator (as used in key pair generation, and signature generation), performing combined multiplications (as in EdDSA signature verification), and more generally supporting any protocol.

The Ristretto map does not substantially change these figures: decoding and encoding imply a modular exponentiation each, which replace the exponentiations involved in point decompression and in conversion back to affine coordinates.

This estimate thus rates the "multiplication by a scalar" operation on the prime-order Ristretto255 group at about 4.2 million cycles on an ARM Cortex-M0+. Keep in mind that it is only an estimate that cannot replace actual benchmarks:

- The "+20%" expression assumes that operations other than multiplications and squarings add up to about 10% of the total cost, as is the case in Curve25519 implementations.
- Any particular usage context may have more available RAM and thus allow for larger windows in the point multiplication algorithm.
- Decoding and encoding normally occur at the boundaries of the protocol, when performing I/O. If a given protocol calls for several operations (e.g. several point multiplications, and operations between the results), then the encoding and decoding could be mutualized, thereby reducing their relative cost.

3.3 Element Representation

A field element is a polynomial with coefficients in $GF(p)$, with degree at most 18. The representation in memory must thus use 19 elements, each being an integer modulo p. As explained

[6]For the purposes of this paragraph, we are considering a constant-time point multiplication, suitable for all purposes, thus without wNAF optimizations that can be applied when processing public values, e.g. for signature verification.

in section 2, we use Montgomery representation for elements in $GF(p)$, with values in the $1..p$ range.

We use 16 bits per $GF(p)$ element. To allow for accessing multiple elements at once, especially with the `ldm` and `stm` opcodes, we enforce 32-bit alignment for the whole array (i.e. array elements with even indices are guaranteed to be 32-bit aligned).

Packing two elements in a single 32-bit word makes the representation relatively compact (40 bytes per field element, including a dummy slot for alignment purposes), which saves RAM and improves efficiency of bulk transfer operations. This also allows making two (non-modular) additions in one operation: 16-bit low halves add with each other without inducing extra carries into the high halves, since $2p < 2^{16}$. On the other hand, use of the compact format makes some operations somewhat harder, notably stack-based direct access to a 16-bit value: there is no opcode that can read or write a single 16-bit element using `sp` as base address register. We still found in our implementations that the compact format yields slightly better performance, and much lower RAM usage, than the 32-bits-per-value format.

A simple way to express things is that since, on an ARM Cortex-M0+, operations are 1 cycle but memory accesses are 2 cycles each, the biggest cost is not computations but moving the data around. Anything that reduces memory exchanges tends to be good for performance.

3.4 Additions and Subtractions

When adding field elements, the individual polynomial coefficients must be added pairwise. The addition or subtraction itself can be done on the packed format, but reducing the result into the expected range ($1..p$) requires splitting words into individual elements, and performing a conditional subtraction of the modulus p. This last operation can be performed in 4 cycles on an ARM Cortex-M0+:

```
subs    r1, r7, r0    @ r1 <- r7 - r0
asrs    r1, #31       @ r1 <- r1 >> 31 (with sign extension)
ands    r1, r7        @ r1 <- r1 & r7
subs    r0, r1        @ r0 <- r0 - r1
```

This code snippet reduces value in `r0`, using `r1` as scratch register. The register `r7` must have been loaded with the constant value $p = 9767$ (`r7` is not modified and can be reused for further reductions). This code subtracts p from `r0` only if the reverse operation (subtracting `r0` from p) would have yielded a strictly negative value (sign bit set); thus, values in $1..p$ are unchanged, and values in $p + 1..2p$ are reduced by subtracting p. The result is in the expected range ($1..p$).

Subtractions can be done in a similar way, but with an extra detail to take into account: the subtraction can yield a negative value. Thus, when subtracting two 32-bit words, a carry bit from the low halves may impact the high halves; the splitting of the word into two 16-bit values will then need to compensate for this potential carry:

```
subs    r3, r5        @ r3 <- r3 - r5
sxth    r4, r3        @ sign-extend low half of r3 into r4
subs    r3, r4        @ r3 <- r3 - r4
asrs    r3, #16       @ r3 <- r3 >> 16 (with sign extension)
```

The `sxth` opcode sign-extends the low half of `r3`, thus interpreting these 16 bits as a signed representation; then, the second `subs` opcode subtracts that sign-extended value from the source: this clears the low half *and* removes the action of the carry resulting from the initial subtraction, if there was one. The arithmetic right shift then recovers the high half with signed interpretation. Once the two halves have thus been separated, the reduction is done by adding p to the value x if and only if $x + 1 < 0$ (the +1 is needed because we want $x = p$ to remain equal to p, and not replace it with 0).

In the course of computations on elliptic curve points, it often happens that several additions or subtractions must be applied successively, sometimes with small factors. For instance, given field elements u, v and w, $2(u - v - w)$ must be computed. It is then efficient to combine the operations, in order to mutualize the RAM accesses and the modular reductions. The intermediate value range is greater, though, preventing use of the 4-cycle conditional additions or subtractions explained above. Instead, we can use a derivative of Montgomery reduction; namely, to reduce value x, we apply Montgomery reduction on xR, where $R = 2^{32} \bmod p$. Moreover, the first step of Montgomery reduction is a multiplication by a constant (modulo 2^{32}); we can merge that multiplication with the multiplication by R. This yields the following code sequence:

```
muls    r0, r7     @ r0 <- r0 * r7
lsrs    r0, #16    @ r0 <- r0 >> 16
muls    r0, r6     @ r0 <- r0 * r6
lsrs    r0, #16    @ r0 <- r0 >> 16
adds    r0, #1     @ r0 <- r0 + 1
```

which is identical to the one used for Montgomery reduction (see section 2.4), but the constants are different: we set `r6` to $p = 9767$, and `r7` to $-(2^{32} \bmod p)/p \bmod 2^{32} = 439\,742$. Exhaustive experiments show that for all inputs x in the 1..509 232 range, the correct reduced value in the 1..p range is obtained.

We implemented dedicated unrolled routines for all such "linear" operations that are needed in our curve point operation routines, with individual costs ranging between 173 and 275 cycles.

3.5 Multiplication

Consider a mutiplication of field elements u and v, each consisting in 19 elements. The generic "schoolbook" multiplication routine, using the formulas in section 2.3, leads to 361 multiplications and 361 additions (since the field modulus is $z^{19} - c$ with $c = 2$, multiplications by c are equivalent to additions). Moreover, extra copies are needed, because the `muls` opcode consumes one of its operands; at the very least, all but 19 of the multiplications must involve an extra copy. The total bare minimum cost of a schoolbook multiplication is then $3 \times 361 - 19 = 1064$ cycles. This is not attainable, since this count ignores all the costs of reading data from RAM and writing it back. Also, the Montgomery reductions must be applied on top of that.

Karatsuba multiplication[37] reduces the cost of a multiplication of polynomials. Suppose that two polynomials u and v, of degree less than m, must be multiplied together. We split the polynomial u into a "low" and "high" halves:

$$u = u_l + z^{m/2} u_h$$

where u_l and u_h are polynomials of degree less than $m/2$. We similarly split v into v_l and v_h. We then have:

$$\begin{aligned} uv &= (u_l + z^{m/2}u_h)(v_l + z^{m/2}v_h) \\ &= u_lv_l + z^{m/2}(u_lv_h + u_hv_l) + z^m u_hv_h \\ &= u_lv_l + z^{m/2}((u_l + u_h)(v_l + v_h) - u_lv_l - u_hv_h) + z^m u_hv_h \end{aligned}$$

We can thus multiply two m-element polynomials by computing three products of polynomials with $m/2$ elements: u_lv_l, u_hv_h, and $(u_l + u_h)(v_l + v_h)$. Applied recursively on these sub-products, Karatsuba multiplication leads to sub-quadratic asymptotic cost $O(m^{\log_2 3}) \approx m^{1.585}$.

The description above assumes that m is even, and that the splits are even. Uneven splits are also possible, although usually less efficient. If you start with $m = 19$, and split into low halves of 10 elements (degree less than 10) and high halves of 9 elements, then the three sub-products are:

- u_lv_l: two polynomials of degree less than 10, result of degree less than 19;
- u_hv_h: two polynomials of degree less than 9, result of degree less than 17;
- $(u_l + u_h)(v_l + v_h)$: two polynomials of degree less than 10, result of degree less than 19. However, since u_h and v_h are one element shorter than u_l and v_l, the top element (degree 18) of this product is necessarily equal to the top element of u_lv_l, and the subsequent polynomial subtraction will cancel out these values. We can thus content ourselves with computing the low 18 elements of $(u_l + u_h)(v_l + v_h)$ (degrees 0 to 17) and ignore the top one.

Asymptotic behaviour is an approximation of the cost for sufficiently large inputs; but our inputs are not necessarily large enough for that approximation to be accurate. Karatsuba split reduces the number of multiplications but increases the number of additions; for small enough inputs, the extra additions overtake the cost savings from doing fewer multiplications. The threshold depends on the relative costs of additions and multiplications. In our case, multiplications are inexpensive, since Montgomery reduction is delayed. Moreover, as was pointed out previously, the costs of exchanging data between registers and RAM tend to be higher than the costs of computations. Thus, estimates based on operation counts that assume ideally free data movements may lead to the wrong conclusions, and only actual experiments will yield proper results.

In our implementation, we found that, on the ARM Cortex-M0+, one level of Karatsuba split is optimal; the operands are split into a low half of 10 elements, and a high half of 9 elements. The computation of $u_l + u_h$ can be done two elements at a time, since elements are expressed over 16 bits and packed by pairs into 32-bit words. Performing further splits yields only worse performance.

The additions and subtractions that follow the three sub-products (the "Karastuba fix-up") must operate on the 32-bit intermediate words (Montgomery reduction has not been applied yet at this point). We combine these operations with the reduction modulo $z^{19} - 2$, and with Montgomery reductions. If we define the following:

$$\begin{aligned}
\alpha &= u_l v_l \\
\beta &= u_h v_h \\
\gamma &= (u_l + u_h)(v_l + vh) \\
w &= uv \bmod z^{19} - 2
\end{aligned}$$

then the output words are computed as:

$$w_i = \alpha_i + \gamma_{i-10} - \alpha_{i-10} - \beta_{i-10} + 2(\beta_{i-1} + \gamma_{i+9} - \alpha_{i+9} - \beta_{i+9})$$

with the convention that out-of-range coefficients are zero (i.e. $\alpha_j = 0$ when $j < 0$ or $j \geq 19$). In the expression above, γ_{i-10}, α_{i-10} and β_{i-10} can be non-zero only if γ_{i+9}, α_{i+9} and β_{i+9} are zero, and vice versa. Each w_i would then entail reading five 32-bit words, but some of these read operations can be shared if we produce the output words in the order: $w_9, w_0, w_{10}, w_1, \ldots$, i.e. computation of w_j is followed by computation of $w_{j-9 \bmod 19}$. Only three memory reads are then needed for each output word on average. Some of these words can be further optimized by noticing that, for instance, $\beta_j = 0$ for $j \geq 17$, and $\gamma_{19} = \alpha_{19}$.

Putting all together, we obtain the individual costs detailed in table 2, for a total cost of 1 574 cycles. Compared to the baseline performance (section 3.2), this is about 7.1% higher than the field multiplication cost reported in [22]. Thus, while the use of the finite field $GF(9767^{19})$ does not yield a faster multiplication routine than with the field $GF(2^{255} - 19)$, it is still competitive, the difference in performance being slight.

Operation	**Cost (cycles)**
function prologue	5
$u_l + u_h$ and $v_l + v_h$	63
$u_l v_l$	410
$u_h v_h$	345
$(u_l + u_h)(v_l + v_h)$	405
Karatsuba fix-up and Montgomery reduction	337
function exit	5
Total	1 574

Table 2: Field multiplication cost.

Squarings. Squarings can be optimized by noticing that:

$$\begin{aligned}
u^2 &= (u_l + z^{10} u_h)^2 \\
&= u_l^2 + 2z^{10} u_l u_h + z^{20} u_h^2
\end{aligned}$$

reducing the 19-element squaring to a 10-element squaring, a 9-element squaring, and a 10×9-element multiplication. However, in our experiments, we found that squarings of polynomials of 10 elements or fewer are almost twice faster than generic multiplications, mostly because all operands can then fit into registers and avoid almost all read and write accesses to RAM. Therefore, better performance is achieved by using Karatsuba multiplication:

$$\begin{aligned} u^2 &= (u_l + z^{10} u_h)^2 \\ &= u_l^2 + z^{10}((u_l + u_h)^2 - u_l^2 - u_h^2) + z^{20} u_h^2 \end{aligned}$$

We obtain the cycle counts detailed in table 3, for a total of 994 cycles. This is very slightly faster than the best reported baseline squaring in $GF(2^{255} - 19)$ (998 cycles, in [31]). A noteworthy point is that squaring costs are only about 63.2% of multiplication costs; in elliptic curve computations, this makes it worthwhile to replace 2 multiplications with 3 squarings. This impacts analysis of elliptic curve formulas; e.g. [10] ranks formulas under the assumption that a squaring cost is 80% of a multiplication cost.

Operation	**Cost (cycles)**
function prologue	5
$u_l + u_h$	30
u_l^2	219
u_h^2	182
$(u_l + u_h)^2$	216
Karatsuba fix-up and Montgomery reduction	337
function exit	5
Total	994

Table 3: Field squaring cost.

Another important remark is that additions and subtractions are relatively expensive: an addition in the field is 173 cycles, i.e. about 11% of the cost of a multiplication, and 17.4% of the cost of a squaring. This highlights that counting multiplications and squarings is not sufficient to get an accurate estimate of a complete operation on elliptic curve points.

3.6 Inversion

Modular inversion can be computed in several ways. The main recommended method is to use Fermat's little theorem; namely, the inverse of u in a finite field of cardinal q is u^{q-2}. Nominally, $u = 0$ does not have an inverse, but the exponentiation yields a result of 0 if the operand is 0, and that turns out to be convenient in some edge cases. For an m-bit exponent $q-2$, $m-1$ squarings will be needed, along with some extra multiplications; since the exponent is known in advance and not secret, the number of extra multiplications can be quite small by using an optimized addition chain on the exponent. In [22], inversion in $GF(2^{255} - 19)$ is performed with 254 squarings, and 11 extra multiplications.

On $GF(9767^{19})$, we can use a much faster method, which computes an inversion in a cost equivalent to only 6 multiplications. The method was initially described by Itoh and Tsujii in the context of binary fields[35], then adapted to other finite field extensions (e.g. see [32]). It uses the following remark:

$$p^n - 1 = (p-1)(1 + p + p^2 + p^3 + \cdots + p^{n-1})$$

Let $r = 1 + p + p^2 + \cdots + p^{n-1}$, and let $x \neq 0$ a field element to invert. By Fermat's little theorem, we have:

$$\begin{aligned}(x^r)^{p-1} &= x^{p^n-1}\\ &= 1\end{aligned}$$

Therefore, x^r is a root of the polynomial $X^{p-1}-1$ over the finite field $GF(p^n)$. That polynomial can have at most $p-1$ roots, and all elements of $GF(p)$ are roots, therefore the roots of $X^{p-1}-1$ over $GF(p^n)$ are exactly the elements of the sub-field $GF(p)$. It follows that $x^r \in GF(p)$.

We can thus compute the inverse of x as:

$$x^{-1} = \frac{x^{r-1}}{x^r}$$

The division by x^r is easy since it requires only an inversion in $GF(p)$, followed by a multiplication of x^{r-1} by that inverse, which is also in $GF(p)$.

The values x^{r-1} and then x^r can be efficiently computed through application of the Frobenius automorphism. In a finite field $GF(p^n)$, we define the j-th Frobenius operator (for $0 \leq j < n$) as:

$$\begin{aligned}\Phi_j : GF(p^n) &\longrightarrow GF(p^n)\\ x &\longmapsto x^{p^j}\end{aligned}$$

Since $GF(p^n)$ has characteristic p, these operators are automorphisms: $\Phi_j(xy) = \Phi_j(x)\Phi_j(y)$ and $\Phi_j(x+y) = \Phi_j(x) + \Phi_j(y)$ for all x and y in $GF(p^n)$. Moreover, when the finite field is defined as the quotient ring $GF(p)[z]/(z^n - c)$, then we have:

$$\Phi_j(z^i) = c^{ij(p-1)/n} z^i$$

This means that computing $\Phi_j(x)$ over a field element x is a simple matter of term-by-term multiplication with the values $c^{ij(p-1)/n}$ in $GF(p)$; these values can be precomputed. In our implementation, application of a Frobenius operator costs 211 cycles, i.e. slightly more than an addition, but much less so than a multiplication. Such optimization of the Frobenius operator is the main reason why we wanted the field extension modulus with the form $z^n - c$ for some constant c in $GF(p)$, and not, for instance, $z^n - z - 1$ (which would also have supported base primes p in an adequate range).

The inversion algorithm (see algorithm 1) leverages these facts to compute x^{r-1} in only 5 multiplications, and 6 applications of a Frobenius operator.

Algorithm 1 Fast inversion in $GF(9767^{19})$

```
Require: $x \in GF(p^n), x \neq 0, p = 9767, n = 19$
Ensure: $1/x$
 1: $t_1 \leftarrow x \cdot \Phi_1(x)$                    ▷ $t_1 = x^{1+p}$
 2: $t_1 \leftarrow t_1 \cdot \Phi_2(t_1)$                ▷ $t_1 = x^{1+p+p^2+p^3}$
 3: $t_1 \leftarrow t_1 \cdot \Phi_4(t_1)$                ▷ $t_1 = x^{1+p+p^2+\cdots+p^7}$
 4: $t_1 \leftarrow x \cdot \Phi_1(t_1)$                  ▷ $t_1 = x^{1+p+p^2+\cdots+p^8}$
 5: $t_1 \leftarrow t_1 \cdot \Phi_9(t_1)$                ▷ $t_1 = x^{1+p+p^2+\cdots+p^{17}}$
 6: $t_1 \leftarrow \Phi_1(t_1)$                          ▷ $t_1 = x^{p+p^2+p^3+\cdots+p^{18}} = x^{r-1}$
 7: $t_2 \leftarrow xt_1$                                 ▷ $t_2 = x^r \in GF(p)$
 8: $t_2 \leftarrow t_2^{p-2}$                            ▷ Inversion in $GF(p)$
 9: $t_1 \leftarrow t_1 t_2$                              ▷ $t_1 = x^{r-1}/x^r = 1/x$
10: return $t_1$
```

In algorithm 1, the following remarks apply:

- Four of the field multiplications are between u and $\Phi_j(u)$ for some element u. The Frobenius operator and the multiplication can be combined to avoid some write operations that are then read again immediately. In our implementation, this saves about 36 cycles each time, i.e. 144 cycles over the complete inversion.
- The multiplication in step 7 is fast because the result is known to be an element of $GF(p)$; thus, only one polynomial coefficient needs to be computed.
- Inversion of t_2 in $GF(p)$ (step 8) can be done with Fermat's little theorem, i.e. by raising the input to the power $p - 2$. With $p = 9767$, this is a matter of only 17 Montgomery multiplications. In our implementation, this step costs only 107 cycles.
- Multiplication by t_2 in step 9 is a simple coefficient-wise multiplication, thus much more efficient than a normal multiplication.

Our implementation achieves the costs listed in table 4, for a total of 9 508 cycles. This is 6.04 times the cost of a single multiplication. Since the algorithm itself involves 5 generic multiplications, this means that the 6 Frobenius operators, the specialized multiplication that yields x^r, the inversion in $GF(p)$, and the final multiplication by x^{-r}, collectively cost about the same as a 6th generic multiplication.

Operation	**Cost (cycles)**
function prologue	7
combined Frobenius and multiplication (step 1)	1 762
combined Frobenius and multiplication (step 2)	1 763
combined Frobenius and multiplication (step 3)	1 763
Frobenius and multiplication (step 4)	1 798
combined Frobenius and multiplication (step 5)	1 763
Frobenius (step 6)	217
x^r	130
x^{-r}	110
$x^{r-1}x^{-r}$	190
function exit	5
Total	9 508

Table 4: Field inversion cost.

3.7 Square Root

Using techniques similar to the fast inversion algorithm exposed in section 3.6, we can obtain a fast square root extraction algorithm, and an even faster quadratic residue test.

Our field has cardinal $q = 9767^{19}$. Since $q = 3 \mod 4$, square roots of element $x \in GF(q)$ are obtained as:

$$\sqrt{x} = \pm x^{(q+1)/4}$$

If x is not a quadratic residue, then $-x$ is a quadratic residue, and this modular exponentiation returns a square root of $-x$.

The exponent can be written as:

$$\frac{p^{19}+1}{4} = (2e-r)\frac{p+1}{4}$$

where:

$$\begin{aligned} d &= & 1 + p^2 + p^4 + \cdots + p^{14} + p^{16} \\ e &= 1 + dp^2 &= 1 + p^2 + p^4 + \cdots + p^{14} + p^{16} + p^{18} \\ f &= pd &= p + p^3 + p^5 + \cdots + p^{15} + p^{17} \\ r &= e + f &= 1 + p + p^2 + p^3 + \cdots + p^{17} + p^{18} \end{aligned}$$

This allows performing the square root computations as:

$$\sqrt{x} = \pm \left(\frac{(x^e)^2}{x^r}\right)^{(p+1)/4}$$

As in the case of the inversion algorithm, x^r is an element of the sub-field $GF(p)$, hence inexpensive to invert; and a^e can be computed with a few multiplications and applications of Frobenius operators. The final exponentiation, with exponent $(p+1)/4$, can be done with 10 squarings and 4 multiplications.

The algorithm can also return whether the operand is a quadratic residue. Indeed, $x \neq 0$ is a quadratic residue if and only if $x^{(q-1)/2} = 1$. This exponent can be written as:

$$\frac{p^{19}-1}{2} = r\frac{p-1}{2}$$

Therefore, $x^{(q-1)/2} = (x^r)^{(p-1)/2}$. In other words, x is a quadratic residue in $GF(q)$ if and only if x^r is a quadratic residue in $GF(p)$ (this also applies for $x = 0$). Since we compute x^r as part of the algorithm, we can also check whether it is a quadratic residue for fewer than 100 extra cycles. Moreover, if we are *only* interested in whether x is a quadratic residue, we can stop there and avoid the final raise to power $(p+1)/4$.

The exact process is described in algorithm 2. The operation costs are detailed in table 5, for a total cost of 26 962 cycles. If the square root is not requested, only the quadratic residue status, then that status is obtained in 9 341 cycles.

Algorithm 2 Fast square root in $GF(9767^{19})$

```
Require: $x \in GF(p^n), p = 9767, n = 19$
Ensure: QR status of $x$; $\sqrt{x}$ if QR, $\sqrt{-x}$ otherwise
 1: $t_1 \leftarrow x \cdot \Phi_2(x)$                ▷ $t_1 = x^{1+p^2}$
 2: $t_1 \leftarrow t_1 \cdot \Phi_4(t_1)$            ▷ $t_1 = x^{1+p^2+p^4+p^6}$
 3: $t_1 \leftarrow t_1 \cdot \Phi_8(t_1)$            ▷ $t_1 = x^{1+p^2+p^4+\cdots+p^{14}}$
 4: $t_1 \leftarrow x \cdot \Phi_2(t_1)$              ▷ $t_1 = x^{1+p^2+p^4+\cdots+p^{16}}$
 5: $t_2 \leftarrow \Phi_1(t_1)$                      ▷ $t_2 = x^{p+p^3+p^5+\cdots+p^{17}} = x^f$
 6: $t_1 \leftarrow x \cdot \Phi_1(t_2)$              ▷ $t_1 = x^{1+p^2+p^4+\cdots+p^{18}} = x^e$
 7: $t_3 \leftarrow t_1 t_2$                          ▷ $t_3 = x^r \in GF(p)$
 8: $t_4 \leftarrow t_3^{(p-1)/2}$                    ▷ $t_4 = 0, 1$ or $-1$
 9: if only QR status is requested then return $(t_4 \neq -1)$
10: $t_3 \leftarrow t_3^{p-2}$                        ▷ Inversion in $GF(p)$
11: $t_1 \leftarrow t_1^2$                            ▷ $t_1 = x^{2e}$
12: $t_1 \leftarrow t_1 t_3$                          ▷ $t_1 = x^{2e}/x^r$
13: $t_1 \leftarrow t_1^{(p+1)/4}$                    ▷ $t_1 = \sqrt{x}$ or $\sqrt{-x}$
14: return $(t_4 \neq -1)$ and $t_1$
```

Operation	**Cost (cycles)**
function prologue	7
combined Frobenius and multiplication (step 1)	1 762
combined Frobenius and multiplication (step 2)	1 763
combined Frobenius and multiplication (step 3)	1 763
Frobenius and multiplication (step 4)	1 798
Frobenius (step 5)	217
Frobenius and multiplication (step 6)	1 798
x^r	129
QR status	100
exit if only QR status requested	4
x^{-r}	113
$(x^e)^2$	999
$(x^e)^2x^{-r}$	195
raise to power $(p+1)/4$	16 311
function exit	7
Total	26 962
(if only QR status requested)	9 341

Table 5: Field square root cost.

3.8 Cube Root

Since $q = 9767^{19} = 2 \bmod 3$, every element in $GF(q)$ has a unique cube root, which is obtained with a modular exponentiation:

$$\sqrt[3]{x} = x^{(2q-1)/3}$$

As in the case of square roots, this exponentiation can be greatly optimized with the Frobenius operator. The exponent is rewritten as:

$$\frac{2p^{19}-1}{3} = e\frac{2p-1}{3} + f\frac{p-2}{3}$$

with e and f defined as in section 3.7. We then compute the cube root as:

$$\begin{aligned}\sqrt[3]{x} &= (x^e)^{(2p-1)/3}(x^f)^{(p-2)/3}\\ &= x^e(x^{2e+f})^{(p-2)/3}\end{aligned}$$

This yields algorithm 3 with costs detailed in table 6 and a total cost of 31 163 cycles.

Algorithm 3 Fast cube root in $GF(9767^{19})$

```
Require: $x \in GF(p^n), p = 9767, n = 19$
Ensure: $\sqrt[3]{x}$
 1: $t_1 \leftarrow x \cdot \Phi_2(x)$            ▷ $t_1 = x^{1+p^2}$
 2: $t_1 \leftarrow t_1 \cdot \Phi_4(t_1)$        ▷ $t_1 = x^{1+p^2+p^4+p^6}$
 3: $t_1 \leftarrow t_1 \cdot \Phi_8(t_1)$        ▷ $t_1 = x^{1+p^2+p^4+\cdots+p^{14}}$
 4: $t_1 \leftarrow x \cdot \Phi_2(t_1)$          ▷ $t_1 = x^{1+p^2+p^4+\cdots+p^{16}}$
 5: $t_2 \leftarrow \Phi_1(t_1)$                  ▷ $t_2 = x^{p+p^3+p^5+\cdots+p^{17}} = x^f$
 6: $t_1 \leftarrow x \cdot \Phi_1(t_2)$          ▷ $t_1 = x^{1+p^2+p^4+\cdots+p^{18}} = x^e$
 7: $t_2 \leftarrow t_1^2 t_2$                    ▷ $t_2 = x^{2e+f}$
 8: $t_2 \leftarrow t_2^{(p-2)/3}$                ▷ $t_2 = (x^{2e+f})^{(p-2)/3}$
 9: $t_1 \leftarrow t_1 t_2$                      ▷ $t_1 = \sqrt[3]{x}$
10: return $t_1$
```

Operation	**Cost (cycles)**
function prologue	7
combined Frobenius and multiplication (step 1)	1 762
combined Frobenius and multiplication (step 2)	1 763
combined Frobenius and multiplication (step 3)	1 763
Frobenius and multiplication (step 4)	1 798
Frobenius (step 5)	217
Frobenius and multiplication (step 6)	1 798
x^{2e+f}	2 579
raise to power $(p-2)/3$	17 890
final multiplication (step 9)	1 581
function exit	5
Total	31 163

Table 6: Field cube root cost.

Other root computations can benefit from such optimizations. In general, every exponentiation in $GF(p^n)$ can be optimized by representing the exponent in base p, and splitting the exponentiation into n exponentiations with short exponents (less than p) over the $\Phi_j(x)$ and whose results are multiplied together; with classic square-and-multiply algorithms, this allows mutualizing all the squaring operations. When the exponentiation is for a k-th root with k small, the various small exponents tend to have common parts, leading to further optimizations, as we just saw for inversions, square roots and cube roots.

4 Curve9767

Using the finite field $GF(9767^{19})$ we defined in section 2 and for which efficient operations were described in section 3, we now proceed to define an elliptic curve over this field. Following the current fashion of naming curves with the concatenation of the term “Curve” and a sequence of seemingly random digits, we call our new curve “Curve9767”, the digits being, of course, the decimal representation of the base modulus p.

4.1 Choosing The Curve

The choice of the exact curve to use was the result of a process taking into account known criteria for security, and with as few arbitrary choices as possible. We describe the steps here.

In all of the following, we keep using the notations used previously: the base field modulus is $p = 9767$, and the field cardinal is $q = p^{19}$.

Curve Equation. Since we want a prime-order curve, we cannot use Montgomery or Edwards curves (which always have an even order). Instead, we concentrate on the short Weierstraß equation $y^2 = x^3 + ax + b$. The choice is then really about the two constants a and b, which are elements of $GF(q)$.

We cannot choose $a = 0$: since the field cardinal $q = 2 \bmod 3$, this would lead to a curve with exactly $q + 1$ elements, which is an even number. Moreover, it would be supersingular, implying in particular a quadratic embedding degree and consequently severe weakness against pairing-based attacks such as MOV[42] and FR[25,26].

Similarly, we cannot choose $b = 0$: since $q = 3 \bmod 4$, this would again yield a supersingular curve with $q + 1$ elements.

There are known curve point addition formulas that can leverage the specific choice $a = -3$ for slightly better performance in some cases, e.g. for point doubling with Jacobian coordinates[10][7]. Moreover, for any non-zero u, the mapping $(x, y) \mapsto (xu^{-2}, yu^{-3})$ is an isomorphism between curve $y^2 = x^3 + ax + b$ and curve $y^2 = x^3 + au^4x + bu^6$, which implies that we can choose the constant a mostly arbitrarily: about half of all possible curves can be transformed efficiently through such an isomorphism into a curve whose equation has $a = -3$.

In all generality, if a is fixed, then b should be chosen pseudorandomly, if we want to claim that a large fraction of possible curves could have been chosen. However, there is no known weakness induced by any specific choice of b; we can set it to a low Hamming weight value $b_i z^i$ for some integer $i \in [1..18]$ (as explained above, we need $b \notin GF(p)$, hence $i \neq 0$). This should not be a controversial optimization, since it is commonly done for other curves. For instance, a similar optimization was done in the choice of Curve25519: the curve equation is $By^2 = x^3 + Ax^2 + x$, but the constant B is fixed to 1 (which does not unduly shrink the space of possible curves, thanks to isomorphisms) *and* the constant A is chosen to be a small integer so as to promote performance[8].

[7]In our specific case, even when using Jacobian coordinates, these formulas don’t actually lead to better performance on the ARM Cortex-M0+ because field squarings are substantially more efficient than multiplications, making 1M+8S generic doubling formulas a better choice than specialized 3M+5S. However, this might not be true of other target architectures, and we’d like to keep our implementation options as open as possible.

Twist Security. The concept of "twist security" is introduced in [8], in the context of a specialized point multiplication routine for Curve25519, based on a Montgomery ladder, in which only the x coordinates of points are used. For any x such that $Q = (x, y)$ is a curve point, there is exactly one other point $-Q = (x, -y)$ with the same x coordinate; therefore, the x coordinate is sufficient to represent Q, up to the sign of y. Since usual Diffie-Hellman uses only the x coordinate of the result as shared secret, this is sufficient for some cryptographic applications: public points Q and $-Q$ lead to the same output, and thus the sign of y is not used. This allows efficient implementations, because the y coordinate needs not be transmitted, and the ladder computes the x coordinates only. However, a consequence is that since y is not available, there is no easy (cheap) way to validate incoming points, i.e. that a received x really corresponds to a point on the curve.

Analysis shows that any field element is either the x coordinate of a point on the intended curve E, or the x coordinate of a point on the *quadratic twist*, i.e. another curve E' such that E and E' become the same curve in a quadratic field extension $GF(q^2)$. If the order of E' admits small prime factors, then this would allow invalid curve attacks[12], in which the attacker sends invalid points (i.e. points on the twisted curve) with a small order, allowing an easier break of Diffie-Hellman in that case, leading to partial information on the victim's private key. *Twist security* is about choosing the curve E such that both E and E' have prime order (or close to prime order, with very small cofactors): in a nutshell, it does not matter whether computations are done on E or E', as long as both are "safe".

In our case, twist security is not really required; even when using an x-only implementation, input point validation is inexpensive thanks to the efficient quadratic residue test (see section 3.7): for a given x, we can compute $x^3 + ax + b$; x is valid if and only if that quantity is a quadratic residue. However, ensuring twist security is "free": it is only an extra parameter to the curve selection, thus with no runtime cost, as long as requiring twist security does not prevent us from using a particularly efficient curve constant. We will then try to obtain twist security in our curve choice.

When using the short Weierstraß equation $y^2 = x^3 + ax + b$ on a field $GF(q)$ where $q = 3 \bmod 4$, the quadratic twist has equation $-y^2 = x^3 + ax + b$, which is isomorphic to the curve of equation $y^2 = x^3 + ax - b$ (with the isomorphism $(x, y) \mapsto (-x, y)$). Therefore, if a is fixed to a given value (e.g. -3) and we look for b with a minimal Hamming weight, then the twisted curve will use the same a, and a constant $-b$ with the same minimal Hamming weight. When enumerating possible curves, we will naturally cover the twisted curves at the same time. In that sense, when E and E' both have prime order, we are free to choose either as our curve. In that case, we will use the one with the smallest order: in a Diffie-Hellman context, when receiving point Q, the defender computes sQ for a secret non-zero scalar s modulo the curve order E; choosing E to be smaller than its twist E' ensures that the computation sQ does not yield the point at infinity $\mathbb{O}$, either as a result or an intermediate value, even if Q is really a point on E' instead of E.

Curve Parameters. Given the criteria explained above, we enumerated all curves $y^2 = x^3 + ax + b$ over $GF(9767^{19})$, with $a = -3$ (i.e. $p - 3$, an element of $GF(p)$) and $b = b_i z^i$ for $b_i \in GF(p)$, $b_i \neq 0$, and $1 \leq i \leq 18$, looking for curves with prime order.

Application of the Frobenius operator Φ_j maps curve $y^2 = x^3 - 3x + b_i z^i$ to curve $y^2 = x^3 - 3x + 2^{ij(p-1)/19} b_i z^i$, which is also in the set of evaluated curves. Therefore, each considered curve is really a set of 19 isomorphic curves. We can thus restrict ourselves to only one curve in each set, which speeds up the search by a factor 19. We (arbitrarily) choose the representative as the one with the smallest value b_i when expressed as an integer in the $0..p-1$ range.

Up to Frobenius isomorphism, there are $18(p-1)/19 = 9252$ curves to consider. Using PARI/GP[48], along with the optional `seadata-small` package (to speed up point counting), we found that exactly 23 of them have a prime order[8]. As luck would have it, exactly two of them are twists of each other; as per our criteria, we then choose as Curve9767 the one with the smallest order, corresponding to $b = 2048z^9$ (the set of 19 isomorphic curves for the twisted curve then corresponds to $b = 359z^9$).

A conventional generator should be selected for the curve. Since the curve has prime order, any point (other than the point at infinity) generates the whole curve; moreover, the ability to solve discrete logarithm relatively to a specific generator is equivalent to the ability to solve it relatively to any other generator. We can thus choose any generator we want. Usually, the choice won't have any impact on performance, but one can imagine some edge cases where coordinates with low Hamming weight are preferable. The value with the lowest Hamming weight is zero. There is no point on Curve9767 with coordinate $y = 0$ (since this would be a point of order 2), but there are two points with $x = 0$: these are the two points $(0, \pm\sqrt{b})$. Both have a y coordinate with Hamming weight 1. As in the case of b within its Frobenius isomorphism class, we arbitrarily choose the point whose y coordinate is the lowest when expressed as an integer in the $0..p-1$ range.

The resulting Curve9767 parameters are summarized in table 7.

Field	$GF(9767)[z]/(z^{19}-2)$
Field order	9767^{19}
Equation	$y^2 = x^3 - 3x + 2048z^9$
Order	6389436622109970582043832278503799542449455630003248488928817956373993578097
Generator	$G = (0, 32z^{14})$

Table 7: Curve9767 definition parameters.

Embedding Degree. For an elliptic curve defined over a finite field $GF(q)$, and a prime r that divides the curve order, such that r is not the field characteristic and r^2 does not divide the curve order, the curve contains r points of order r. The *embedding degree* is the minimum integer $k > 0$ such that the same curve over the extension field $GF(q^k)$ contains r^2 points of order r. It has been shown by Balasubramanian and Koblitz[7] that k is the smallest positive integer such that r divides $q^k - 1$; in other words, k is the multiplicative order of q modulo r. k is always a divisor of $r-1$.

Pairing-based attacks like MOV[42] and FR[25,26] rely on transferring the elliptic curve discrete logarithm problem into the the discrete logarithm problem in the multiplicative sub-

[8]The search script is provided with the Curve9767 source code. Enumeration took 1 hour and 40 minutes on a 3.1 GHz x86 server, using a single core.

group of $GF(q^k)$. Therefore, these attacks are possible only if k is small enough that the best known sub-exponential algorithms for the latter problem are faster than generic attacks on the elliptic curve. For an ordinary curve which has not been chosen to be especially pairing-friendly, we expect k to be a very large integer. Various bodies have emitted recommendations that insist on k being larger than a given threshold; for instance, ANSI X9.62:2005[1] requires $k \geq 100$, while SafeCurves[11] goes much beyond (in their words, the "overkill approach") and requires $k \geq (r-1)/100$.

In the case of Curve9767, the curve order itself is prime, hence the only possible value for r is the curve order. The embedding degree then happens to be $k = r - 1 \approx 2^{251.82}$, i.e. the maximum possible value. This makes Curve9767 as immune to MOV and FR attacks as it is possible for an elliptic curve to be.

Complex Multiplication Discriminant. For an elliptic curve defined over a finite field $GF(q)$ and with order r, the *trace* is the value $t = q + 1 - r$. By Hasse's theorem, $|t| \leq 2\sqrt{q}$; thus, $t^2 - 4q$ is a negative integer. Write that quantity as DV^2, where D is a square-free negative integer, and V is a positive integer. The value D is the *complex multiplication field discriminant*[9].

When $|D|$ is very small, it may accept low-degree (i.e. efficiently computable) curve endomorphisms that can be used to speed up point multiplications[27,40]. This has been used in curves specially designed to that effect, e.g. secp256k1[18] and Four$\mathbb{Q}$[19]. However, when a curve has not been specifically chosen for a small discriminant, it is expected that the value of $|D|$ is large. Curves with a small discriminant are certainly not broken, but an *unexpected* small discriminant would be indicative of some unaccounted for underlying structure, which would be suspicious.

In Curve9767, the $t^2 - 4q$ quantity is already square-free (i.e. $V = 1$) leading to a very large discriminant $D \approx -2^{253.82}$, as is expected of most ordinary curves.

4.2 Point Representation

In our implementation, a point Q on the curve is the combination of three elements (x, y, N):

- x and y are the affine coordinates of Q; they are elements of $GF(q)$, in the representation used in section 3.3 (40 bytes each, including the dummy slot for 32-bit alignment).
- N is the "neutral flag": an integer with value 1 (if $Q = \mathbb{O}$) or 0 (if $Q \neq \mathbb{O}$).

We encode N over a 32-bit field, again for alignment purposes. When $N = 1$, the contents of x and y are unspecified; since we use the exact-width type `uint16_t`, access does not lead to "undefined behavior" in the C standard sense, even if not explicitly set in the code[10], but these values are ultimately ignored since the point at infinity does not have coordinates. A consequence is that $\mathbb{O}$ has multiple representations, while all other points have a unique in-memory representation.

[9] Strictly speaking, when D is a multiple of 4, the actual discriminant is defined to be $4D$. But this cannot happen for an odd-order curve over an odd-characteristic field, because then t must be odd, implying that $D = 1 \bmod 4$.

[10] Exact-width types are not allowed to have any padding bits or trap representations, therefore they always have a readable value, even if it is not specified.

4.3 Point Addition

We implement point addition by applying the affine equations, as shown in section 1. We want a *complete*, *constant-time* routine, i.e. one that works on all combinations of inputs, with an execution time and memory access pattern independent of input values (see section 5.1 for details). This is achieved with the process described in algorithm 4.

Algorithm 4 Point addition for Curve9767

```
Require: $Q_1 = (x_1, y_1, N_1)$ and $Q_2 = (x_2, y_2, N_2)$ points on Curve9767
Ensure: $Q_3 = Q_1 + Q_2$
 1: $e_x \leftarrow \mathsf{EQ}(x_1, x_2)$                 ▷ $e_x = 1$ if $x_1 = x_2$, 0 otherwise
 2: $e_y \leftarrow \mathsf{EQ}(y_1, y_2)$                 ▷ $e_y = 1$ if $y_1 = y_2$, 0 otherwise
 3: $t_1 \leftarrow x_2 - x_1$
 4: $t_3 \leftarrow 2y_1$
 5: CONDCOPY($\&t_1, t_3, e_x$)                ▷ $t_1$ is the denominator of $\lambda$
 6: $t_2 \leftarrow y_2 - y_1$
 7: $t_3 \leftarrow 3x_1^2 - 3$
 8: CONDCOPY($\&t_2, t_3, e_x$)                ▷ $t_2$ is the numerator of $\lambda$
 9: $t_1 \leftarrow t_2/t_1$                        ▷ $t_1 = \lambda$
10: $x_3 \leftarrow \lambda^2 - x_1 - x_2$
11: $y_3 \leftarrow \lambda(x_1 - x_3) - y_1$
12: CONDCOPY($\&x_3, x_2, N_1$)
13: CONDCOPY($\&x_3, x_1, N_2$)
14: CONDCOPY($\&y_3, y_2, N_1$)
15: CONDCOPY($\&y_3, y_1, N_2$)
16: $N_3 \leftarrow (N_1N_2) + (1 - N_1)(1 - N_2)e_x(1 - e_y)$
17: return $Q_3 = (x_3, y_3, N_3)$
```

In algorithm 4, two helper functions are used:

- EQ(u, v) returns 1 if $u = v$, 0 if $u \neq v$.
- CONDCOPY($\&u, v, F$) overwrites u with v if $F = 1$, but leaves u unmodified if $F = 0$.

Both functions are implemented with constant-time code: for instance, in CONDCOPY, all words of u and v are read, and all words of u written to, regardless of whether F is 0 or 1.

This description is formal; in the actual implementation, some operations are combined to lower memory traffic. Typically, the conditional copies in step 12 and 13 are done in a single loop; similarly for steps 14 and 15.

We can see that the algorithm implements all edge cases properly:

- If $Q_1 = \mathbb{O}$ and $Q_2 = \mathbb{O}$, then $N_1 = 1$ and $N_2 = 1$, leading to $N_3 = 1$, i.e. $Q_1 + Q_2 = \mathbb{O}$.
- If $Q_1 = \mathbb{O}$ and $Q_2 \neq \mathbb{O}$, then (x_3, y_3) is set to (x_2, y_2) in steps 12 and 14, but not modified in steps 13 and 15. Also, N_3 is set to 0. The result is thus point Q_2, as expected.
- If $Q_1 \neq \mathbb{O}$ and $Q_2 = \mathbb{O}$, then (x_3, y_3) is set to (x_1, y_1) in steps 13 and 15, but not modified in steps 12 and 14. Also, N_3 is set to 0. The result is thus point Q_1, as expected.
- Otherwise, $Q_1 \neq \mathbb{O}$ and $Q_2 \neq \mathbb{O}$, i.e. $N_1 = 0$ and $N_2 = 0$. The following sub-cases may happen:

- If $Q_1 = Q_2$ then $e_x = 1$ and $e_y = 1$. The numerator and denominator of λ are computed to be $3x_1^2 + a$ and $2y_1$, respectively, as befits a point doubling operation. The result neutral flag N_3 is properly set to 0: there is no point of order 2 on the curve, thus the result cannot be the point at infinity.
- If $Q_1 = -Q_2$, then $e_x = 1$ and $e_y = 0$; this leads to $N_3 = 1$ in the final step, i.e. the point at infinity is properly returned.
- Otherwise, $Q_1 \neq Q_2$ and $Q_1 \neq -Q_2$; this implies that $x_1 \neq x_2$, hence $e_x = 0$. The numerator and denominator of λ are set to $y_2 - y_1$ and $x_2 - x_1$, respectively, in application of the generic point addition formula. N_3 is set to 0: the result cannot be the point at infinity.

Our optimized implementation for ARM Cortex-M0+ computes a point addition in a total of 16 331 cycles, i.e. about 10.4 times the cost a field multiplication. This cost is detailed in table 8. We may notice that since the process involves one inversion (9 508 cycles), two multiplications (1 574 cycles each) and two squarings (994 cycles each), the overhead for all "linear" operations (subtractions, conditional copies...) is 1 687 cycles, i.e. about 10.3% of the total. This function furthermore follows the C ABI: it saves and restores registers properly and thus can be called from external application code.

Operation	**Cost (cycles)**
function prologue	20
e_x and e_y	116
denominator of λ	285
x_1^2	999
numerator of λ	300
division t_2/t_1 (inversion + multiplication)	11 093
x_3	1 253
y_3	1 958
conditional copy of (x_1, y_1) or (x_2, y_2)	269
N_3	22
function exit	16
Total	16 331

Table 8: Point addition cost.

Completeness. In the context of elliptic curves, *complete formulas* are formulas that work in all cases, including edge cases such as adding a point to itself, or to the point at infinity. In practice, applications that use elliptic curves in various cryptographic protocols need *complete routines* that can add points without edge cases (that could lead to incorrect results) and without timing variations when an internally handled edge-case is encountered. Complete formulas are a means through which a complete routine can be obtained. Here, we implemented a complete routine which is *not* based on complete formulas, but is still efficient. Notably, making the point addition function complete does not require difficult trade-offs with regard to performance. An incomplete addition routine that cannot handle doublings (when adding a point to itself) would save about 1 500 cycles (it would avoid computing $3x_1^2 + a$ and $2y_1$, as well as some CONDCOPY calls), less than 10% of the point addition cost.

We argue that obtention of a complete routine whose efficiency is close to that of any potential incomplete routine is sufficient for security in all generality. A developer who is intent on reimplementing curve operations in an incomplete or non-timing-resistant way will be able to do so, but will also succeed in ruining the best complete formulas. In that sense, complete formulas are a nice but not strictly required mechanism for achieving complete constant-time routines, and do not in themselves provide absolute protection against implementation mishaps.

4.4 Repeated Doublings

In order to speed up point multiplication, we implemented an optimized function for multiple point doublings. That function takes as input parameters a point $Q_1 = (x_1, y_1)$ and an integer $k \geq 0$; it returns the point $2^k Q_1$. The parameter k is not secret.

The two special cases $k = 0$ and $k = 1$ are first handled, by copying the input into the output (66 cycles) or tail-calling the generic point addition routine (16 337 cycles), respectively. When $k \geq 2$, the doublings are performed by using Jacobian coordinates. This is only an *internal* use: the result is converted to affine coordinates after the k-th doubling. It should be noted that point doublings are "safe" in Curve9767, because its order is odd: if the input point Q_1 is the point at infinity, then $2^k Q_1$ is the point at infinity, but if Q_1 is not the point at infinity, then none of the successive $2^j Q_1$ values is the point at infinity. Therefore, the only edge case to cover is $Q_1 = \mathbb{O}$, and it is handled in a very simple way: the "neutral flag" N_1 is simply copied to the result.

For point (x, y), the Jacobian coordinates $(X{:}Y{:}Z)$ are such that $x = X/Z^2$ and $y = Y/Z^3$. Since the input point is in affine coordinates, we can optimize the first two doublings. Following an idea of [39], we can implement the first doubling in only four squarings, and some linear operations; if $Q_2 = (X_2{:}Y_2{:}Z_2) = 2Q_1$, then the following holds:

$$\begin{aligned}
X_2 &= x_1^4 - 2ax_1^2 + a^2 - 8bx_1 \\
Y_2 &= y_1^4 + 18by_1^2 + 3ax_1^2 - 6a^2x_1^2 - 24abx_1 - 27b^2 - a^3 \\
Z_2 &= 2y_1
\end{aligned}$$

Thanks to our choice of curve constants $a = -3$ and $b = 2048z^9$ with very low Hamming weight, multiplications by a and by b are inexpensive.

Remaining doublings use the 1M+8S formulas from [10], which are valid for all short Weierstraß curves in Jacobian coordinates (we do not use the 3M+5S or 4M+4S formulas that leverage $a = -3$, since that does not yield any performance benefit on the ARM Cortex-M0+, thanks to the high speed of squarings relatively to multiplications). We recall these formulas here, for the doubling of point $(X{:}Y{:}Z)$ into point $(X'{:}Y'{:}Z')$:

$$\begin{aligned}
T_1 &= X^2 \\
T_2 &= Y^2 \\
T_3 &= T_2^2 \\
T_4 &= Z^2 \\
T_5 &= 2((X + T_2)^2 - T_1 - T_3) \\
T_6 &= 3T_1 + aT_4^2 \\
X' &= T_6^2 - 2T_5 \\
Y' &= T_6(T_5 - X') - 8T_3 \\
Z' &= (Y + Z)^2 - T_2 - T_4
\end{aligned}$$

Note that the first doubling set $Z = 2y_1$; therefore, the computations of $T_4 = Z^2$ and T_4^2 (as part of the computation of T_6) really compute $4y_1^2$ and $16y_1^4$. Since y_1^2 and y_1^4 were already computed as part of the first doubling, we can save two squarings in the second doubling.

The total function cost for $k \geq 2$ is $7\,584 + 11\,392k$; this includes the cost of converting back the result to affine coordinates. Table 9 details the cost items. For $k = 4$, this means a cost of 53 152 cycles, i.e. about 81.3% of the 65 348 cycles that would have been used to call the generic point addition routine four times (this optimization saves 756 152 cycles from a complete point multiplication by a scalar, which is not negligible).

Operation	**Cost (cycles)**
function prologue	28
first doubling	5 675
second doubling	9 394
subsequent doublings ($k - 2$ times)	11 392
conversion to affine coordinates	15 255
function exit	16
Total	$7\,584 + 11\,392k$

Table 9: Point multiplication by 2^k cost.

4.5 Point Multiplication By A Scalar

Generic Point Multiplication. Generic point multiplication receives a point Q and multiplies it by the scalar s. In our implementation, scalars are integers modulo r (where r is the curve prime order); scalars are *decoded* from sequences of bytes using unsigned little-endian convention. Two scalar decoding methods are provided, one that ensures that the value is in the 0 to $r - 1$ range, the other reducing the source value modulo r. In either case, the scalar value for point multiplication is less than r. Operations on scalars are not critical for performance; therefore, we use a simple, generic and compact routine in C. For multiplications and modular reductions, Montgomery multiplication is used. The total compiled code footprint for all scalar operations is 1 064 bytes (when compiled with GCC 7.3.0 for an ARM Cortex-M0+ target with "`-Os`" optimization flag). Like the rest of our code, the scalar implementation is fully constant-time.

To compute sQ, we used a simple four-bit window. For a window of w bits, the process is the following:

1. Let $k = \lfloor(\log r)/w\rfloor$. We will use the binary representation of the scalar by chunks of w bits, and there will be exactly k chunks (the last chunk might be incomplete).
2. Compute and store in a dedicated RAM space (the *window*, usually on the stack) the points iQ for $i = 1$ to 2^{w-1}. This can use the generic point addition routine, calling it $2^{w-1} - 1$ times.
3. Add $2^{w-1}\sum_{i=0}^{k-1} 2^{wi}$ to s (modulo r).
4. Start with $Q' = \mathbb{O}$.
5. For $i = 0$ to $k - 1$:
   (a) Extract the i-th w-bit chunk from the scalar: $j = \lfloor s/2^{wi}\rfloor \bmod 2^w$.
   (b) Look up point $T = |j - 2^{w-1}|Q$ from the window; if $j = 2^{w-1}$, the lookup returns $T = \mathbb{O}$.
   (c) If $j < 2^w$, set $T \longleftarrow -T$ (i.e. negating the y coordinate of T).
   (d) Set $Q' \longleftarrow 2^w Q' + T$. The multiplication by 2^w uses the optimized repeated doublings procedure described in section 4.4, and the addition with T uses the generic point addition routine. When $i = 0$, the doublings can be skipped, since it is statically known that $Q' = \mathbb{O}$ at this point.
6. Return Q'.

Note that for a window of w bits, we only store 2^{w-1} points. We then use a lookup index skewed by 2^{w-1}, and obtain the actual point to add with a conditional negation. For instance, if using a 4-bit window $w = 4$, we store points $Q, 2Q, 3Q, \ldots 8Q$; the lookup index j is between 0 and 8 (inclusive); and the final point T will range from $-8Q$ to $+7Q$, instead of $0Q$ to $15Q$. The addition of the specific constant to the scalar (in step 3) counterbalances this skew.

Within the window, we only store the x and y coordinates of the points iQ. The "neutral flag" of the looked-up point T is adjusted afterwards (it is set to 1 if $Q = \mathbb{O}$ or if the lookup index $j = 0$).

The window size is a trade-off. With a larger window, fewer iterations are needed, thus reducing the number of window lookups and point additions; large windows also make repeated doublings slightly more efficient (since our repeated doublings procedure has a 11 392-cycle cost for each doubling *plus* a fixed 7 584-cycle overhead). On the other hand, larger windows increase the lookup time (we use a constant-time lookup with a cost proportional to the number of stored points) and, more importantly, increase temporary RAM usage. Systems that use the ARM Cortex-M0+ usually have severe RAM constraints. Each point in the window uses 80 bytes (40 bytes per coordinate, including the two extra bytes for 32-bit alignment); a 4-bit window thus implies 640 bytes of (temporary) storage. Depending on the usage context, a larger window may or may not be tolerable.

We might note that typical point multiplication routines on Edwards25519 store windows with points in projective, inverted or extended coordinates, using three or four field elements per point, hence at least 96 or 128 bytes. Since our Curve9767 points are in affine coordinates, they use less RAM, and may thus allow larger windows for a given RAM budget. Of course, the Montgomery ladder (on Curve25519) does not use a window and is even more compact in RAM.

Our generic point multiplication routine has been measured to work in 4 493 999 cycles. During development, we also wrote another version that was keeping the intermediate point Q' in Jacobian coordinates; doublings we thus more efficient (we avoided the 7 584-cycle overhead for each multiplication by 16), through point additions (adding an affine point from the window to the current point in Jacobian coordinates) were slightly more expensive (about one thousand extra cycles per addition). This yielded a point multiplication routine in about 4.07 million cycles, i.e. 9.4% fewer than our current implementation. We did not keep that variant for the following reasons:

- The addition routine in Jacobian coordinates required a nonnegligible amount of extra code, mostly for all the "linear" operations.
- Handling of edge cases (when the current point Q' is the point at infinity) required extra flags and more conditional copies.
- The method could not scale to combined multiplications, as described below (computing $s_1Q + s_2G$). When multiplying a single point Q by a scalar s which is such that $0 \leq s < r$, it can be shown that none of the point additions in the main loop is in fact a doubling (adding a point to itself). However, this is not true when doing a combined multiplication: intermediate values may lead to a hidden doubling, and the pure Jacobian point addition routine does not handle that edge case correctly.

We thus prefer sticking to affine coordinates, even though they lead to a slightly slower point multiplication routine. Compared to the baseline (Curve25519), Curve9767 then provides a point multiplication routine which is about 1.29 times slower. Depending on context, this may or may not be tolerable. However, this slowdown factor is less than the "1.5 to 2.9 factor" from the analysis in [54]; in that sense, this result shows that the design strategy of Curve9767 is worth some attention.

Combined Point Multiplications. Some cryptographic protocols require computing $s_1Q_1 + s_2Q_2$ for two points Q_1 and Q_2. In particular, verification of ECDSA or Schnorr signatures uses that operation, with Q_1 being the public key, and Q_2 the conventional curve generator point. Instead of doing both point multiplications separately and adding the results together, we can mutualize the doublings, using "Shamir's trick" (originally described in the context of ElGamal signature verification and credited by ElGamal to a private communication from Shamir[23]). Namely:

- Two windows are computed, for points Q_1 and Q_2.
- A single accumulator point Q' is kept.
- At each loop iteration, two lookups are performed, using indices from each scalar, and resulting in points T_1 and T_2. The doubling-and-add computation is then $Q' \leftarrow 2^wQ' + T_1 + T_2$.

It is also possible to compute a *combined window* with all points $i_1Q_1 + i_2Q_2$ for $0 \leq i_1 < 2^{w-1}$ and $0 \leq i_2 < 2^w$, but for a given RAM budget, this is usually not worth the effort, the RAM being better spent on two individual windows with twice as many bits.

This process naturally extends to more than two points. Each extra point requires its own window, but all doublings are mutualized.

Our implementation of the combined point multiplication routine, with source point Q_2 being the conventional curve generator (this is the operation needed for Schnorr signature verification), computes $s_1 Q_1 + s_2 G$ in 5 590 769 cycles. Since the curve generator is fixed, its window can be precomputed and stored in ROM (Flash), so that RAM usage is no more than with the generic point multiplication routine.

Generator Multiplication. Multiplying a point which is known in advance (normally the conventional curve generator G) is the operation used in key pair generation, and also for each signature generation. Several optimizations are possible:

- Since the point is known at compile-time, its window can be precomputed and stored in ROM/Flash. This saves the dynamic computation time.
- ROM size constaints are usually less strict in embedded systems than RAM constraints, because ROM is cheaper[11]. This allows the use of larger windows.
- A process similar to combined point multiplications can be used: the multiplier s can be split into several chunks. For instance, if s is split into two halves s_1 and s_2, with $s = s_1 + 2^{128} s_2$, and s_1 and s_2 being each less than 2^{128}, then $sG = s_1 G + s_2 2^{128} G$, which can leverage the mutualization of doublings, provided that $2^{128} G$ is precomputed and stored (preferably along with its precomputed window). Since the sub-scalars s_1 and s_2 are half-width, the number of iterations is halved.

In our implementation, to compute sG, we split s into four 64-bit chunks, and we store precomputed 4-bit windows for G, $2^{64} G$, $2^{128} G$ and $2^{192} G$. Only 16 internal iterations are used, each involving a multiplication by 16, four lookups, and four point additions (for the first iteration, the accumulator point Q' is the point at infinity, and we can avoid the multiplication by 16 and one of the point additions). In total we compute a multiplication of the generator G by a scalar in 1 877 847 cycles.

We used a four-way scalar split and 4-bit windows for implementation convenience; however, both the number of scalar chunks and the size of the windows can be adjusted, for various trade-offs between implementation speed and ROM usage. In our case, the four precomputed windows add up to 2560 bytes of ROM.

4.6 Point Compression

The in-RAM format for a point uses 84 bytes (including the "neutral flag", and alignment padding). However, curve points can be encoded in a much more compact format, over only 32 bytes (specifically, 255 bits, the last bit is not used).

Field Element Encoding. For a field element $u = \sum_{i=0}^{19} u_i z_i$, there are 19 polynomial coefficients to encode. Each coefficient is an integer in the 0 to $p - 1 = 9766$ range. The in-RAM values use Montgomery representation and furthermore encode 0 as the integer p;

[11]As a rule of thumb, each SRAM bit needs 6 transistors, but a ROM bit only requires 1 transistor-equivalent space.

however, we convert back the coefficients to non-Montgomery representation and into the $0..p-1$ range so that encoding formats do not force a specific implementation strategy.

For a compact encoding, we encode the first 18 coefficients by groups of three. Each group $(u_{3i}, u_{3i+1}, u_{3i+2})$ uses exactly 40 bits (5 bytes):

- Each coefficient is split into an 11-bit low part $l_j = u_j \bmod 2^{11}$), and a high part $h_j = \lfloor u_j/2^{11} \rfloor$. This is implemented with simple masks and shifts. Since $u_j < 9767$, the high part h_j is lower than 5.
- The low and high parts of the three coefficients are assembled into value:

$$v_i = l_{3i} + 2^{11} l_{3i+1} + 2^{22} l_{3i+2} + 2^{33}(h_{3i} + 5h_{3i+1} + 25h_{3i+2})$$

  Note that since $l_j < 2^{11}$ and $h_j < 5$, it is guaranteed that $v_i < 2^{40}$.
- The value v_i is encoded over 5 bytes in unsigned little-endian convention.

The 18 coefficients u_0 to u_{17} yield 6 groups of three, hence a total of 30 bytes. The last coefficient (u_{18}) is then encoded in unsigned little-endian convention over the last two bytes. Since $u_{18} < 9767$, it uses at most 14 bits, and the two most significant bits of the last byte are free.

Decoding must recover the l_j and h_j elements from the received bytes. Obtaining the high parts (h_j) entails divisions by 5; for constant-time implementations, one can use the fact that $\lfloor x/5 \rfloor = \lfloor (103x)/2^9 \rfloor$ for all integers x in at least the 0..127 range[12].

The decoding routine should detect and report invalid encodings, i.e. encodings that lead to coefficients not in the 0..9766 range.

Point Encoding. Since each point (x, y) on the curve fulfills the curve equation $y^2 = x^3 + ax + b$, knowledge of x is sufficient to recompute y^2, from which y can be obtained with a square root extraction. The fast square root extraction algorithm described in section 3.7 makes this process efficient. Since y^2 admits two square roots, an extra bit is needed to designate a specific y value.

We define the *sign* of a field element u as follows:

- If $u = 0$ then its sign is zero.
- Otherwise, let i be the largest index such that $u_i \neq 0$. We define that the sign of u is one if $u_i > p/2$, zero otherwise. (This uses the normalized u_i value in the $0..p-1$ range, not in Montgomery representation).

It is easily seen that if $u \neq 0$, then u and $-u$ have opposite signs (i.e. exactly one of u and $-u$ has sign 1, the other having sign 0).

The encoding of a Curve9767 point (x, y) into 32 bytes then consists of the encoding of x, with the sign of y inserted into the next-to-top bit of the last byte (i.e. the bit of numerical value 64 within the 32nd byte of the encoding); the top bit of the last byte (of numerical value 128 within that byte) is cleared. The decoding process then entails decoding the value of x (masking out the two top bits of the last byte), computing y^2, extracting the square root

[12]Division opcodes are not constant-time on many CPU. Optimizing compilers can implement divisions by constants through multiplications and shifts, using the techniques from [30], but they may prefer to use division opcodes, especially when optimizing for code size instead of raw speed. Expliciting the use of multiplications and shifts avoids such issues.

y, and finally replacing y with $-y$ if the sign bit of the recomputed y does not match the next-to-top bit in the last byte of the encoding[13].

The decoding function reports an error in the following cases:

- The top bit of the last byte is not zero.
- The first 254 bits do not encode a valid field element (at least one coefficient is out-of-range).
- A value x is validly encoded, but $x^3 + ax + b$ is not a quadratic residue, and there is thus no curve point with that value as x coordinate.

There is no formally defined encoding for the point at infinity. However, if requested to encode $\mathbb{O}$, the encoding function produces an all-ones pattern (all bytes of value `0xFF`, except the last byte which is set to `0x7F`). This is not a valid encoding (it would yield out-of-range coefficients). Similarly, when the decoding function detects an invalid encoding, it reliably sets the destination point to the point-at-infinity in addition to reporting the error. In that sense, it is possible to use that invalid all-ones pattern as the encoding of the point at infinity. It is up to the calling application to decide whether neutral points should be allowed or not; most protocols don't tolerate neutral points.

In our implementation, point encoding takes 1 527 cycles, while decoding uses 32 228 cycles (field multiplication, squaring, and square roots use our assembly routines, but the rest of the code is written in C).

4.7 Hash-To-Curve

The *hash-to-curve* functionality maps arbitrary input bit sequences to curve points in a way which is indifferentiable from a random oracle. Some cryptographic protocols can tolerate weaker properties, but in general we want the resulting point to be such that, informally, all curve points could have been obtained with quasi-uniform probability, and no information is leaked about the discrete logarithm of the result relatively to a given base point. We moreover require *constant-time* hashing, i.e. not leaking any information on the source value through timing-based side channels; this, in particular, prevents us from using rejection sampling methods in which pseudorandom x values are generated from the input with a strong pseudorandom generator until one is found such that $x^3 + ax + b$ is a quadratic residue[14].

Since we work with field $GF(q)$ with $q = 2 \bmod 3$, we can use a process based on Icart's map[33] and formally proven[15] to be indifferentiable from a random oracle, when the underlying hash function is itself modeled as a random oracle. It consists of three elements:

- MapToField: an input sequence of bytes is mapped to a field element u by interpreting the sequence as an integer U (using unsigned little-endian convention) then converting it to base p: $U = \sum_i U_i p^i$. The first (least significant) 19 digits U_0 to U_{18} are then used as the polynomial coefficients of u.

[13]Since Curve9767's order is odd, there is no point with coordinate $y = 0$; therefore, there exists no value x such that $x^3 + ax + b = 0$, avoiding the edge case of $y = 0$ but a requested sign bit of 1.

[14]With our fast square root and quadratic residue tests, such a process would hash an arbitrary input in an *average* cost under 60 000 cycles, but occasionally much higher.

– IcartMap: from a given field element u, a curve point is obtained. If $u = 0$ then the point at infinity $\mathbb{O}$ is produced; otherwise, the point (x, y) is produced with the following formulas:

$$\begin{aligned} v &= \frac{3a - u^4}{6u} \\ x &= \left(v^2 - b - \frac{u^6}{27}\right)^{1/3} + \frac{u^2}{3} \\ y &= ux + v \end{aligned}$$

– HashToCurve: a message m is used as input to an extendable-output function (XOF) such as SHAKE[34]; a 96-byte output is obtained, split into two 48-byte halves d_1 and d_2. We then define:

$$\mathsf{HashToCurve}(m) = \mathsf{IcartMap}(\mathsf{MapToField}(d_1)) + \mathsf{IcartMap}(\mathsf{MapToField}(d_2))$$

Using 48 bytes (i.e. 384 bits) for each half implies that MapToField's output is quasi-uniform with bias lower than 2^{-132} (since the field cardinal is lower than 2^{252}), i.e. appropriate for the "128-bit" security level that Curve9767 provides. The conversion to base 9767 is done with repeated divisions by 9767, themselves implemented with multiplications and shifts only, using the techniques described in [30].

Our Curve9767 implementation comes with a perfunctory SHAKE implementation; our hash-to-curve function takes as input the SHAKE context, pre-loaded with the input message m and ready to produce bytes. It is up to the caller to organize the injection of m into SHAKE, preferably with a domain separation header to avoid unwanted interactions with other protocols and operations that use SHAKE on the same input m. The hash-to-curve operation is computed in 195 211 cycles. Out of these, each MapToField uses 20 082 cycles; this function was written in C and compiled with code size optimizations ("`-Os`") and could probably be made to run faster with handmade assembly optimizations. The SHAKE invocation itself, with our C implementation also compiled with code size optimizations, amounts to about 34 000 cycles. Icart's map is evaluated in 50 976 cycles.

Maps other than Icart's could have been used. In particular, the Shallue-Woestijne-Ulas map[55,57], as simplified in [15] for curves defined over fields $GF(q)$ with $q = 3 \bmod 4$ (which is the case of Curve9767), can be implemented with a few operations, mostly one inversion, one quadratic residue test, one square root extraction, and a few multiplications and squarings. In our case, it is slightly more expensive than Icart's map, though the difference is slight.

More discussion on the practical implementation of hash-to-curve procedures can be found in [24].

4.8 Higher-Level Protocols

In order to have benchmarks for Curve9767 when applied in realistic protocols, we defined and implemented Diffie-Hellman key exchange and Schnorr signatures. When a XOF is required, we use SHAKE256[15]. All uses of SHAKE include "domain separation strings", i.e. conventional headers for the XOF input that avoid the same output occurring in different contexts. All our domain separation strings start with "`curve9767-`" and end with a colon character "`:`". When a string is specified below, e.g. as "`curve9767-keygen:`", the ASCII encoding of the string as a sequence of bytes, without the double-quote characters and without any terminating NUL byte, is used.

Key Pair Generation. From a given random seed (presumably obtained from a cryptographically strong RNG, with entropy at least 128 bits), we generate a private key s (an integer modulo the curve order r), an additional secret t used for signature generation (32 bytes), and the public key $Q = sG$ with G being the curve conventional generator. The process is the following:

- The concatenation of the domain separation string "`curve9767-keygen:`" and the seed is injected into a new SHAKE256 instance.
- SHAKE256 is used to produce 96 bytes of output. The first 64 bytes are interpreted as an integer with unsigned little-endian convention; that integer is reduced modulo the curve order r, yielding the secret scalar s. The remaining 32 bytes from the SHAKE256 output are the additional secret t. It may theoretically happen that we obtain $s = 0$; in that case, we set $s = 1$. This is only a theoretical concern, since there is no known seed value that results in such an outcome, and while it makes the value 1 conceptually twice as probable as any other, the bias is negligible.
- The public key $Q = sG$ is computed.

The same process is used for Diffie-Hellman key pairs and signature key pairs. In the former case, the t value may be skipped, since it is used only for signatures (but SHAKE256 produces output by chunks of 136 bytes, so there is no saving in performance obtained by not generating t). Note that in all generality, key exchange key pairs and signature key pairs should be separate; they have different lifecycles and it is never recommended to use the same private key in two different cryptographic algorithms. Nothing prevents us, though, from using the same *process* (hence the same implementation code) for generating both kinds of key pairs, provided that they work on different seeds.

The cost of key pair generation is almost entirely that of the computation of the public key $Q = sG$, at least on an ARM Cortex-M0+, where SHAKE is inexpensive compared to curve point multiplications. The public key computation uses the "multiplication of the generator by a scalar".

Diffie-Hellman Key Exchange. Each party in a Diffie-Hellman key exchange executes the following steps:

- A new key pair (s, Q) is generated, if using ephemeral Diffie-Hellman. For static Diffie-Hellman, the key pair is recovered from storage, and used for multiple Diffie-Hellman instances.

[15]Nominally, we only target the 128-bit security level, and SHAKE128 would be sufficient. However, using SHAKE256 makes no difference in performance in our case, and the "256" figure has a greater marketing power.

- The public key Q is encoded and sent to the peer.
- A public key (32 bytes) is received from the peer, and decoded as a point Q'.
- The point sQ' is computed, then its x coordinate is encoded as a sequence of 32 bytes; this is the *pre-master secret*. We use only the x coordinate, without the sign bit from the y coordinate, in order to follow traditional Diffie-Hellman on elliptic curves[2], and to make the process compatible with x-only ladder implementations of point multiplication.
- The concatenation of the domain separation string "`curve9767-ecdh:`" and the pre-master secret are input into a new SHAKE256 instance, whose output is the shared secret between the two peers participating to the exchange. Since SHAKE256 is a XOF, the two parties can obtain unbounded amounts of shared key material, e.g. to power both symmetric encryption and MAC for two unidirectional data tunnels.

The decoding of the received point Q' may fail. In usual contexts, it is acceptable to simply abort the protocol in such a case. In order to support unusual usage contexts in which the key exchange is used as part of a larger protocol in which points are not observable and attackers should not be able to observe which key exchanges succeed or fail, an alternate pre-master secret is used when Q' fails to decode properly. The alternate pre-master secret is the 32-byte SHAKE256 output computed over an input consisting of the concatenation of the domain separation string "`curve9767-ecdh-failed:`", the encoding (over 32 bytes, unsigned little-endian convention) of the secret scalar s, and the 32 bytes received as purported encoded Q' point. Our implementation always computes both the normal pre-master secret and the alternate one, and selects the latter in case the decoding failed (and the point multiplication was performed over invalid data). This process ensures that, from the outside, the ECDH process always results in some unpredictable key that is still deterministically obtained for a given 32-byte sequence purportedly encoding Q'.

Almost all of the computation time in Diffie-Hellman is spent in the two point multiplications, for computing $Q = sG$ (as part of key pair generation) and sQ', the latter requiring the generic point multiplication routine.

Schnorr Signatures. We define Schnorr signatures in a process similar to EdDSA[9,36]. The message to sign or to verify is provided as a hash value h, obtained from some collision-resistant hash function[16]. Whenever h is used, we really use the concatenation of an identifier for the hash function, and the value h itself. The identifier is the ASCII encoding of the decimal representation of the standard OID for the hash function, followed by a colon character. For instance, if using SHA3-256, the identifier string is: "`2.16.840.1.101.3.4.2.8:`"

To generate a signature, using the public/private key pair (s, t, Q):

1. Concatenate the domain separation string "`curve9767-sign-k:`", the additional secret t, the hash function identifier string, and the hash value h. This is the input for a new SHAKE256 instance. 64 bytes of output are obtained from SHAKE256, and interpreted as an integer (unsigned little-endian encoding) which is then reduced modulo r

[16]This "hash function" may be the identity function, as for the "Pure EdDSA" mode. This avoids relying on the collision resistance of a hash function; however, such hash-less processing requires verifiers to already know the public key and the signature value when the beginning of the data is being known, which prevents streamed processing and is a problem for some tasks on memory-constrained devices, e.g. X.509 certificate chain validation as part of TLS. We therefore recommend always using a proper hash function first, e.g. SHA3.

(the curve order), yielding the scalar k. For completeness, if $k = 0$, it is replaced with 1 (this happens only with negligible probability).

2. Compute the curve point $C = kG$ and encode it as value c (32 bytes). This is the full point encoding, including the sign bit for the y coordinate.
3. Concatenate the domain separation string "`curve9767-sign-e:`", the value c, the encoding of the public key Q, the hash function identifier and the hash value h. This is the input for a new SHAKE256 instance. Generate 64 bytes of output, interpret them as an integer (unsigned little-endian encoding), and reduce that integer modulo the curve order r. This yields the scalar e.
4. Compute $d = k + es \bmod r$.
5. The signature is the concatenation of c (32 bytes) and d (encoded over exactly 32 bytes with unsigned little-endian convention).

This signature generation process is deterministic: for the same input (hashed) message h and private key, the same signature is obtained. It is not strictly required that this process is used to generate k; any mechanism that selects k uniformly and unpredictably in the $1..r-1$ range can be used. However, the deterministic process described above has the advantage of not requiring a strong random source, and its determinism makes it testable against known-answer vectors. Conversely, determinism may increase vulnerability to some classes of physical attacks, especially fault attacks. See section 5.1 for more details.

To verify a signature, the following process is used:

1. Split the signature (64 bytes) into its two halves c and d (32 bytes each).
2. Decode d as an integer (unsigned little-endian convention). If $d \geq r$, the signature is invalid.
3. Recompute the challenge value e as in step 3 of the signature generation process.
4. Compute the point $C = dG - eQ$, using the alleged signer's public key e.
5. Encode point C. The signature is valid if that encoding matches c (the first half of the signature), invalid otherwise.

The signature generation cost consists almost entirely of the computation of kG (multiplication of the curve generator by a scalar). The signature verification cost is dominated by $dG - eQ$, which is a combined point multiplication process.

5 Implementation Issues and Benchmarks

5.1 Side Channel Attacks

Constant-Time Code. Among side channel attacks, a well-known category consists of timing attacks, or, more generally, side channel attacks that exploits measures based on time (but not necessarily of the execution time of the target system itself). These attacks include all sorts of cache attacks, that try to obtain information on secret values based on the memory access pattern of the attacked system and its effect on various cache memories. Timing-based side channels are "special" because they can often be exercised remotely: either the timing differences can be measured over a high-speed network, or the attacker has control of a generic system close to the target (e.g. another VM co-hosted on the same hardware) and can use the abilities of such systems at measuring very short amounts of time. All other side channel attacks require some special measuring hardware in the physical vicinity of the target system, and can often be ruled out based on usage context.

Constant-time coding is a relatively confusing terminology that designates code which does not necessarily execute in a fixed amount of time, but such that any timing-related measures yield no information whatsoever on secret values. Constant-time code makes no memory access at secret-dependent values, performs no conditional jump based on secret boolean values, and avoids use of any hardware opcode with a varying execution time (a category which includes some multiplication opcodes on some platforms[50]).

It shall be noted that the two Curve25519 implementations we use as baseline are *not* truly constant-time: when performing the conditional swap in each iteration of the Montgomery ladder, they only exchange the pointers to the relevant field elements, not the values. Subsequent memory accesses then happen at addresses that depend on the conditional boolean, which is a private key bit. It is asserted in [31] that:

Note that for internal memories of Cortex M4 and M0 access timing is deterministic.

This is not true in all generality. The ARM Cortex-M0 and M4 cores do not include any cache by themselves and issue read and write requests with timings that do not depend on the target address. However, the *system* in which these cores are integrated may induce timing differences. These cores are not full CPUs in their own right; they are hardware designs that a CPU designer uses in a larger chip, along with extra pieces such as a memory controller. *In general*, RAM is provided with a SRAM block that offers deterministic and uniform access timing, but this is not always the case. Memory controller designs with cache capabilities commercially exist[17]. Other potential sources of address-dependent timing differences include automatic arbitration of concurrent memory accesses (when other cores, or peripherals, access RAM concurrently to the CPU) or refresh cycles for DRAM. Accesses to ROM/Flash may also have caches and other wait states (for instance, STM32F407 boards with an ARM Cortex-M4 implement both data and instruction caches for all accesses to Flash).

Therefore, while on a specific microcontroller, a not-truly constant-time implementation may get away with making memory accesses at secret-dependent addresses, this is a relatively fragile assertion, and a generic software implementation should use true constant-time code by default.

Our implementation is truly constant-time. In particular, all lookups in the windows for point multiplication use a constant-time implementation that reads all values from the window, and combines them with bitwise operations to extract the right one. For a 4-bit window (containing eight pre-computed points), the lookup process executed in 777 cycles. Since the generic point multiplication entails 63 such lookups, true constant-time discipline implies an overhead of 48 591 cycles, which is not large in practice (about 1.1% of the total time) but should conceptually be taken into account when comparing Curve9767 with the baseline Curve25519 implementations that are not truly constant-time in that sense (i.e. a fair comparison would first deduce these 48 591 cycles from our code's performance, or add a similar amount to the baseline implementation performance).

We also applied constant-time discipline more generally; all our functions are constant-time, including code paths that are usually safe. For instance, public keys or signature values are normally exchanged publicly; we still decode them in constant-time and do not even leak (through timings) whether the decoding was successful or not. While this maniacal insistence on full constant-time is useless in most contexts, we feel that it may matter in some unusual cases and thus should be the default for any generic-purpose implementation. Moreover, the runtime overhead is usually negligible or very small; following constant-time discipline mostly implies forfeiting conditional jumps ("`if`" clauses) and propagating an error status through the call tree.

Power Analysis. Side channel attacks can rarely be addressed in all generality, since they rely on specific hardware properties and usage context. We can have a generic "constant-time" implementation because most CPUs have similar timing-related properties (namely, caches whose behaviour depends on the accessed address, not on the stored value), and also because timing attacks that can be enacted remotely use measures that are amenable to an abstract description, owing to the fact that the measuring apparatus is itself a generic computer with merely a cycle counter. This simple context does not extend to other side channel attacks, e.g. power analysis attacks. Consequently, it is not usually feasible to make a software implementation that can be said to be immune to side channel attacks *in abstracto*.

However, it is known that some "generic" mitigations can help with a nonnegligible proportion of particular situations. In the case of elliptic curve implementations, projective coordinates can be *randomized*: given a point $(X{:}Y{:}Z)$, one can always generate a random nonzero field element μ and multiply it with all three coordinates, since $(\mu X{:}\mu Y{:}\mu Z)$ represents the exact same curve point[17]. If randomization is applied regularly throughout a curve operation (e.g. after each doubling in a double-and-add point multiplication algorithm), then the extra randomness is expected to somehow blur the information leaking through side channels and make analysis more expensive, especially in terms of number of required traces. The effectiveness of this countermeasure varies widely depending on context, but in most it will help defenders.

Using affine coordinates prevents applying that kind of randomization. In order to use randomization, one has to use redundant coordinate systems. On short Weierstraß curves, Jacobian coordinates provide in general the best performance for point multiplication, but not for combined multiplications or other operations since they are incomplete formulas and don't handle edge cases properly. If a Curve9767 implementation must be made resistant to side channel attacks such as power analysis, in the sense explained above, then we recommend using projective coordinates with the complete formulas from [53]. With these formulas, on a short Weierstraß curve with $a = -3$, doublings cost 8M+3S, along with a number of "linear" operations (including some multiplications by b, which are fast on Curve9767 since b has low Hamming weight). The number of such extra operations is known to be relatively high (21 additions and 2 multiplications by b) and we estimate that they collectively add an overhead of 20%; this would put the cost of a point doubling at close to 19 000 cycles on an ARM Cortex-M0+. Similarly, for generic point additions (12M and 31 "linear" operations), we estimate the cost around 23 000 cycles. Each randomization is an extra 5 000 cycles, assuming a very fast random generator[18]. In total, assuming a 4-bit window, extra randomization for each

[17]Other coordinate systems, e.g. Jacobian coordinates, can also be randomized in a similar way.

[18]The random μ can be slightly biased, allowing generation of the 19 coefficients by generating random 31-bit values and applying Montgomery reduction on each of them; if the 31-bit values are obtained from dedicated hardware or a very fast process, the reduction themselves won't cost more than a basic field addition.

doubling, and a 1 100-cycle window lookup operation[19], we can estimate a total side-channel-resistant point multiplication cost of about 7.5 million cycles. This is only an estimate; we did not implement it.

Fault Attacks. Fault attacks are a kind of side channel attack where the attacker forces the computation to derail in some way, through a well-targeted physical intervention, such as sending a short-time voltage glitch (abnormally high or low voltage for a small amount of time) or chip alteration (cutting or bridging specific chip wires with lasers under microscopic inspection).

Deterministic algorithms are known to be more vulnerable to fault attacks, since they allow attackers to repeat experiments with the same intermediate values in all computation steps; this has been applied in particular to signature algorithms[3,49]. The Schnorr signature scheme which we described in section 4.8 is deterministic: for a given signature key (s, t, Q) and hashed message h, the per-signature secret scalar k is generated with a deterministic pseudo-random process. Having a fully specified deterministic process has quality assurance benefits: the signature scheme can be tested against known-answer test vectors[51]. However, randomization can be applied nonetheless: the signature verification process does not (and cannot) rely on such deterministic generation. In order to capture both the immunity to random generators of poor quality[20] and to still randomize data to make fault attacks harder, the generation of k can be amended by appending a newly-generated random value to the concatenation of the domain separation string, the additional secret t, the hash function identifier string and the hash value h; this extra input to SHAKE256 will make the process non-repeatable, thus increasing the difficulty of fault exploitation by attackers.

5.2 Benchmarks

As described in section 3.1, all measures were performed on a SAM D20 Xplained Pro board. The microcontroller is configured to use the internal 8 MHz oscillator, with no wait state for reading Flash memory. The internal oscillator is also configured to power a 32-bit counter. No interrupts are used; the counter value is read directly[21]. The benchmark code runs a target function in a loop; the loop is invoked three times, with 1, 10 and 100 iterations. The cycle count is measured three times. The same loop is used for all functions, to avoid variability (7 pointer-sized arguments are passed to the target function; as part of the ABI, the callee can ignore extra arguments, since the caller is responsible for removing them afterwards).

The loop overhead depends on the C compiler version and compilation options; in our tests (GCC 6.3.1, optimization flags "`-Os`"), it appears that the loop has a fixed 38 cycles overhead, and an additional 29 cycles per iteration. We could thus obtain the exact cycle counts for each function call. Since the board is used without any interrupts, measures are perfectly reproducible.

[19]Points in projective coordinates are larger than in affine coordinates, hence constant-time lookup is more expensive.

[20]Notably, fault attacks can also impact hardware RNGs and force them to produce predictable output.

[21]This gives about 9 minutes after boot to make measures, before the counter overflows.

The table 10 lists all measured execution times; they are reported both in raw cycle counts, and as a cost relative to the cost of a multiplication. For the low-level field operations, the implementation uses an internal ABI that does not save registers; the measure was made through a wrapper that adds 31 cycles of overhead per call (these 31 cycles were subtracted to obtain the values in the table). For all operations implemented in assembly, the measured cycle counts match manual counting exactly.

Operation	**Cost (cycles)**	**Cost (rel. to mul)**
Field: multiplication ($*$)	1 574	1.00M
Field: squaring ($*$)	994	0.63M
Field: inversion ($*$)	9 508	6.04M
Field: square root extraction ($*$)	26 962	17.13M
Field: test quadratic residue status ($*$)	9 341	5.93M
Field: cube root extraction ($*$)	31 163	19.80M
Generic curve point addition	16 331	10.38M
Curve point $\times 2$ (doubling)	16 337	10.38M
Curve point $\times 4$	30 368	19.29M
Curve point $\times 8$	41 760	26.53M
Curve point $\times 16$	53 152	33.77M
Constant-time lookup in 8-point window	777	0.49M
Curve point decoding (point decompression)	32 228	20.48M
Curve point encoding (compression)	1 527	0.97M
Generic point multiplication by a scalar	4 493 999	2 855.15M
Generator multiplication by a scalar	1 877 847	1 193.04M
Two combined point multiplications	5 590 769	3 551.95M
MapToField	20 082	12.76M
Icart's map	50 976	32.39M
Hash 48 bytes to a curve point	195 211	124.02M
ECDH: key pair generation	1 937 792	1 231.13M
ECDH: compute shared secret from peer data	4 598 756	2 921.70M
Schnorr signature: generate	2 054 110	1 305.03M
Schnorr signature: verify	5 688 642	3 614.13M

Table 10: All benchmarks. Operations tagged with ($*$) use the internal non-standard ABI that does not preserve registers.

5.3 Other Architectures

While we concentrated on improving performance on the ARM Cortex-M0+, Curve9767 is not necessarily slow on other architectures. Use of a small 14-bit modulus p does not exercise abilities of bigger CPUs at computing multiplications on larger operands. However, many modern CPUs have SIMD units that can compute several operations on small operands in parallel; such units should prove effective at implementing operations on $GF(9767^{19})$ elements.

ARM Cortex-M4. The ARM Cortex-M4 implements the ARMv7-M architecture. It is backward compatible with the ARMv6-M architecture; thus, our implementation should run just fine, with very similar timings, on the M4. However, that CPU offers many extra instructions, including some from the "DSP extension" that incarnate some SIMD abilities. Most interesting are the `smlad` and `smladx` opcodes, that can perform two 16×16 multiplications and add both 32-bit results to a given accumulator register, all in a single cycle; on the M0+, the equivalent operations take 6 cycles (4 cycles for the two multiplications and two additions, and 2 cycles for copies to avoid consuming the multiplication inputs). Moreover, the ARMv7-M instruction set allows full access to all registers, as well as many non-consuming operations, and various literal operands. We expect considerable speed-ups on the M4, compared with the M0+, when optimized assembly leveraging the M4 abilities is written.

x86 with SSE2 and AVX2. The x86 instruction set now includes extensive SIMD instructions. The SSE2 instructions operate on 128-bit registers. The `pmullw` and `pmulhuw` opcodes compute eight 16×16 unsigned multiplications in parallel, returning the low or high 16-bit halves, respectively. On an Intel Skylake core, each instruction has a latency of 5 cycles, but a reciprocal throughput of 0.5 cycles per instruction, meaning that eight full $16 \times 16 \longrightarrow 32$ multiplications can be performed at each cycle. Since polynomial multiplications do not have any carry propagation, considerable internal parallelism can be leveraged. AVX2 opcodes further improve that situation, by offering 256-bit registers and basically doubling all operations: the `vpmullw` and `vpmulhuw` have the same timing characteristics as their SSE2 counterparts, but compute sixteen 16×16 unsigned multiplications in parallel.

Conversely, the inversion in $GF(p)$ to compute x^{-r} from x^r is not parallel, and we suppose that its relative cost within the inversion routine will grow. On the ARM Cortex-M0+, its cost is mostly negligible (110 cycles out of a total of 9 508), but this might not be true in an optimized inversion routine that leverages SSE2 or AVX2 for multiplications and Frobenius operators. In that case, it is possible that for such architectures, more classic projective coordinate systems become more attractive than affine coordinates for point multiplications. Inversion, square roots and cube roots would still be fast enough to provide benefits, when compared with prime-order fields, for conversion to affine coordinates, point compression, and hash-to-curve operations.

6 Conclusion And Future Work

In this article, we presented Curve9767, a new elliptic curve defined over a finite field extension $GF(p^n)$, where both the modulus p and the extension degree n where specially chosen to promote performance on small architectures such as the ARM Cortex-M0+. Our novel results include in particular the following:

- an optimization of Montgomery reduction for a small modulus;
- choosing a modulus p such that these fast reductions can be used but also mutualized as part of a multiplication of polynomials;
- using a finite field extension $GF(p^n)$ to leverage fast Itoh-Tsujii inversion for efficient constant-time curve computations in affine coordinates;
- fast square root and cube root in $GF(p^n)$.

In total, generic curve point multiplication is about 1.29 times slower with Curve9767 than the optimized Curve25519 Montgomery ladder, on the ARM Cortex-M0+. On the other hand, our curve offers very fast routines for a number of other operations (e.g. point compression, or hash-to-curve); maybe more importantly, it has prime order, which simplifies analysis for use in larger protocols. The relatively small difference in performance shows that affine coordinates and fast inversion can be a viable implementation strategy for an elliptic curve, offering an alternate path to the projective coordinate systems that have been prevalent in elliptic curve implementation research over the last two decades.

Future work on Curve9767 will include the following:

- Making optimized implementations for other architectures, notably the ARM Cortex-M4, and x86 systems with SSE2/AVX2. Whether SIMD opcodes will allow competitive performance on "big CPUs" is as yet an open question.
- Exploring formal validation of the correctness of the implementations. Computations in $GF(p^n)$ have some informal advantages in that respect: since they don't have carries to propagate, they don't suffer from rare carry propagation bugs. Moreover, a small modulus p allows for exhaustive tests: for instance, the correction of our fast Montgomery reduction routine modulo p has been exhaustively tested for all inputs x such that $1 \leq x \leq 3\,654\,952\,486$.
- Exploring other field choices, in particular smaller moduli p for use in 8-bit systems that can only do $8 \times 8 \longrightarrow 16$ multiplications. This might be combined with other field extensions such as $GF(p)[z]/(z^n - z - c)$ for some constant c. Such an extension polynomial would increase the cost of Frobenius operators, but also expand the set of usable values for p.

References

1. Accredited Standard Committee X9, Inc., *ANSI X9.62: Public Key Cryptography for the Financial Services Industry: the Elliptic Curve Digital Signature Algorithm (ECDSA)*, 2005.
2. Accredited Standard Committee X9, Inc., *ANSI X9.63: Public Key Cryptography for the Financial Services Industry: Key Agreement and Key Transport Using Elliptic Curve Cryptography*, 2001.
3. C. Ambrose, J. Bos, B. Fay, M. Joye, M. Lochter and B. Murray, *Differential Attacks on Deterministic Signatures*,
`https://eprint.iacr.org/2017/975`
4. T. Arcieri, H. de Valence and I. Lovecruft, *The Ristretto Group*,
`https://ristretto.group/`
5. S. Arita, K. Matsuo, K. Nagao and M. Shimura, *A Weil Descent Attack against Elliptic Curve Cryptosystems over Quartic Extension Fields*, IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, vol. E89-A, issue. 5, 2006.
6. D. Bailey and C. Paar, *Optimal extension fields for fast arithmetic in public key algorithms*, Advances in Cryptology – Crypto 1998, Lecture Notes in Computer Science, vol. 1462, 1998.
7. R. Balasubramanian and N. Koblitz, *The Improbability That an Elliptic Curve Has Subexponential Discrete Log Problem under the Menezes-Okamoto-Vanstone Algorithm*, Journal of Cryptology, vol. 11, pp. 141-145, 1998.
8. D. Bernstein, *Curve25519: new Diffie-Hellman speed records*, PKC 2006, Lecture Notes in Computer Science, vol. 3958, pp. 207–228, 2006.
9. D. Bernstein, N. Duif, T. Lange, P. Schwabe and B.-Y. Yang, *High-speed high-security signatures*, Journal of Cryptographic Engineering, vol. 2, issue 2, pp. 77-89, 2012.

10. D. Bernstein and T. Lange, *Explicit-Formulas Database*, `https://hyperelliptic.org/EFD/`
11. D. Bernstein and T. Lange, *SafeCurves: choosing safe curves for elliptic-curve cryptography*, `https://safecurves.cr.yp.to/`
12. I. Biehl, B. Meyer and V. Müller, *Differential Fault Attacks on Elliptic Curve Cryptosystems*, Advances in Cryptology - CRYPTO 2000, Lecture Notes in Computer Science, vol. 1880, pp. 131-146, 2000.
13. S. Blake-Wilson, N. Bolyard, V. Gupta, C. Hawk and B. Moeller, *Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)*, `https://tools.ietf.org/html/rfc4492`
14. R. Brent and H. Kung, *Systolic VLSI arrays for linear-time GCD computation*, VLSI 1983, pp. 145-154, 1983.
15. E. Brier, J.-S. Coron, T. Icart, D. Madore, H. Randriam and M. Tibouchi, *Efficient Indifferentiable Hashing into Ordinary Elliptic Curves*, Advances in Cryptology - CRYPTO 2010, Lecture Notes in Computer Science, vol. 6223, pp. 237-254, 2010.
16. Buildroot, *Making Embedded Linux Easy*, `https://buildroot.org/`
17. CAST, Inc., *CACHE-CTRL: AHB Cache Controller Core*, `http://www.cast-inc.com/ip-cores/peripherals/amba/cache-ctrl/`
18. Certicom Research, *SEC 2: Recommended Elliptic Curve Domain Parameters*, `http://www.secg.org/sec2-v2.pdf`
19. C. Costello and P. Longa, *Four$\mathbb{Q}$: Four-Dimensional Decompositions on a $\mathbb{Q}$-curve over the Mersenne Prime*, Advances in Cryptology - ASIACRYPT 2015, Lecture Notes in Computer Science, vol. 9452, pp. 214-235, 2015.
20. C. Cremers and D. Jackson, *Prime, Order Please! Revisiting Small Subgroup and Invalid Curve Attacks on Protocols using Diffie-Hellman*, IEEE 32nd Computer Security Foundations Symposium (CSF), 2019.
21. C. Diem, *The GHS attack in odd characteristic*, Journal of the Ramanujan Mathematical Society, vol. 18, issue 1, pp. 1-32, 2003.
22. M. Düll, B. Haase, G. Hinterwälder, M. Hutter, C. Paar, A. Sánchez and P. Schwabe, *High-speed Curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers*, Designs, Codes and Cryptography, vol. 77, issue 2-3, pp.493-514, 2015.
23. T. ElGamal, *A public key cryptosystem and a signature scheme based on discrete logarithm*, IEEE Transactions on Information Theory, vol. 31, pp. 469-472, 1985.
24. A. Faz-Hernandez, S. Scott, N. Sullivan, R. Wahby and C. Wood, *Hashing to Elliptic Curves*, Internet-Draft (November 02, 2005), `https://tools.ietf.org/html/draft-irtf-cfrg-hash-to-curve-05`
25. , G. Frey and H.-G. Rück, *A remark concerning m-divisibility and the discrete logarithm problem in the divisor class group of curves*, Mathematics of Computation, vol. 62, issue 206, pp. 865-874, 1994.
26. G. Frey, M. Müller and H.-G. Rück, *The Tate pairing and the discrete logarithm applied to elliptic curve cryptosystems*, IEEE Transactions on Information Theory, vol. 45, issue 5, pp. 1717-1719, 1999.
27. R. Gallant, J. Lambert and S. Vanstone, *Faster Point Multiplication on Elliptic Curves with Efficient Endomorphisms*, Advances in Cryptology - CRYPTO 2001, Lecture Notes in Computer Science, vol. 20139, pp. 190-200, 2001.
28. P. Gaudry, *Index calculus for abelian varieties of small dimension and the elliptic curve discrete logarithm problem*, Journal of Symbolic Computation, vol. 44, issue 12, pp. 1690-1702, 2009.
29. P. Gaudry, F. Hess and N. Smart, *Constructive and destructive facets of Weil descent on elliptic curves*, Journal of Cryptology, vol. 15, issue 1, pp. 19-46, 2002.
30. T. Grandlund and P. Montgomery, *Division by Invariant Integers using Multiplication*, ACM SIGPLAN Notices, vol. 29, issue 6, pp. 61-72, 1994.

31. B. Haaser and B. Labrique, *AuCPace: Efficient verifier-based PAKE protocol tailored for the IIoT*, IACR Transactions on Cryptographic Hardware and Embedded Systems, 2019(2), pp. 1-48, 2019.
32. D. Hankerson, A. Menezes and S. Vanstone, *Guide to Elliptic Curve Cryptography*, Springer-Verlag, 2003.
33. T. Icart, *How to hash into elliptic curves*, Advances in Cryptography - CRYPTO 2009, Lecture Notes in Computer Science, vol. 5677, pp. 303-316, 2009.
34. Information Technology Laboratory, *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*, National Institute of Standard and Technology, FIPS 202, 2015.
35. T. Itoh and S. Tsujii, *A Fast Algorithm for Computing Multiplicative Inverses in $GF(2^m)$ Using Normal Bases*, Information and Computation, vol. 78, pp. 171-177, 1988.
36. S. Josefsson and I. Liusvaara, *Edwards-Curve Digital Signature Algorithm (EdDSA)*, `https://tools.ietf.org/html/rfc8032`
37. A. Karatsuba and Y. Ofman, *Multiplication of Many-Digital Numbers by Automatic Computers*, Proceedings of the USSR Academy of Sciences, vol. 145, pp. 293-294, 1962.
38. A. Langley, M. Hamburg and S. Turner, *Elliptic Curves for Security*, `https://tools.ietf.org/html/rfc7748`
39. D.-P. Le and B. Nguyen, *Fast Point Quadrupling on Elliptic Curves*, Proceedings of the Third Symposium on Information and Communication Technology (SoICT '12), pp. 218-222, 2012.
40. P. Longa and F. Sica, *Four-Dimensional Gallant-Lambert-Vanstone Scalar Multiplication*, Journal of Cryptology, vol. 27, issue 2, pp. 248-283, 2014.
41. luigi1111 and R. Spagni, *Disclosure of a Major Bug in CryptoNote Based Currencies*, `https://www.getmonero.org/2017/05/17/disclosure-of-a-major-bug-in-cryptonote-based-currencies.html`
42. A. Menezes, T. Okamoto and S. Vanstone, *Reducing elliptic curve logarithms to logarithms in a finite field*, IEEE Transactions on Information Theory, vol. 39, issue 5, pp. 1639-1646, 1993.
43. Microchip, *SAM D20 Family* (microcontroller datasheet), `http://ww1.microchip.com/downloads/en/DeviceDoc/SAM_D20_%20Family_Datasheet_DS60001504C.pdf`
44. P. Mihăilescu, *Optimal Galois field bases which are not normal*, presented at the Workshop on Fast Software Encryption in Haifa, 1997.
45. P. Montgomery, *Modular multiplication without trial division*, Mathematics of Computation, vol. 44, pp. 519–521, 1985.
46. Y. Nir, S. Josefsson and M. Pegourie-Gonnard, *Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS) Versions 1.2 and Earlier*, `https://tools.ietf.org/html/rfc8422`
47. T. Nishinaga and M. Mambo, *Implementation of µNaCl on 32-bit ARM Cortex-M0*, IEICE Transactions on Information and Systems, vol. E99-D, issue 8, 2016.
48. PARI/GP, `https://pari.math.u-bordeaux.fr/`
49. D. Poddebniak, J. Somorovsky, S. Schinzel, M. Lochter and P. Rösler, *Attacking Deterministic Signature Schemes Using Fault Attacks*, 2018 IEEE European Symposium on Security and Privacy (EuroS&P), pp. 338-352, 2018.
50. T. Pornin, *Constant-Time Mul*, `https://www.bearssl.org/ctmul.html`
51. T. Pornin, *Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)*, `https://tools.ietf.org/html/rfc6979`
52. QEMU, *the FAST! processor emulator*, `https://www.qemu.org/`
53. J. Renes, C. Costello and L. Batina, *Complete addition formulas for prime order elliptic curves*, Advances in Cryptology – Eurocrypt 2016, Lecture Notes in Computer Science, vol. 9665, pp. 403-428, 2016.

54. P. Schwabe and D. Sprenkels, *The complete cost of cofactor $h = 1$*, Progress in Cryptology - INDOCRYPT 2019, Lecture Notes in Computer Science, vol. 11898, pp. 375-397, 2019.
55. A. Shallue and C. van de Woestijne, *Construction of rational points on elliptic curves over finite fields*, Algorithm Number Theory Symposium - ANTS 2006, Lecture Notes in Computer Science, vol. 4076, pp. 510-524, 2006.
56. J. Thomas, J. Keller and G. Larsen, *The calculation of multiplicative inverses over $GF(p)$ efficiently where p is a Mersenne prime*, IEE Transactions on Computers, vol. 35, issue 5, pp. 478-482, 1986.
57. M. Ulas, *Rational Points on Certain Hyperelliptic Curves over Finite Fields*, Bulletin of the Polish Academy of Sciences - Mathematics, vol. 55, issue 2, pp. 97-104, 2007.
58. W. Zhang, D. Lin, H. Zhang, X. Zhou and Y. Gao, *A Lightweight FourQ Primitive on ARM Cortex-M0*, 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications / 12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE), 2018.

A Unused or Failed Ideas

It is uncommon in scientific articles to describe failures. However, we feel that the ideas described in this section, while not applicable or not interesting enough in our case, might be of interest in other contexts.

A.1 Signed Integers

All our computations with integers modulo p used nonnegative integers. When such integers are represented as values in the $0..p-1$ range, we need the product of two such values to fit in the output operand size; when our largest multiplication opcode produces a 32-bit output, this limits the modulus p to 2^{16}. In particular, that implementation strategy cannot cope with modulus $p = 65\,537$ (a Mersenne prime) since multiplying $p-1$ with itself would yield 2^{32}, truncated to 0 because of the limited range of the multiplier output.

This limitation can be worked around by using *signed* integers. For instance, values modulo 65 537 can be represented by integers in the $-32\,768..+32\,768$ range. In that case, the maximum absolute value of a product of two such integers will be 2^{30}, well within the representation limit of signed integers on 32-bit words (-2^{31} to $2^{31}-1$).

In our case, we want to mutualize modular reductions, meaning that we need to accumulate intermediate results without overflowing the representable range of values. With an unsigned representation of $GF(p)$ and an extension polynomial $z^n - 2$, this requires $(2n-1)p^2 < 2^{32}$ (here we only consider representability, not the specificities of the fast Montgomery reduction). If using a *signed* representation of $GF(p)$, then values are only up to $\lceil p/2 \rceil$ in absolute value; the representable range is halved (2^{31}) to account for the sign bit, leading to the new requirement: $(2n-1)p^2/4 < 2^{31}$. Generally speaking, using signed integers increases the possible range of prime moduli p by a factor $\sqrt{2}$.

We did not use signed integers for Curve9767, for the following reasons:

- Signed integers make some operations, in particular Montgomery reduction, but also combined additions or subtractions of two elements at a time, more complicated and expensive.

– Conversely, the main parameter for performance is the field extension degree n, and the larger range is not enough to allow us to obtain a field of at least 250 bits with a prime degree n smaller than 19. The largest prime p such that $33(\lceil p/2 \rceil)^2 < 2^{31}$ and $z^{17} - 2$ is irreducible in $GF(p)[z]$ is $p = 15\,913$ and leads to a field of size $p^{17} \approx 2^{237.28}$, which falls too short of the target "128-bit security level", even taking into account the traditional allowance for small multiplicative constants to cost estimates.

Therefore, there was no benefit to using signed representation in our case.

The technique may still be useful in other contexts, in particular when working with Mersenne primes, such as 17, 257 or 65 537. The internal representation range of values can even be slightly extended to allow for easier and faster reduction. For instance, the following routine computes a multiplication of two integers modulo 65 537:

```
int32_t
mul_mod_65537(int32_t x, int32_t y)
{
    x *= y;
    x = (x & 0xFFFF) - (x >> 16);
    x += 32767;
    x = (x & 0xFFFF) - (x >> 16);
    return x - 32767;
}
```

If the two inputs are in the −46 340 to +46 340 range, then the intermediate product will fit in the representable range (no overflow); then the first reduction step brings it down to the −32 767 to +98 302 range. With the addition of the 32 767 constant, the range becomes 0..+131 069, and the second reduction step brings it down to −1..+65 535. The final subtraction of 32 767 (compensating the addition of the same constant two lines before) makes the final range −32 768..+32 768, i.e. fully reduced.

A.2 Towers of Fields

A preliminary idea for this work was to use $GF(p^n)$ with p being a prime with easy reduction, and $n = 2^m$ a power of two. In particular, one could take $p = 65\,537$, and $n = 16$ (using the "signed integer" representation detailed in the previous section). The field $GF(p^{16})$ can be defined as a quadratic extension of $GF(p^8)$, itself a quadratic extension of $GF(p^4)$, and so on. In $GF(p)$, we can choose a non-quadratic residue d_0. Then, we recursively define $GF(p^{2^{i+1}})$ as involving d_{i+1}, a formal square root of d_i. Operations in such a tower of fields are inexpensive; there are natural analogs to Karatsuba multiplication. Inversion is efficient:

$$\frac{1}{u_0 + d_i u_1} = \frac{u_0 - d_i u_1}{u_0^2 + d_{i-1} u_1^2}$$

We can thus compute inversion in $GF(p^{2^{i+1}})$ at the cost of two squarings, two multiplications and one inversion in $GF(p^{2^i})$; this last operation then applies the same method recursively, down the tower. At the lowest level, only an inversion in $GF(p)$ is required. Exact performance depends on the implementation architecture (notably its abilities are parallel evaluation, with

SIMD units) but getting inversion cost down to only three times that of a multiplication is plausible.

We abandoned that idea because curves based on field towers with quartic extension degrees seem vulnerable to Weil descent attacks; an attack with asymptotic cost $O(p^{3n/8})$ has been described[28]. Using such a field for our curve would have required a complex argumentation to explain that the attack cost would be still too high in practice for a specific size; this would have been a "hard sale".

Using field towers may still be useful in different contexts, e.g. to build universal hash functions for MAC-building purposes, especially with small Mersenne primes such as $p = 257$, for some lightweight architectures.

A.3 Alternate Inverse Computations

The Itoh-Tsujii inversion algorithm that we used in section 3.6 is an optimization on Fermat's little theorem. There are other strategies for computing inversions in a finite field; we present two here, which work, but have worse performance than Itoh-Tsujii.

Binary GCD. The binary GCD algorithm was introduced under the name *plus-minus* by Brent and Kung[14]. Nominally for inverting integers against an odd modulus, it can be adapted to polynomials, and is in general a *division* algorithm. Consider the problem of dividing x by y in finite field $GF(p^n)$, the finite field being defined with the extension polynomial $\mathcal{M}$ (crucially, the coefficient of degree zero of $\mathcal{M}$ is not equal to zero). We have $y \neq 0$. We consider four variables a, b, u and v which are polynomials in $GF(p)[z]$ (all will have degree less than n, except for the starting value of b, which is equal to $\mathcal{M}$), and an extra small integer δ. The process is described in algorithm 5.

Algorithm 5 Division in $GF(p^n)$ with binary GCD

```
Require: $x, y \in GF(p^n), y \neq 0, GF(p^n) = GF(p)[z]/\mathcal{M}$
Ensure: $x/y$
 1: $a \leftarrow y, b \leftarrow \mathcal{M}, u \leftarrow x, v \leftarrow 0$
 2: $\delta \leftarrow 0$
 3: for $1 \leq i \leq 2n$ do
 4:     if $a_0 = 0$ then
 5:         $(a, u) \leftarrow (a/z, u/z \bmod \mathcal{M})$
 6:         $\delta \leftarrow \delta - 1$
 7:     else if $\delta \geq 0$ then
 8:         $(a, u) \leftarrow ((b_0 a - a_0 b)/z, (b_0 u - a_0 v)/z \bmod \mathcal{M})$
 9:     else
10:         $(b, v) \leftarrow ((a_0 b - b_0 a)/z, (a_0 v - b_0 u)/z \bmod \mathcal{M})$
11:         $(a, b, u, v) \leftarrow (b, a, v, u)$
12:         $\delta \leftarrow -\delta$
13: return $v/b_0$
```

The following invariants are maintained throughout the algorithm:

- $ax = uy \bmod \mathcal{M}$ and $bx = vy \bmod \mathcal{M}$.
- $b_0 \neq 0$.
- If the maximum possible size of a is n_a (i.e. the highest degree of a non-zero coefficient is at most $n_a - 1$) and the maximum size of b is n_b, then $\delta = n_a - n_b$, and every iteration decreases $n_a + n_b$ by 1.

The algorithm converges after a maximum of $2n$ iterations on $a = 0$ and b a polynomial of degree 0; at that point, we have $b_0 x = vy \bmod \mathcal{M}$, hence the result. Classical descriptions of this algorithm use a test on a to stop when it reaches 0; here, we use a constant number of iterations to help with constant-time implementations.

In a constant-time implementation, each iteration involves reading and rewriting four polynomials (a, b, u and v, with multipliers in $GF(p)$). Some optimizations can be obtained with the following remarks:

- The decisions for k consecutive iterations depend only on δ and the k low degree coefficients of a and b. It is possible to aggregate k iterations working only on these values (which might fit all in registers) and mutualize the updates on a, b, u and v into a multiplication of each by polynomials of degree less than k (with some divisions by z^k).
- If computing an inversion (i.e. $x = 1$) instead of a division, u is initially small and some of the first iterations can be made slightly faster.
- In the last iteration, since we are interested only in v, we can avoid updating a, b and u.

Nevertheless, our attempts at optimizing this algorithm did not yield a cost lower than 12 times the cost of a multiplication, hence not competitive with Itoh-Tsujii.

Thomas-Keller-Larsen. In 1986, Thomas, Keller and Larsen described different inversion algorithms for modular integers[56]; their main algorithm was dedicated to Mersenne primes, but another one was more generic and can be adapted to polynomials when working modulo $\mathcal{M} = z^n - c$. The main idea is to repeatedly multiply the value to invert with custom factors of increasing degree, each shrinking the value by one element. Algorithm 6 describes the process.

Algorithm 6 Inversion in $GF(p^n)$ with the Thomas-Keller-Larsen algorithm

```
Require: $y \in GF(p^n), y \neq 0, GF(p^n) = GF(p)[z]/(z^n - c)$
Ensure: $1/y$
1: $a \leftarrow y, r \leftarrow 1$
2: for $i = n - 1$ down to 1 do
3:     if $a_i \neq 0$ then
4:         $q_{n-i} \leftarrow 1/a_i$
5:         for $j = n - 1 - i$ down to 0 do
6:             $q_j = (1/a_i) \sum_{k=j+1}^{\min(n-i,i+j)} q_k a_{i+j-k}$
7:         $a \leftarrow qa + c \bmod z^i$ (where $q = \sum_{j=0}^{n-i} q_j z^j$)
8:         $r \leftarrow qr$
9: return $r/r_0$
```

The algorithm works on the following invariants:

- $ar = y$.
- At the entry of each iteration of the outer loop, the degree of a is at most i; upon exit, it is at most $i - 1$.

The polynomial q which is computed in each loop iteration (when $a_i \neq 0$) is the unique polynomial such that $qa = z^n + t$ for a polynomial t of degree at most $i - 1$. In the finite field, we have $qa = t + c \bmod (z^n - c)$, hence multiplying a by q (and r by q too, to maintain the first invariant) yields $t + c$, of degree at most $i - 1$. Since t has degree less than i, it can be obtained by considering the product qa modulo z^i.

In a constant-time implementation, q is always computed even if $a_i = 0$: the constant-time inversion of 0 is assumed to yield some value, which we ignore, and a fixing step is added to avoid modifying a and r in such a case. This fixing step is only linear in the degree n, thus inexpensive relatively to the rest of the algorithm.

We can avoid computing an inversion in $GF(p)$ at each iteration by multiplying $a_i^{n-i}q$ instead of q; however, this implies computing the powers of a_i and saving them, increasing memory traffic. Depending on the implementation platform, this may decrease or increase overall cost.

Computing each q grows in cost as i approaches $n/2$, then decreases afterwards, because then the degree of a becomes smaller and smaller. However, the value r is the product of $n-1$ polynomials q of degrees 1 to $n-1$, and cannot really be made less expensive than the cost of $(n-1)/2$ multiplications in the field, making this algorithm less efficient than the Itoh-Tsujii method.