2DT-GLS: Faster and exception-free scalar multiplication in the GLS254 binary curve

Marius A. Aardal and Diego F. Aranha

Department of Computer Science, Aarhus University, Denmark
{maardal,dfaranha}@cs.au.dk

Abstract. We revisit and improve performance of arithmetic in the binary GLS254 curve by introducing the 2DT-GLS scalar multiplication algorithm. The algorithm includes theoretical and practice-oriented contributions of potential independent interest: (i) for the first time, a proof that the GLS scalar multiplication algorithm does not incur exceptions, such that faster incomplete formulas can be used; (ii) faster dedicated atomic formulas that alleviate the cost of precomputation; (iii) a table compression technique that reduces the storage needed for precomputed points; (iv) a refined constant-time scalar decomposition algorithm that is more robust to rounding. We also present the first GLS254 implementation for Armv8. With our contributions, we set new speed records for constant-time scalar multiplication by 34.5% and 6% on 64-bit Arm and Intel platforms, respectively.

Keywords: Binary elliptic curves · Software implementation · GLS254.

1 Introduction

Elliptic Curve Cryptography (ECC) has become the *de facto* standard for instantiating public key cryptography, with security based on the conjectured-as-exponential hardness of solving discrete logarithms over elliptic curve groups (ECDLP problem). Scalar multiplication, in particular for the unknown point scenario, is the most expensive operation in cryptographic protocols with security guarantees based on the ECDLP. Since its introduction in 1985, there was plenty of research in finding efficient and secure implementation strategies, and choosing optimal parameters to improve performance [2,23,5].

An early milestone in this research was the idea due to Gallant-Lambert-Vanstone (GLV) [13] of exploiting efficient endomorphisms to accelerate scalar multiplication. In the large characteristic case with the curve $E : y^2 = x^3 + b$ defined over $\mathbb{F}_p$ for prime p, it initially manifested as evaluating $\psi : (x, y) \rightarrow (\beta x, y)$ for β a non-trivial cube root of unity. The technique was later generalized to Galbraith–Lin–Scott (GLS) curves defined over $\mathbb{F}_{p^2}$, and to exploit two or more endomorphisms as in the Four$\mathbb{Q}$ curve [27] and the genus-2 case [4]. In the characteristic 2 case, Koblitz curves are the classical example of curves equipped with endomorphisms [32]; a class later extended to include binary GLS curves [16].

Beyond performance, implementation security is also relevant, especially on embedded targets where side-channel attacks are more feasible. The classical countermeasure against these attacks is to formulate the arithmetic in a *regular* way, such as constant-time implementation against timing attacks. This translates to removing secret-dependent branches and memory accesses, and employing *complete* point addition formulas without corner cases [33]. In the case of exploiting endomorphisms, additional care needs to be taken for a correct and secure implementation, such as using the GLV-SAC recoding technique [10] or by explicitly proving correctness of the specific scalar multiplication algorithm [9,5].

In this paper, we revisit the implementation of scalar multiplication in binary GLS curves at the 128-bit security level by improving efficiency and correctness of implementations of the GLS254 curve. Our contributions are:

- A variant of the GLS scalar multiplication algorithm that changes the computation/storage trade-off to use a 2D table. The resulting 2DT algorithm spends more precomputation to reduce point additions in the main loop.
- The first proof of correctness for the GLS scalar multiplication algorithm with the λ-projective group law from [31]. We show that there are no corner cases in the main loop of the algorithm (with exception of possibly the last iteration), which enables the use of faster incomplete formulas.
- Faster dedicated formulas to reduce the cost of precomputation, and a table compression technique that exploits the endomorphism to reduce storage.
- A refined scalar decomposition algorithm that can be easily implemented in constant time. The algorithm has robust parity and length guarantees that fill some gaps in previous works [31].
- An efficient formulation of arithmetic in $\mathbb{F}_{2^{254}}$ targeting Arm processors. The field arithmetic uses the interleaved representation proposed in the CHES'16 Rump Session [30]. We take the opportunity to include this formulation in the formal research literature, initially presented informally. Furthermore, this also closes affirmatively a question posed in [21] about the efficiency of binary curves in Armv8 processors, deemed "unclear".

With these contributions, we obtain speed records in the 64-bit Armv8 and Intel platforms, improving on previous results by 34.5% and 6%, respectively. While the latter speedup may seem small, we remark that it comes after decades of successful research in improving performance of ECC, so diminishing returns are expected. Due to the upcoming move to post-quantum cryptography, these techniques could be of limited practicality, but we believe they are relevant for applications not necessarily needing long-term security and involving the computation of many scalar multiplications, such as private set intersection protocols [34]. The proof and table compression technique may find further application in accelerating GLV/GLS scalar multiplication in pairing-based cryptography.

The rest of the document is organized as follows. Section 1 discusses preliminaries on binary GLS curves and their efficient implementation. Section 3 introduces the 2DT-GLS algorithm and its correctness proof. Section 4 pushes these ideas further by presenting the scalar decomposition algorithm, followed by dedicated formulas in Section 5. Section 6 discusses the implementation of field arithmetic in Armv8, with experimental results in Section 7. The interested reader will also find a treatment of point compression for binary GLS curves in the Appendix, together with formulas that did not fit in the main body.

2 Preliminaries

An ordinary *binary elliptic curve* in Weierstrass form is defined as

$$E/\mathbb{F}_q : y^2 + xy = x^3 + ax^2 + b \tag{1}$$

with $q = 2^m$ and coefficients $a, b \in \mathbb{F}_q,\ b \neq 0$. For any extension field $\mathbb{F}_{q^k}$, the points $P = (x, y) \in \mathbb{F}_{q^k} \times \mathbb{F}_{q^k}$ that satisfy the equation form an abelian group $E_{a,b}(\mathbb{F}_{q^k})$ together with a point at infinity ∞, which acts as the identity. The group law is denoted with additive notation $P + Q$, such that the scalar multiplication operation is written as kP.

2.1 Binary GLS curves

In the interest of defining notation for later use, we briefly summarize the theory of binary GLS curves from [16].

Let E be an ordinary binary curve as defined previously. From Hasse's theorem, $\#E(\mathbb{F}_q) = q + 1 - t$ for some trace t satisfying $|t| \leq 2\sqrt{q}$. Pick some $a' \in \mathbb{F}_{q^2}$ with $\mathrm{Tr}'(a') = 1$, where Tr' is the field trace from $\mathbb{F}_{q^2}$ to $\mathbb{F}_2$ defined as $\mathrm{Tr}'(c) = \sum_{i=0}^{2m-1} c^{2^i}$. It can be shown that $E' = E_{a',b}$ is the quadratic twist of E over $\mathbb{F}_{q^2}$ with $\#E'(\mathbb{F}_{q^2}) = (q-1)^2 + t^2$, and that E and E' are isomorphic over $\mathbb{F}_{q^4}$ under an involutive twisting isomorphism ϕ.

An endomorphism ψ over $\mathbb{F}_{q^2}$ can be constructed for E' by composing ϕ with the q-power Frobenius map π as $\psi = \phi\pi\phi^{-1}$. Evaluating ψ over points $P \in E'(\mathbb{F}_{q^2})$ only requires field additions [31].

The curve E' would in this scenario be referred to as a binary GLS curve. Assume $\#E'(\mathbb{F}_{q^2}) = hr$ where h is a small cofactor and r is prime. Let $\mathcal{G}$ be the unique order-r subgroup $E'(\mathbb{F}_{q^2})[r]$. Then ψ restricted to $\mathcal{G}$ has an eigenvalue $\mu \in \mathbb{Z}$ so that for $P \in \mathcal{G}$, $\psi(P) = \mu P$. The eigenvalue satisfies that $\mu^2 = -1 \bmod r$.

2.2 λ-projective coordinates for GLS scalar multiplication

In [31], Oliveira et al. introduced the λ-projective coordinate system. To date, it is empirically the most efficient point representation for binary elliptic curves. Given an affine point $P = (x, y)$ with $x \neq 0$, its λ-affine representation is (x, λ) with $\lambda = x + \frac{y}{x}$. In λ-projective coordinates, the affine point can be represented by any triple $(X, \varLambda, Z)$ with $X = xZ$, $\varLambda = \lambda Z$ and $Z \neq 0$. The point at infinity can now be represented as $\infty = (1, 1, 0)$. The curve equation in (1) is correspondingly transformed to

$$(\varLambda^2 + \varLambda Z + aZ^2)X^2 = X^4 + bZ^4. \tag{2}$$

The constant-time binary GLS scalar multiplication algorithm from [31] is included in Algorithm 1. It is a constant-time left-to-right double-and-add algorithm using λ-projective coordinates, combining the Joye-Tunstall regular recoding algorithm [20] with the GLV interleaving technique. For the GLV method, the scalar k is decomposed into two subscalars k_1, k_2 such that $k \equiv k_1 + k_2\mu$

Algorithm 1: Constant-time scalar multiplication (Oliveira et al. [31])

```
Input: $P \in \mathcal{G}$ in $\lambda$-affine coordinates, $k \in [1, r-1]$, window size $w$
Output: $kP$ in $\lambda$-affine coordinates
1  Decompose $k$ into subscalars $k_1, k_2$.
2  $c_j \leftarrow 1 - (k_j \bmod 2)$ for $j = 1, 2$.
3  $k_j \leftarrow k_j + c_j$
4  $\ell \leftarrow \lceil \frac{m}{w-1} \rceil$
5  Compute width-$w$ length-$\ell$ odd signed regular recoding $\overline{k}_1, \overline{k}_2$ of $k_1, k_2$.
6  Compute $T[i] = (2i+1)P$ for all odd $i \in \{0, \ldots, 2^{w-2} - 1\}$.
7  Convert $T$ to $\lambda$-affine coordinates using a simultaneous inversion.
8  Perform a linear pass over $T$ to recover $P_{j,\ell-1} = \overline{k}_{j,\ell-1}P$ for $j = 1, 2$.
9  $Q \leftarrow P_{1,\ell-1} + \psi(P_{2,\ell-1})$
10 for $i$ from $\ell - 2$ downto 0 do
11     $Q \leftarrow 2^{w-2}Q$
12     Perform a linear pass over $T$ to recover $P_{j,i} = \overline{k}_{j,i}P$ for $j = 1, 2$.
13     $Q \leftarrow 2Q + P_{1,i} + \psi(P_{2,j})$
14 $Q \leftarrow Q - c_1P - c_2\psi(P)$
15 Convert $Q$ to $\lambda$-affine coordinates.
16 return $Q$;
```

(mod r) and the subscalars are of roughly half the length of k. The two smaller scalar multiplications can then be computed in an interleaved fashion to save half of the point doublings.

The Joye-Tunstall regular recoding algorithm ensures the regular form of the algorithm and allows for a width-w windowing strategy. Specifically, the subscalars are recoded into $\ell = \lceil m/(w-1) \rceil$ odd signed digits of $w-1$ bits using Algorithm 6 from [9] (which is a constant-length modification of Algorithm 6 from [20]). By initially computing a table $T[i] = (2i+1)P$ for all positive digits (in a phase known as the precomputation), the main loop can process the scalars one digit at a time, reducing the number of iterations by a factor $w-1$. To be resistant against (cache-)timing attacks, each lookup requires a linear pass over the entire table, and there can be no branches dependent on c_j, k_j. An additional consideration is that the regular recoding algorithm requires the subscalars to be odd. To ensure this, the subscalars are modified to be odd in line 3, and at the end the result is corrected at the cost of two point additions.

However, both [31] and the subsequent [30] suffer from a lack of rigor. First and foremost, no proof has been presented for correctness of the scalar multiplication algorithm. The λ-projective group law formulas are incomplete, so it could potentially fail in corner cases. It also relies on *ad-hoc* tricks for constant-time scalar decomposition, with no proof of correctness or length guarantees.

2.3 GLS254 and the choice of parameters

Previous works have benchmarked their implementation of scalar multiplications over a GLS curve specially crafted for efficiency at the 128-bit security level. For the GLS254 curve, one chooses $m = 127$, such that the base field can be defined as $\mathbb{F}_q \equiv \mathbb{F}_2[z]/(z^{127}+z^{63}+1)$ and its quadratic extension as $\mathbb{F}_{q^2} \equiv \mathbb{F}_q[u]/(u^2+u+1)$. The curve coefficients should be chosen to have minimal Hamming weight such that multiplying by them is as efficient as possible. We performed a parameter search that reproduced the curve chosen at [30]. By fixing $a' = u$ and searching for the shortest $b = (z^i + 1)$ such that the curve has order $2r$ for prime r, we were able to confirm that $i = 27$ is the smallest choice, giving a 254-bit r. This means that a multiplication by b can be computed with a single shifted addition.

To protect against Weil descent and generalized Gaudry-Hess-Smart (gGHS) attacks [14,18], several precautions must be taken. We pick m to be prime, as is the case for GLS254. In addition, the choice of b must be verified to not allow the attack, which happens with negligibly small probability for random b [16]. We used the MAGMA implementation of [7] available at https://github.com/JJChiDguez/gGHS-check to clear our particular choice. This particular check, together with the curve generation method geared towards efficiency, satisfies rigidity concerns [3]. We stress that the ECDLP in binary curves remains infeasible for the parameter range used in this work [11].

3 Scalar multiplication in GLS curves

In this section, we begin by presenting a new scalar multiplication variant for binary GLS curves. It combines the Shamir-Straus' trick [8] for multiple scalar multiplication with a new table compression technique using the GLS endomorphism. We refer to it as the 2DT variant because it builds a two-dimensional table $T[i, j] = iP + j\psi(P)$ instead of $T[i] = iP$. Then, in Subsection 3.2, we prove that the GLS scalar multiplication algorithms are exception-free.

3.1 The 2DT variant

As in some *fast* variants of the Shamir-Straus' trick [8] for multiple scalar multiplication, the idea is to precompute $T[i, j] = iP + j\psi(P)$ for odd i, j. In the scalar multiplication loop, we then save roughly one point addition per iteration of the main loop by computing $2Q + T[i, j]$ instead of $2Q + T[i] + \psi(T[j])$.

This method was previously deemed noncompetitive due to the blowup in the size of the table. Because the subscalar regular recoding uses signed digits, we need to efficiently retrieve $s_1 iP + s_2 j\psi(P)$ for any $i, j \in \{1, ..., 2^{w-1} - 1\}$ and sign combination $s_1, s_2 \in \{\pm 1\}$. The standard approach would be to build a table of $iP \pm j\psi(P)$ and then use conditional negations to get the two other combinations. The 2D table would then store $2^{2(w-2)+1}$ points. Even with specialized formulas for the precomputation, the cost in terms of storage and field operations is too high compared to the conventional 1DT algorithm.

The crucial new observation is that the efficiently computable GLS endomorphism ψ can also be used to compress the 2D table by a factor of 2. As $\psi^2(P) = -P$ for any $P \in \mathcal{G}$, we obtain the identity

$$-\psi(T[j, i]) = iP - j\psi(P).$$

It implies that we can generate all combinations from a table that only stores $iP + j\psi(P)$ for positive i, j. The rest of the combinations can be efficiently retrieved using conditional negations and conditional applications of ψ.

This compression trick not only halve the amount of precomputation needed, but also halves the time needed to do a linear pass through the table in the main loop. With new specialized group law formulas for the precomputation (see Section 5), the 2DT algorithm is able to compete for the constant-time scalar multiplication speed record (see Section 7). The 2DT variant is presented in Algorithm 2. For $w = 2$, the only difference is that a complete formula must be used for $2Q + P_1$ at $i = 1$ as well.

Algorithm 2: Constant-time 2DT scalar multiplication

```
   Input: $P \in \mathcal{G}$ in $\lambda$-affine coordinates, $k \in [1, r-1]$, window size $w > 2$
   Output: $kP$ in $\lambda$-affine coordinates
1  Decompose $k$ into odd subscalars $k_1, k_2$ using Algorithm 3.
2  $\ell \leftarrow \lceil \frac{m+1}{w-1} \rceil$
3  Compute width-$w$ length-$\ell$ odd signed regular recoding $\overline{k}_1, \overline{k}_2$ of $k_1, k_2$.
4  Compute $T[i, j] = (2i+1)P + (2j+1)\psi(P)$ for all odd $i, j \in \{0, \ldots, 2^{w-2} - 1\}$.
5  Convert $T$ to $\lambda$-affine coordinates using a simultaneous inversion.
6  Perform a linear pass over $T$ to recover $P_{\ell-1} = \overline{k}_{1,\ell-1}P + \overline{k}_{2,l-1}\psi(P)$
7  $Q \leftarrow P_{\ell-1}$
8  for $i$ from $\ell - 2$ downto 1 do
9      $Q \leftarrow 2^{w-2}Q$
10     Perform a linear pass over $T$ to recover $P_i = \overline{k}_{1,i}P + \overline{k}_{2,i}\psi(P)$
11     $Q \leftarrow 2Q + P_i$
12 Repeat the steps for $i = 0$, but use a complete formula for $2Q + P_0$.
13 Convert $Q$ to $\lambda$-affine coordinates.
14 return $Q$;
```

3.2 Proof of exception-free scalar multiplication

We will now prove that the scalar multiplication algorithms presented here and in [31] (with a minor modification) is correct on all valid inputs. The core issue is that the underlying λ-projective group law formulas from [31] are not complete, meaning that they output the wrong result in some corner cases. Without these exceptions, correctness would be trivial.

One could explicitly handle these exceptions in constant time using complete formulas, but this would come at a high performance cost. Here, we prove that exceptional cases can only occur in the last iteration(s) of the main loop. By using complete formulas at the very end, correctness is ensured at only a minor performance penalty.

For clarity, the proof will be tailored to the 2DT algorithm. However, it can be easily adapted to the 1DT algorithm. The proof can be seen as a two-dimensional extension of the argument from Proposition 1 in [5]. We will for now assume that the scalar decomposition produces subscalars of bit-length at most $m + 1$, and defer the discussion about how to achieve this to Section 4.

The proof crucially relies on the structure of the lattice discussed in [13,12] that emerge in the GLV method for scalar decomposition;

$$\mathcal{L} = \{(x, y) \in \mathbb{Z}^2 : x + y\mu \equiv 0 \pmod{r}\}.$$

Here r is the large prime order of $\#E'(\mathbb{F}_{q^2}) = hr$ and μ the eigenvalue of ψ restricted to $\mathcal{G}$. For our purposes, it is very useful to think about $\mathcal{L}$ as the lattice of decompositions of zero (as done in [9]).

Our proof requires that the norm of the shortest non-zero vector in $\mathcal{L}$ is at least $(q-1)/\sqrt{2}$. The structure of $\mathcal{L}$ is determined by the order of $E'(\mathbb{F}_{q^2})$. The bound required on the shortest norm might not be obtainable in general, so we focus on the subclass of GLS curves most relevant for cryptography. As the (affine) point $(0, 0)$ is always in $E'(\mathbb{F}_{q^2})$, $2|h$. To ensure that the discrete log problem in $\mathcal{G}$ is as hard as possible, the optimal choice is to pick a curve with $h = 2$, which is easy in practice. Restricting our proof to this subclass of GLS curves, we can give an explicit solution to the SVP in $\mathcal{L}$.

Lemma 1. *Let $q = 2^m$. Let E' be a binary GLS curve with $\#E'(\mathbb{F}_{q^2}) = 2r$ for an odd prime r. Let $E/\mathbb{F}_q$ be the curve such that E' over $\mathbb{F}_{q^2}$ is the quadratic twist of $E(\mathbb{F}_{q^2})$. Define*

$$v_1 = \Big(\frac{(q-1)+t}{2}, \frac{(q-1)-t}{2}\Big) \text{ and } v_2 = \Big(\frac{(q-1)-t}{2}, \frac{-(q-1)-t}{2}\Big),$$

where t is the trace of the q-th power Frobenius endomorphism on E. Then v_1, v_2 form an orthogonal basis for the lattice $\mathcal{L}$. $\|v_1\| = \|v_2\| = \sqrt{r} = \min_{v \in \mathcal{L}\setminus\{0\}} \|v\|$.

Proof. Smith gives in sections 6 and 4 of [35] the basis $v_1' = (q-1, -t)$, v_2 for $\mathcal{L}$. However it is not orthogonal and $\|v_1'\| = \sqrt{2r}$. The latter follows from $\#E'(\mathbb{F}_{q^2}) = (q-1)^2 + t^2$ (see Theorem 2 in [12]). We make the small adjustment of replacing v_1' with $v_1 = v_1' - v_2$. It can easily be checked that the basis v_1, v_2 is orthogonal and that $\|v_1\| = \|v_2\| = \sqrt{r}$. □

For the sake of proving scalar multiplication exception-free, the importance of Lemma 1 is showing that the minimal norm in $\mathcal{L}$ is large. This is what enables the subsequent proof to succeed. Note that we also require m to be prime, which is needed for security reasons anyways.

Theorem 1. *Let the notation be as in Lemma 1. Let $m > 4$ be prime and $2 \leq w \leq m$. Then Algorithm 2 is exception-free.*

Proof. Let us start by identifying the exceptional cases of the λ-projective formulas. The formula for $P + Q$ breaks down whenever $P = \pm Q$, $P = \infty$ or $Q = \infty$. The point ∞ does not have a λ-affine representation, so these last two cases are only a concern when the points are λ-projective. The $2P$ formula has no exceptional cases. Finally, the atomic formula for $2Q + P$ breaks down when $P = \pm 2Q$ or $Q = \infty$.

We will argue that all the exceptions that can occur in Algorithm 1 encode an element of $\mathcal{L}$. By this, we mean that they define some $z_1, z_2 \in \mathbb{Z}$ such that $z_1P + z_2\psi(P) = \infty$. Then $(z_1, z_2) \in \mathcal{L}$.

If $(z_1, z_2) \neq (0, 0)$, we can show that either $|z_1|$ or $|z_2|$ must be at least an m-bit integer. v_1, v_2 form an orthogonal basis of $\mathcal{L}$, and are both a solution to the SVP in $\mathcal{L}$ with norm $\sqrt{r}$. Using the Hasse bound we get that $\|v_1\|, \|v_2\| \geq (q-1)/\sqrt{2}$. Now assume for contradiction that $|z_1|, |z_2| < q/2$. Then

$$\|(z_1, z_2)\| = \sqrt{z_1^2 + z_2^2} \leq \sqrt{2\Big(\frac{q}{2} - 1\Big)^2} = \frac{q-2}{\sqrt{2}} < \|v_1\|, \|v_2\|.$$

This is a contradiction, since v_1, v_2 have minimal norm in $\mathcal{L} \setminus \{\mathbf{0}\}$. Thus, it must be the case that $|z_1| \geq q/2$ or $|z_2| \geq q/2$.

We now have all the tools needed to prove that no exceptions occur in Algorithm 2, and we will start with the precomputation stage. Assume that an exception did occur in the computation of $iP + j\psi(P)$ for some odd $i, j \in \{1, \ldots, 2^{w-1}-1\}$. $P, \psi(P) \neq \infty$, so there can only be an exception if $iP = sj\psi(P)$ for some $s \in \{\pm 1\}$. Then $(i, -sj) \in \mathcal{L} \setminus \{\mathbf{0}\}$. However, this is a contradiction, as neither $|i| = i$ nor $|-sj| = j$ are m-bit integers.

Next is the main loop. Let $Q_i = z_{1,i}P + z_{2,i}\psi(P)$ denote the value of Q after iteration i. No exception occurred in the precomputation stage, meaning Q is correctly initialized to $Q_{\ell-1} = \overline{k}_{1,\ell-1}P + \overline{k}_{2,\ell-1}\psi(P)$. Then at iteration i,

$$Q_i = (2^{w-1}z_{1,i+1} + \overline{k}_{1,i})P + (2^{w-1}z_{2,i+1} + \overline{k}_{2,i})\psi(P).$$

Observe that as long as no exceptions occur, we have the invariant that

$$0 < |z_{j,i+1}| \leq |z_{j,i}| \leq 2^{(\ell-i)(w-1)} - 1 \text{ for } j = 1, 2.$$

Assume the first exception occurs at iteration i. The $w-2$ doublings are exception-free, so the exception must have been caused by the computation of $2Q_{i+1} + P_i$. Q_{i+1} cannot have been ∞. This is because $2Q \neq \infty$ for any $Q \neq \infty$ and the incomplete $2Q+P$ formula does not output ∞ for any P, Q on the curve. Hence, the first exception must have occurred because $2^{w-1}Q_{i+1} = sP_i$ for an $s \in \{\pm 1\}$. This is equivalent to

$$(2^{w-1}z_{1,i+1} - s\overline{k}_{1,i})P + (2^{w-1}z_{2,i+1} - s\overline{k}_{2,i})\psi(P) = \infty$$

Define $z'_{j,i} = 2^{w-1}z_{j,i+1} - s\overline{k}_{j,i}$. Notice that $-s\overline{k}_{j,i}$ is a valid digit of the regular recoding. The invariants for $|z_{j,i}|$ also hold for $|z'_{j,i}|$. Hence, $(z'_{1,i}, z'_{2,i}) \in \mathcal{L} \setminus \{\mathbf{0}\}$.

Since m is prime, it holds for all $2 \leq w \leq m$ that

$$2^{(\lceil \frac{m}{w-1} \rceil - 1)(w-1)} - 1 \leq 2^{(\frac{m}{w-1} + \frac{w-2}{w-1} - 1)(w-1)} - 1 = 2^{m-1} - 1$$

This means that $|z'_{1,i}|$ and $|z'_{2,i}|$ are of at most $m-1$ bits while $i \geq \ell - \lceil \frac{m}{w-1} \rceil + 1$. The implication is that the first exception could not have occurred in these iterations, so we must have that $i \leq \ell - \lceil \frac{m}{w-1} \rceil$.

For $w = 2$, this means that $i \leq 1$. For all other w, this means that $i = 0$. But these are exactly the iterations that use a complete formula for the computation of $2Q_{i+1} + P_i$ for the respective values of w. Thus, it is impossible for the first exception to occur in these last iterations. The conclusion is that there can be no exception in the main loop. □

4 Scalar decomposition with parity and length guarantees

The GLV method for scalar decomposition needs a bit of care when required to run in constant-time while preserving length guarantees. The GLV method uses a reduced basis $\{u_1, u_2\}$ of some sublattice $\mathcal{L}'$ of $\mathcal{L}$ (see Section 3.2) to solve the CVP problem for $(k, 0)$ in $\mathcal{L}'$ using Babai rounding [13]. For a given basis, there exist unique constants $N, \alpha_1, \alpha_2 \in \mathbb{Z}$ such that

$$(k, 0) = \beta_1 u_1 - \beta_2 u_2,$$

where $\beta_i = \frac{\alpha_i}{N} k$. The subscalars k_1, k_2 are then computed as

$$(k_1, k_2) = (k, 0) - b_1 u_1 - b_2 u_2,$$

where $b_i = \lceil \beta_i \rfloor$. The magnitude of the subscalars can then be bounded by some expression depending on the norm of the basis vectors.

The issue for constant-time implementations is the computation of the b_i's. Ideally we would compute them using divisions, but unfortunately divisions do not run in constant time in most processors [1].

The standard solution was first introduced in [4] and further analyzed in [9]. The idea is to approximate the computation of the b_i's using integer divisions by powers of 2, which can be implemented in constant-time using shifts. Choose some integer d such that $k < 2^d$, and precompute the constants $c_i = \lfloor \frac{\alpha_i}{N} 2^d \rceil$. Then at runtime compute b_i as $b_i' = \lfloor \frac{c_i}{2^d} k \rfloor$.

This approach introduces rounding errors. It was shown in Lemma 1 of [9] that b_i' will either be b_i or incorrectly rounded down to $b_i - 1$. This does not affect the correctness of the decompositon. However, it does negatively impact the bounds on $|k_1|, |k_2|$. If the bounds become too loose, we might need more iterations of the main loop of the scalar multiplication to ensure correctness.

We present Algorithm 3 for constant time scalar decomposition using the optimal basis from Lemma 1. In Lemma 2 we prove that it outputs subscalars of at most $m+1$ bits. Without rounding errors it would be m, but the extra bit does not affect the number of iterations of the main loop (see Corollary 1).

In fact, not handling rounding errors leads to more efficient scalar multiplication. Using the fact that the rounding errors are one-sided, we introduce a new trick to ensure that the k_1, k_2 are always odd, without affecting the length guarantees. The two point additions needed to correct for even subscalars (see Subsection 2.2) are no longer needed, leading to a simpler and faster algorithm. Note that this optimization applies to both 1DT and 2DT scalar multiplication.

[1] See https://www.bearssl.org/constanttime.html.

Algorithm 3: Constant-time fixed-parity scalar decomposition for binary GLS curves with $h = 2$

```
    Input: $k \in [1, r-1]$
    Consts.: $N = \#E'(\mathbb{F}_{q^2})$, $d = \lceil \frac{2m}{W} \rceil \cdot W$, for $W$ the machine word size.
        $c_i = \lfloor \frac{\alpha_i}{N} 2^d \rceil$ for $i = 1, 2$ and $\alpha_1 = q - 1 + t$, $\alpha_2 = q - 1 - t$.
    Output: Odd $k_1, k_2$ such that $k_1 + k_2\mu \equiv k \pmod r$
1   $b_i \leftarrow c_i k \gg d$ for $i = 1, 2$ and
2   $(k_1, k_2) \leftarrow (k, 0) - b_1 v_1 - b_2 v_2$
3   if $\alpha_1 \equiv 0 \pmod 4$ then $(u_1, u_2) \leftarrow (v_2, v_1)$
4   else $(u_1, u_2) \leftarrow (v_1, v_2)$
5   $p_i \leftarrow k_i + 1 \bmod 2$ for $i = 1, 2$
6   $(k_1, k_2) \leftarrow (k_1, k_2) - p_1 u_1 - p_2 u_2$
7   return $k_1$, $k_2$
```

Lemma 2. *Let the notation be as in Lemma 1 and assume that $h = 2$ and $m > 4$. Algorithm 3 on input $k \in [1, r-1]$ outputs a valid decomposition k_1, k_2. The subscalars are odd and $|k_1|, |k_2| < 2q$.*

Proof. Let k_1, k_2 be the output of the GLV method on input k and let k_1', k_2' the output of Algorithm 3. We start with correctness. Per definition, $(k, 0) + \mathcal{L} \in \mathbb{Z}^2/\mathcal{L}$ is the set of valid decompositions of k. Algorithm 2 produces (k_1', k_2') by adding integer multiples of v_1 and v_2 to $(k, 0)$. Hence, $(k_1', k_2') \in (k, 0) + \mathcal{L}$.

Next, let's bound the magnitude of the subscalars. The basis vectors are orthogonal with norm $\sqrt{r}$. By the same argument as in Lemma 3 of [12] it follows that $\|(k_1, k_2)\| \leq \sqrt{r/2}$. To make the analysis independent of r, we can upperbound it as $r \leq (q+1)^2/2$ using the Hasse bound. Then $\|v_1\|, \|v_2\| \leq (q+1)/\sqrt{2}$ and $\|(k_1, k_1)\| \leq (q+1)/2$.

It can be easily verified that α_1, α_2, N are specified such that $(k, 0) = \beta_1 v_1 + \beta_2 v_2$. Since $k < r \leq (q+1)^2/2 \leq q^2 \leq 2^d$, it follows from Lemma 1 of [9] that b_i' is either b_i or incorrectly rounded down to $b_i - 1$.

Let r_i be the bit that is 1 if such a rounding error occurred when computing b_i'. Let s_i be the bit that is 1 if v_i was subtracted from (k_1, k_2) at line 8. Then using the triangle inequality, we can derive the bound on the subscalars.

$$\begin{aligned}
|k_1'|, |k_2'| &\leq \|(k_1', k_2')\| \\
&= \left\| \sum_{i=1}^{2} (\beta_i - (b_i - r_i + s_i)) v_i \right\| \\
&\leq \left\| \sum_{i=1}^{2} (\beta_i - b_i) v_i \right\| + \|v_1\| + \|v_2\| \\
&\leq \left(\frac{q+1}{2} \right) + 2\left(\frac{q+1}{\sqrt{2}} \right) \\
&< 2q && \text{(Assuming } m > 4\text{)}
\end{aligned}$$

Finally, we will show that k_1', k_2' are odd. The proof of Lemma 1 establishes that t is odd. $\frac{(q-1)+t}{2} = \frac{(q-1)-t}{2} + t$, meaning exactly one of the the coordinates of v_1 are odd. By symmetry, only the other coordinate of v_2 is odd. Because $\alpha_1 = 2\left(\frac{(q-1)+t}{2}\right)$, $\alpha_1 \equiv 0 \pmod 4$ exactly when the 1st coordinate of v_2 is odd. Then u_i is the basis vector with the odd i-th coordinate. Subtracting (k_1', k_2') by u_i flips the parity of k_i' but leaves the parity of the other subscalar unchanged. $p_i = 1$ exactly when k_i is even, meaning that the subscalars output are odd. □

Corollary 1. *Let $m > 4$ be a prime number. For any window size $2 < w \leq m$, the number of digits needed to recode the subscalars output by Algorithm 3 is the same as one would need for the subscalars output by the GLV method with no rounding errors. For $w = 2$, one more digit is required.*

5 New formulas for faster precomputation

The 2DT scalar multiplication variant represents a different strategy for utilizing precomputation. The table grows quadratically faster than its 1DT counterpart, so reducing the cost of its precomputation is crucial. For both the 1DT and 2DT variants, we present more efficient strategies for the precomputation stage. The 2DT precomputation (Algorithm 5) uses the 1DT precomputation (Algorithm 4) as a subroutine, which makes a case for the fairness of our optimization efforts.

The precomputation algorithms depend on several new atomic group law formulas. Compared to doing the operations using the existing formulas from [31], they provide a significant saving in the number of field multiplications and squarings needed. Because these formulas are derived by combining the original group law formulas, they do not introduce additional exceptions. The new formulas that are nontrivial to derive are included in Appendix B. An overview of the cost of the specialized λ-projective group law formulas is given in Table 1.

It follows from the same argument as in Theorem 1 that the new precomputation algorithms are exception-free. At any step we compute $iP + j\psi(P)$ for small coefficients i, j, where at least one of them are nonzero. Then $(i, \pm j) \notin \mathcal{L}$, meaning that there cannot be any exceptions.

Algorithm 4: Precomputation-1D

```
   Input: $P \in \mathcal{G}$ in $\lambda$-affine coordinates, window size $w > 2$.
   Output: Table $T$ of size $2^{w-2}$ with $T[i] = (2i+1)P$ in $\lambda$-affine coordinates.
1  $T[0] \leftarrow P$
2  $T[1] \leftarrow 3P$
3  for $i$ from $0$ to $2^{w-3} - 2$ do
4      $T[2i+3], T[2i+2] \leftarrow 2T[i+1] \pm P$
5  Convert $T$ to $\lambda$-affine coordinates using simultaneous inversion.
6  return $T$
```

Algorithm 5: Precomputation-2D

```
Input: $P \in \mathcal{G}$ in $\lambda$-affine coordinates, window size $w > 2$.
Output: Table $T$ of size $2^{w-2} \times 2^{w-2}$ with $T[i, j] = (2i + 1)P + (2j + 1)\psi(P)$
        in $\lambda$-affine coordinates.
1  $R \leftarrow$ Precomputation-1D($P$, $w$)
2  for $i$ from 0 to $2^{w-2} - 1$ do
3      $T[i, i] \leftarrow R[i] + \psi(R[i])$
4  for $j$ from 1 to $2^{w-2} - 1$ do
5      $Q \leftarrow \psi(R[j])$
6      for $i$ from 0 to $j - 1$ do
7          $T[i, j], T[j, i] \leftarrow R[i] \pm Q$
8          $T[j, i] \leftarrow \psi(T[j, i])$
9  Convert $T$ to $\lambda$-affine coordinates using simultaneous inversion.
10 return $T$
```

Table 1. Cost of the λ-projective group law formulas with respect to the number of multiplications, multiplications by curve coefficients a and b and squarings $(\tilde{m}, \tilde{m_a}, \tilde{m_b}, \tilde{s})$ over the extension field $\mathbb{F}_{q^2}$. For the mixed point representations, Q is λ-projective while P, P_1 and P_2 are λ-affine. The formulas that have not been derived or that provided insignificant speedups are marked with '-'.

Op.\Rep.	**Projective**	**Mixed**	**Affine**
$2P$	$4\tilde{m} + \tilde{m_a} + 4\tilde{s}/3\tilde{m} + 4\tilde{m_a} + \tilde{m_b} + 4\tilde{s}$	-	$\tilde{m} + 3\tilde{s}$
$3P$	-	-	$4\tilde{m} + \tilde{m_a} + 4\tilde{s}$
$P + Q$	$11\tilde{m} + 2\tilde{s}$	$8\tilde{m} + 2\tilde{s}$	$5\tilde{m} + 2\tilde{s}$
$P \pm Q$	-	$12\tilde{m} + 5\tilde{s}$	$6\tilde{m} + 4\tilde{s}$
2Q+P	-	$10\tilde{m} + \tilde{m_a} + 6\tilde{s}$	-
$2Q + P_1 + P_2$	-	$17\tilde{m} + \tilde{m_a} + 8\tilde{s}$	-
$P + \psi(P)$	-	-	$3.5\tilde{m} + 1.5\tilde{s}$

6 Binary field arithmetic for Arm

This section details our Arm implementation of the GLS254 curve. The focus will be on the field arithmetic. It was implemented specifically for the platform, relying heavily on 128-bit Arm Neon vector instructions to achieve high performance. The rest of the curve implementation is almost exclusively written in C, and therefore does not differ much from the Intel implementation from [31].

Specifically, our implementation targets Armv8 AArch64, which introduces some new useful instructions for cryptographic implementations. In particular, we take advantage of the new `PMULL` vector instruction for 64-bit binary polynomial multiplication, a direct analogue of `PCLMULQDQ` for Intel. For convenience of implementation, we use C intrinsics for the Arm Neon vector instructions.

This section first details the implementation of the base field $\mathbb{F}_q$ with $q = 2^m$ and $m = 127$, then how we implement the quadratic extension field $\mathbb{F}_{q^2}$ on top.

6.1 Arithmetic in the base field $\mathbb{F}_q$

Representation of elements. One benefit of the choice of field, is that the bit vector representation of $a \in \mathbb{F}_q$ is 127 bits long, meaning that we can fit it in a single 128-bit Neon vector register. We denote $a[0], a[1]$ as respectively the least significant and most significant word of the 128-bit register that stores $a \in \mathbb{F}_{2^{127}}$. $a[0]$ stores the bit vector for the terms z^0 to z^{63}, $a[1]$ terms z^{64} to z^{126}. We will sometimes use the notation $a = \{a[0], a[1]\}$ to show the contents of the register.

An efficiency issue for Arm AArch64, compared to AArch32, is that it cannot reference the upper word $a[1]$ of a 128-bit register as a separate 64-bit register [15]. Instead, one needs to use the Arm Neon instruction `EXT`. It takes two registers a, b and outputs $\{a[1], b[0]\}$. The lower half of this output can then be referenced for further computation. Table 2 gives an overview of all the Neon instructions that we used for our implementation.

Table 2. Arm Neon 128-bit vector instructions used. The first 128-bit operand is denoted a, the second b. The output is also stored in a 128-bit register.

Symbol	Description	Neon Instruction
$\oplus, \wedge$	Bitwise XOR, AND	`EOR`, `AND`
$\ll_{128}, \gg_{128}$	Logical shift (no carry between words)	`SHL`, `SHR`
pmull_bot	Multiply binary polynomials $a[0], b[0]$	`PMULL`
pmull_top	Multiply binary polynomials $a[1], b[1]$	`PMULL2`
extract	Outputs $\{a[1], b[0]\}$	`EXT`

Polynomial multiplication. The polynomial multiplication algorithm takes as input two binary polynomials $a, b \in \mathbb{F}_q$ and outputs their polynomial product c. The degree of c can be up to twice the degree of the operands. Hence it must be stored in two 128-bit vector registers c_0, c_1, where c_0 stores the lower half.

For polynomial multiplication we use the Arm Neon implementation from [15]. It efficiently performs 128-bit polynomial multiplication using the new `PMULL` instructions. While they managed to implement it using 3 multiplications with the Karatsuba algorithm on AArch32, the high number of `EXT`s this would require on AArch64 meant that they instead opted for an algorithm with an extra `PMULL`.

Polynomial squaring and field multi-squaring. Polynomial squaring of an $a \in \mathbb{F}_q$ can be trivially implemented as $c_0 \leftarrow \text{pmull\_bot}(a, a)$, $c_1 \leftarrow \text{pmull\_top}(a, a)$.

For multi-squaring in settings where it does not need to be computed in constant time, we implemented the technique from [1,6]. It uses lookup tables that are precomputed offline to compute the reduced result of a^{2^k}. However, for smaller Arm processors like the Cortex-A55, this method only outperforms the naive loop implementation for $k > 12$. This is a lot higher than the threshold of $k > 5$ for Intel [31]. It is an example of the higher cost of memory access on smaller devices, which results in a lower yield for precomputation strategies.

Modular reduction. To compute a field multiplication or squaring, the result of the polynomial algorithm must be reduced modulo $f(z)$. We here present novel algorithms for efficient modular reduction, using exclusively Arm Neon vector instructions. Like the polynomial multiplication algorithm from [15], they attempt to minimize the number of accesses to the top half of the 128-bit registers, each of which incurs the cost of an `EXT`.

For reducing the result of a polynomial multiplication, we use Algorithm 6. It implements the lazy reduction technique from [30]. Instead of reducing $f(z)$, we reduce by the redundant trinomial $z \cdot f(z) = z^{128} + z^{64} + z$. Reductions by $zf(z)$ are roughly 40% faster than proper reductions by $zf(z)$. The result can have degree up to 127 instead of 126, but as the result still fits in a 128-bit register, this makes no difference. As $(c \bmod zf(z)) \bmod f(z) = c \bmod f(z)$, one can easily recover the properly reduced result from the output of the lazy reduction. This is done by conditionally adding $z^{63} + 1$ to it when bit 127 is set.

Algorithm 6: Lazy reduction by $z \cdot f(z) = z^{128} + z^{64} + z$

```
Input: 254-bit polynomial stored in two 128-bit registers $c_0, c_1$.
Output: 128-bit register $a$ storing $c(z) \bmod f(z)$.
Temps.: Uses 128-bit registers $t_0, t_1, t_2$.
1  $t_0[0] \leftarrow 0$
2  $t_1[0] \leftarrow c_1[0] \gg 63$
3  $t_0 \leftarrow \text{extract}(c_1, t_0)$
4  $t_2[0] \leftarrow c_1[0] \oplus t_0[0]$
5  $t_1[0] \leftarrow t_1[0] \oplus t_2[0]$
6  $t_0 \leftarrow \text{extract}(t_0, t_1)$
7  $a \leftarrow c_0 \oplus t_0$
8  $t_2 \leftarrow t_2 \ll_{128} 1$
9  $a \leftarrow a \oplus t_2$
10 return a
```

It is possible to reduce a squaring slightly faster, exploiting the fact that the result only has bits set at even positions. Thus, we can remove the logic from Algorithm 6 that handles the carry of bit 191 from the left shift by 1. Concretely, this means removing lines 2 and 5, and replacing t_2 by t_1.

Field inversion. Field inversion is done in the same way as described in [31], using the Itoh-Tsujii algorithm [19]. We generated our addition chain for $m-1 = 126$ using McLoughlin's *addchain* library [29].

$$1 \to 2 \to 3 \to 6 \to 12 \to 24 \to 30 \to 48 \to 96 \to 126$$

The cost of a field inversion is therefore $m - 1$ squarings and 9 multiplications. The steps after 30 involve multi-squarings with $k > 12$. When the inversion does not have to be in constant time, these steps can then be sped up using the table-based multi-squaring approach.

6.2 Arithmetic in the extension field $\mathbb{F}_{q^2}$

The elements of $\mathbb{F}_{q^2}$ can be represented as polynomials $a_1 u + a_0$, with coefficients $a_0, a_1 \in \mathbb{F}_q$. Therefore, we need two 128-bit registers to represent them. The extension field arithmetic can be implemented from the base field arithmetic, using the identities presented in [31].

In [30], Oliveira et al present an algorithm for simultaneously reducing both coefficients of an element in $\mathbb{F}_{q^2}$ at the cost of only a single base field reduction. We have included the Arm Neon implementation in Algorithm 7.

Algorithm 7: Lazy simultaneous reduction by $zf(z) = z^{128} + z^{64} + z$ for coordinate-wise reduction in $\mathbb{F}_{q^2}$ (Oliveira et al. [30])

```
Input: Unreduced polynomial stored in interleaved 128-bit registers
       $c_0, c_1, c_2, c_3$.
Output: 128-bit register $a$ storing $c(z)$ mod $zf(z)$.
Temps.: Uses 128-bit register $t$.
1 $c_2 \leftarrow c_2 \oplus c_3$
2 $t \leftarrow c_3 \lll_{128} 1$
3 $c_1 \leftarrow c_1 \oplus t$
4 $c_1 \leftarrow c_1 \oplus c_2$
5 $t \leftarrow c_2 \ggg_{128} 63$
6 $c_1 \leftarrow c_1 \oplus t$
7 $t \leftarrow t \lll_{128} c_2$
8 $c_0 \leftarrow c_0 \oplus t$
9 return $c_0, c_1$
```

However, the reduction algorithm requires the field elements to be kept in an interleaved representation. For an $a \in \mathbb{F}_{q^2}$, let a_0, a_1 be the 128-bit registers storing each of its coefficients. Then the interleaved representation of a is

$$a'_0 = \{a_0[0], a_1[0]\},\ \ a'_1 = \{a_0[1], a_1[1]\}. \tag{3}$$

Note that a'_0, a'_1 store a_0 in the lower half and a_1 in the upper half. The larger input to the reduction algorithm must also be in an interleaved representation. Let c_0, c_1, c_2, c_3 be the non-interleaved 128-bit registers storing the result of a polynomial multiplication or squaring. Because this result is computed using the identites from [31], the result is already reduced as a polynomial in u, but has coefficients that need further reduction in $\mathbb{F}_q$. Then c_0, c_1 store the unreduced constant coefficient and c_2, c_3 the other. The interleaved representation c'_0, c'_1, c'_2, c'_3 of these unreduced coefficients continue the pattern from (3). c'_0, c'_1 are c_0, c_2 interleaved, and c'_2, c'_3 are c_1, c_3 interleaved.

In order to reap the benefits of the reduction algorithm, we had to implement the $\mathbb{F}_{q^2}$ arithmetic directly in the interleaved representation. To do this, we manually merged and interleaved the base field algorithms to compute the identities from [31]. The only exception is inversion, where a standard base field inversion is used as a subroutine, which again uses all the arithmetic operations discusses in the previous section. While the abstraction between base field and extension field is somewhat broken for the sake of performance, the base field implementation is still the crucial foundation.

7 Results and discussion

Our implementations, together with SAGE scripts for verification and operation counts, can be found at https://github.com/dfaranha/gls254.

7.1 Operation counts for binary GLS scalar multiplication

Table 3. The cost of the scalar multiplications with respect to the number of inversions, multiplications and squarings ($\tilde{i}$, $\tilde{m}$, $\tilde{s}$) over $\mathbb{F}_{q^2}$. The total cost in field multiplications are estimated using $\tilde{i}_{\text{non-ct}} = 18\tilde{m}$, $\tilde{i}_{\text{ct}} = 27\tilde{m}$ and $\tilde{s} = 0.4\tilde{m}$ based on our benchmarks, and rounded to the nearest integer. "prev" denotes the cost for previous work.

Variant\\w	**3**	**4**	**5**	**6**
Precomp.				
1DT (prev)	$\tilde{i} + 12\tilde{m} + 6\tilde{s}$	$\tilde{i} + 38\tilde{m} + 14\tilde{s}$	$\tilde{i} + 90\tilde{m} + 30\tilde{s}$	$\tilde{i} + 194\tilde{m} + 62\tilde{s}$
1DT	$\tilde{i} + 6\tilde{m} + 4\tilde{s}$	$\tilde{i} + 31\tilde{m} + 13\tilde{s}$	$\tilde{i} + 81\tilde{m} + 31\tilde{s}$	$\tilde{i} + 181\tilde{m} + 67\tilde{s}$
2DT	$2\tilde{i} + 36\tilde{m} + 11\tilde{s}$	$2\tilde{i} + 158\tilde{m} + 43\tilde{s}$	$2\tilde{i} + 594\tilde{m} + 155\tilde{s}$	$2\tilde{i} + 2234\tilde{m} + 571\tilde{s}$
Main loop				
1DT (both)	$1273\tilde{m} + 764\tilde{s}$	$979\tilde{m} + 680\tilde{s}$	$819\tilde{m} + 628\tilde{s}$	$738\tilde{m} + 608\tilde{s}$
2DT	$823\tilde{m} + 633\tilde{s}$	$676\tilde{m} + 591\tilde{s}$	$593\tilde{m} + 561\tilde{s}$	$554\tilde{m} + 553\tilde{s}$
Total				
1DT (prev)	$2\tilde{i} + 1309\tilde{m} + 780\tilde{s}$	$2\tilde{i} + 1041\tilde{m} + 704\tilde{s}$	$2\tilde{i} + 933\tilde{m} + 668\tilde{s}$	$2\tilde{i} + 956\tilde{m} + 680\tilde{s}$
1DT	$2\tilde{i} + 1281\tilde{m} + 768\tilde{s}$	$2\tilde{i} + 1012\tilde{m} + 693\tilde{s}$	$2\tilde{i} + 902\tilde{m} + 659\tilde{s}$	$2\tilde{i} + 921\tilde{m} + 675\tilde{s}$
2DT	$3\tilde{i} + 861\tilde{m} + 644\tilde{s}$	$3\tilde{i} + 839\tilde{m} + 634\tilde{s}$	$3\tilde{i} + 1189\tilde{m} + 716\tilde{s}$	$3\tilde{i} + 2790\tilde{m} + 1124\tilde{s}$
Est. mult.				
1DT (prev)	$1666\tilde{m}$	$1368\tilde{m}$	$1245\tilde{m}$	$1273\tilde{m}$
1DT	$1633\tilde{m}$	$1334\tilde{m}$	$1211\tilde{m}$	$1236\tilde{m}$
2DT	$1182\tilde{m}$	$1155\tilde{m}$	$1538\tilde{m}$	$3303\tilde{m}$

Throughout our work, we have used field operation counts as a measure of the complexity of the scalar multiplication variants. With an understanding of the relative cost of the operations, the count gives a platform-independent estimate of the relative performance of the algorithms. In particular, it guided our choice of window size. However, it crucially does not capture the space-time trade-offs of a particular architecture. For the variants discussed in this paper, which are precisely different strategies for how to use space, this trade-off has a significant impact. This will be apparent in the next subsection.

Table 3 gives an overview the operation counts for the variants. Additions and multiplications by the curve coefficients are ignored due to their insignificant impact on performance. We include the costs of the 1DT algorithm without the new formulas and scalar decomposition to highlight the impact of our contributions. For the sake of fairness, it has been modified to be exception-free in the same way as the others.

As expected, the 2DT algorithm spends more time on precomputation and less in the main loop. We see that the model predicts $w = 5$ to be the sweet spot for 1DT and $w = 4$ for 2DT. Notably, 2DT $w = 3$ is predicted to be faster than 1DT $w = 5$ using only half the amount of space.

For a simpler comparison, we estimate the total cost in terms of field multiplications. The relative cost of each operation will of course differ from processor to processor, so we tried to go for the middle-ground based on our benchmarks. With this simplification, the model predicts that 1DT with $w = 5$ should be 2.7% faster from our contributions. The 2DT approach with $w = 4$ is predicted to be 7.2% faster than 1DT in previous work with $w = 5$, and 4.6% faster than 1DT with $w = 5$ from this work. Non-constant-time field inversion is used to convert points from projective to affine in the precomputation table only, since it does not depend on the (secret) scalar.

7.2 Implementation timings

We start by describing our benchmarks for the Armv8 AArch64 implementation, written from scratch. We used the ODROID C4 single board computer, as we wanted a smaller device that could be representative for the majority of Arm devices. It comes with a Quad-Core Cortex-A55, which is considered a mid-range processor. We employ `clang` from LLVM 13 with optimization level `-O3`.

Table 4. Benchmarks (in clock cycles) of the field arithmetic on an Arm Cortex-A55 2.0 GHz. The cost of reduction is included in the cost of the multiplication and squaring. Base field reduction is mod $zf(z)$. Op/m_b, Op/$\widetilde{m}$ denotes the cost relative to respectively base/extension field multiplication.

Field op.	$\mathbb{F}_{2^{127}}$		$\mathbb{F}_{2^{254}}$	
	Cycles	Op/m_b	Cycles	Op/$\widetilde{m}$
Multiplication	35	1.00	68	1.00
Reduction	16	0.46	15	0.22
Squaring	18	0.51	26	0.38
Inversion (ct.)	1 716	49.03	1 815	26.69
(non-ct.)	1 165	33.29	1 228	18.06

The benchmarks for our field implementation are presented in Table 4. Notice that non-constant time inversions that use lookup tables are roughly 33% faster.

Table 5 presents our scalar multiplication timings in GLS254 and comparisons to related work. Compared to Intel, there are not a lot of efficient implementations specialized for Arm at the 128-bit security level. Four$\mathbb{Q}$ [9] is the

Table 5. Constant-time variable base scalar multiplication benchmarks that are mostly performed on an Arm Quad-Core Cortex-A55 2.0 GHz. Memory is measured in terms of the number of elliptic curve points stored in the online precomputed table.

Implementation	**Algorithm**	**Memory**	**Cycles**
Lenngren [24] (Cortex-A55)	Curve25519	0	157,182
Longa [27] (Cortex-A55)	Four$\mathbb{Q}$	8	191,184
Longa [27] (Cortex-A15)	Four$\mathbb{Q}$	8	132,000
This work (Cortex-A55)	GLS254 1DT $w = 5$	8	92,460
	GLS254 2DT $w = 3$	**4**	**86,525**
	GLS254 2DT $w = 4$	16	91,682

Table 6. Protected variable base scalar multiplication benchmarks for 64-bit Intel Core i7 4770 Haswell at 3.4GHz, and Core i7 7700 Kaby Lake at 3.6GHz, both with TurboBoost disabled. Memory is measured in terms of the number of elliptic curve points stored in the online precomputed table.

Implementation	**Algorithm**	**Memory**	**Cycles**
Longa et al. [9] (Haswell)	Four$\mathbb{Q}$	8	56,000
Longa et al. [9] (Kaby Lake)	Four$\mathbb{Q}$	8	47,052
Oliveira et al. [31] (Haswell)	GLS254 1DT $w = 5$	8	48,301
Oliveira et al. [30] (Skylake)	GLS254 1DT $w = 5$	8	38,044
This work (Haswell)	GLS254 1DT $w = 5$	8	45,966
	GLS254 2DT $w = 3$	**4**	**45,253**
	GLS254 2DT $w = 4$	16	47,184
This work (Kaby Lake)	GLS254 1DT $w = 5$	8	36,480
	GLS254 2DT $w = 3$	**4**	**35,739**
	GLS254 2DT $w = 4$	16	38,076

closest competitor on Intel, and they also provide specialized implementations for Arm [27]. We benchmarked their Armv8 AArch64 implementation on our machine and included their Armv7 timings from [27] for the sake of fairness. A notable outlier is Lenngren's implementation for Curve2559, which is a much closer competitor on Arm than any Curve25519 implementation on Intel.

As the first GLS254 implementation for Armv8, we are able to claim the constant-time scalar multiplication speed record by 34.5% in comparison to the previous state of the art. Contrary to the operation counts in Table 3, it is the 2DT $w = 3$ algorithm that is the superior variant. We do not compare against [26] due to the radically different choices of target platform.

For our Intel implementation, we extended the AVX-accelerated code from [34] with the new formulas and 2DT variant. Due to space limitations, we report timings for field arithmetic in Appendix C. The scalar multiplication benchmarks are presented in Table 6. We benchmarked our code on an older Core i7 4770 Haswell processor, and a Core i7 7700 Kaby Lake as the closest to the Skylake in [30]; both using `clang` from LLVM 13 and optimization level `-O3`. For 1DT

$w = 5$, we achieve small speedups of 4.8% in Haswell and 4.1% for Skylake over the previous state of the art. The 2DT $w = 3$ variant achieves a further speedup of 2%. Surprisingly, 2DT $w = 4$ performs relatively poorly due to expensive conditional moves within the longer linear pass. The cumulative speedup over previous work is around 6% on both machines. In comparison to Four$\mathbb{Q}$, our timings are 24% faster and set a new speed record for constant-time scalar multiplication in Intel processors.

Removing the linear pass for the sake of experimentation, 2DT $w = 4$ outperforms the other variants on all the platforms benchmarked, as predicted by the operation counts. The relative cost of the linear pass seems to be the determining factor for how much of a speedup the new approach yields in practice. In general, the linear pass incurs a relatively high cost, favoring solutions that minimize the size of the lookup table. Hence, we see that the 2DT $w = 3$ variant claims the speed record across the board, using only half as much space as the previous record holder 1DT $w = 5$.

Acknowledgements

We thank Aurore Guillevic for discussions about preliminary results of this work, and Jonas Tambjerg Morthorst for help on the early Arm implementation.

References

1. Ahmadi, O., Hankerson, D., Rodríguez-Henríquez, F.: Parallel Formulations of Scalar Multiplication on Koblitz Curves. J. UCS. **14**(3), 481–504 (2008)
2. Bernstein, D.J.: Curve25519: New Diffie-Hellman Speed Records. In: Public Key Cryptography. LNCS, vol. 3958, pp. 207–228. Springer (2006)
3. Bernstein, D.J., Lange, T.: SafeCurves: choosing safe curves for elliptic-curve cryptography. https://safecurves.cr.yp.to/
4. Bos, J.W., Costello, C., Hisil, H., Lauter, K.E.: High-Performance Scalar Multiplication Using 8-Dimensional GLV/GLS Decomposition. In: CHES. LNCS, vol. 8086, pp. 331–348. Springer (2013)
5. Bos, J.W., Costello, C., Longa, P., Naehrig, M.: Selecting elliptic curves for cryptography: an efficiency and security analysis. J. Cryptogr. Eng. **6**(4), 259–286 (2016)
6. Bos, J.W., Kleinjung, T., Niederhagen, R., Schwabe, P.: ECC2K-130 on cell cpus. In: AFRICACRYPT. LNCS, vol. 6055, pp. 225–242. Springer (2010)
7. Chi, J., Oliveira, T.: Attacking a Binary GLS Elliptic Curve with Magma. In: LATINCRYPT. LNCS, vol. 9230, pp. 308–326. Springer (2015)
8. Ciet, M., Lange, T., Sica, F., Quisquater, J.: Improved Algorithms for Efficient Arithmetic on Elliptic Curves Using Fast Endomorphisms. In: EUROCRYPT. LNCS, vol. 2656, pp. 388–400. Springer (2003)
9. Costello, C., Longa, P.: Four$\mathbb{Q}$: four-dimensional decompositions on a $\mathbb{Q}$-curve over the Mersenne prime. IACR Cryptol. ePrint Arch. p. 565 (2015)
10. Faz-Hernández, A., Longa, P., Sánchez, A.H.: Efficient and secure algorithms for GLV-based scalar multiplication and their implementation on GLV-GLS curves (extended version). J. Cryptogr. Eng. **5**(1), 31–52 (2015)
11. Galbraith, S.D., Gaudry, P.: Recent progress on the elliptic curve discrete logarithm problem. Des. Codes Cryptogr. **78**(1), 51–72 (2016)

12. Galbraith, S.D., Lin, X., Scott, M.: Endomorphisms for Faster Elliptic Curve Cryptography on a Large Class of Curves. In: EUROCRYPT. LNCS, vol. 5479, pp. 518–535. Springer (2009)
13. Gallant, R.P., Lambert, R.J., Vanstone, S.A.: Faster Point Multiplication on Elliptic Curves with Efficient Endomorphisms. In: CRYPTO. LNCS, vol. 2139, pp. 190–200. Springer (2001)
14. Gaudry, P., Hess, F., Smart, N.P.: Constructive and Destructive Facets of Weil Descent on Elliptic Curves. J. Cryptol. **15**(1), 19–46 (2002)
15. Gouvêa, C.P.L., López-Hernández, J.C.: Implementing GCM on ARMv8. In: CT-RSA. LNCS, vol. 9048, pp. 167–180. Springer (2015)
16. Hankerson, D., Karabina, K., Menezes, A.: Analyzing the Galbraith-Lin-Scott Point Multiplication Method for Elliptic Curves over Binary Fields. IEEE Trans. Computers **58**(10), 1411–1420 (2009)
17. Hankerson, D., Vanstone, S., Menezes, A.: Guide to Elliptic Curve Cryptography. Springer-Verlag (2004)
18. Hess, F.: Generalising the GHS Attack on the Elliptic Curve Discrete Logarithm Problem. LMS Journal of Computation and Mathematics **7**, 167–192 (2004)
19. Itoh, T., Tsujii, S.: A Fast Algorithm for Computing Multiplicative Inverses in GF(2^m) Using Normal Bases. Inf. Comput. **78**(3), 171–177 (1988)
20. Joye, M., Tunstall, M.: Exponent Recoding and Regular Exponentiation Algorithms. In: AFRICACRYPT. LNCS, vol. 5580, pp. 334–349. Springer (2009)
21. Kales, D., Rechberger, C., Schneider, T., Senker, M., Weinert, C.: Mobile Private Contact Discovery at Scale. In: USENIX Security Symposium. pp. 1447–1464. USENIX Association (2019)
22. Klaus Pommerening: Quadratic equations in finite fields of characteristic 2. https://www.staff.uni-mainz.de/pommeren/MathMisc/QuGlChar2.pdf (2012)
23. Koblitz, A.H., Koblitz, N., Menezes, A.: Elliptic curve cryptography: The serpentine course of a paradigm shift. Journal of Number theory **131**(5), 781–814 (2011)
24. Lenngren, E.: AArch64 optimized implementation for X25519. https://github.com/Emill/X25519-AArch64
25. Lidl, R., Niederreiter, H.: Finite fields. Cambridge University Press (1997)
26. Liu, Z., Longa, P., Pereira, G.C.C.F., Reparaz, O., Seo, H.: Four \mathbb Q on embedded devices with strong countermeasures against side-channel attacks. In: CHES. LNCS, vol. 10529, pp. 665–686. Springer (2017)
27. Longa, P.: Four$\mathbb{Q}$NEON: Faster Elliptic Curve Scalar Multiplications on ARM Processors. In: SAC. LNCS, vol. 10532, pp. 501–519. Springer (2016)
28. López-Hernández, J.C., Dahab, R.: New Point Compression Algorithms for Binary Curves. In: ITW. pp. 126–130. IEEE (2006)
29. McLoughlin, M.B.: addchain: Cryptographic Addition Chain Generation in Go. Repository https://github.com/mmcloughlin/addchain (Oct 2021)
30. Oliveira, T., López-Hernández, J.C., Aranha, D.F., Rodríguez-Henríquez, F.: Improving the performance of the GLS254. Presentation at CHES 2016 Rump Session.
31. Oliveira, T., López-Hernández, J.C., Aranha, D.F., Rodríguez-Henríquez, F.: Two is the fastest prime: lambda coordinates for binary elliptic curves. J. Cryptogr. Eng. **4**(1), 3–17 (2014)
32. Oliveira, T., López-Hernández, J.C., Cervantes-Vázquez, D., Rodríguez-Henríquez, F.: Koblitz Curves over Quadratic Fields. J. Cryptol. **32**(3), 867–894 (2019)
33. Renes, J., Costello, C., Batina, L.: Complete Addition Formulas for Prime Order Elliptic Curves. In: EUROCRYPT. LNCS, vol. 9665, pp. 403–428. Springer (2016)
34. Resende, A.C.D., Aranha, D.F.: Faster Unbalanced Private Set Intersection. In: Financial Cryptography. LNCS, vol. 10957, pp. 203–221. Springer (2018)

35. Smith, B.: Easy scalar decompositions for efficient scalar multiplication on elliptic curves and genus 2 Jacobians. CoRR **abs/1310.5250** (2013)

Appendix

A Point compression for binary GLS curves

As a last construction, we present a new point compression algorithm for binary GLS curves for points in λ-affine form. The best known point compression algorithm for elliptic curves over $\mathbb{F}_{2^n}$ is Algorithm 5 from [28]. It compresses an affine point (x, y) of $2n$ bits to n bits, and is the first to do so without needing an inversion. However, it requires $\mathrm{Tr}(a) = 1$ and n to be odd. The latter condition is a problem for binary GLS curves, since they are defined over $\mathbb{F}_{2^{2m}}$ for an odd prime m. Our new algorithm adapts the techniques of [28] to this setting.

First we need some notation. Let $E = \mathbb{F}_{q^m}$ be the finite field extension of $K = \mathbb{F}_q$. Then the field trace $\mathrm{Tr}_{E/K} : E \to K$ is defined as $\mathrm{Tr}_{E/K}(c) = \sum_{i=0}^{m-1} c^{q^i}$ [25]. For our purposes, we define $q = 2^m$, $\mathrm{Tr}' = \mathrm{Tr}_{\mathbb{F}_{q^2}/\mathbb{F}_2}$ and $\mathrm{Tr} = \mathrm{Tr}_{\mathbb{F}_q/\mathbb{F}_2}$.

The point decompression algorithm needs to solve a quadratic equation $\lambda^2 + \lambda = c$ in $\mathbb{F}_{q^2}$. The equation has a solution if and only if $\mathrm{Tr}'(c) = 0$ [22, p. 54]. If a solution exists, it can be efficiently found using the technique from [16] that was generalized in [31]. Given a solution $\hat{\lambda}$, the other solution is $\hat{\lambda} + 1$.

Our algorithm works for any point $P = (x, \lambda)$ in the subgroup S of large prime order r. The compression algorithm computes $C_P = x + \mathrm{lsb}(\lambda_0)u$ of m bits. Here $\mathrm{lsb} : \mathbb{F}_q \to \mathbb{F}_2$ is the function that on input $d = d_0 + ... + d_{q-1}z^{q-1} \in \mathbb{F}_q$ outputs d_0. P can then be recovered from C_P as follows.

Algorithm 8: Decompression algorithm

```
  Input: $C_P = x + \mathrm{lsb}(\lambda_0)u$
  Output: $P = (x,\ \lambda) \in \mathcal{G}$
1 $t \leftarrow \mathrm{Tr}(C_{P,1}) + 1$
2 $x \leftarrow c + tu$
3 Find solution $\lambda'$ for $\lambda^2 + \lambda = b/x^2 + x^2 + a$
4 if $t = \mathrm{lsb}(\lambda'_0)$ then
5     $\lambda \leftarrow \lambda'$
6 else
7     $\lambda \leftarrow \lambda' + 1$
8 return $(x,\ \lambda)$
```

Lemma 3. *Let $P = (x, \lambda) \in \mathcal{G}$. Then Algorithm 8 recovers P from C_P.*

Proof. We are going to need some properties of the trace function. Firstly, $\mathrm{Tr}_{E/K}$ is a linear transformation from E onto K, where E, K are viewed as vector spaces over K [22, p. 55]. The trace is also transitive, meaning for a finite extension F of E, $\mathrm{Tr}_{F/K}(c) = \mathrm{Tr}_{E/K}(\mathrm{Tr}_{F/E}(c))$ [22, p. 56]. For binary elliptic curves it holds that for all $P = (x, \lambda)$ of odd order, $\mathrm{Tr}'(x) = \mathrm{Tr}'(a)$ [17, p. 130]. Finally, because m is odd, $\mathrm{Tr}(d) = d$ for $d \in \mathbb{F}_2$.

From the transitivity of the trace function, we get that for $c = c_0 + c_1 u \in \mathbb{F}_{q^2}$ that $\mathrm{Tr}'(c) = \mathrm{Tr}(c + c^q) = \mathrm{Tr}((c_0 + c_1 u) + (c_0 + c_1 + c_1 u)) = \mathrm{Tr}(c_1)$. As binary GLS curves have $\mathrm{Tr}'(a) = 1$, it follows that $\mathrm{Tr}(x) = 1$ for all $P = (x, \lambda) \in S$. Then $t = \mathrm{Tr}(C_{P,1}) + 1 = (\mathrm{Tr}(x_1) + \mathrm{Tr}(\mathrm{lsb}(\lambda_0))) + \mathrm{Tr}(x_1) = \mathrm{lsb}(\lambda_0)$.

Next, we correctly recover x as $C_P + tu$. The λ-affine curve equation is $\lambda^2 x^2 + \lambda x^2 = x^4 + ax^2 + b$. Dividing both sides by x^2, we get a quadratic equation in standard form, which can be solved using the technique from [16]. Note that $x \neq 0$ when P is represented in λ-affine coordinates. Given the candidate solutions $\hat{\lambda}$ and $\hat{\lambda} + 1$, we recover the correct one from t. $\square$

B The new formulas used for precomputation and exception-free execution

Proposition 1. *Let $P = (x, \lambda)$ be a point on $E'_{a,b}(\mathbb{F}_{q^2})$ with $3P \neq \mathcal{O}$. Then $3P$ can be computed in λ-projective coordinates as follows.*

$$\begin{aligned}
T &= \lambda^2 + \lambda + a\\
A &= (x + T)^2 + T\\
X_{3P} &= x \cdot A^2\\
Z_{3P} &= A \cdot (A + T)\\
\Lambda_{3P} &= T^2 \cdot T + Z_{3P}(\lambda + 1)
\end{aligned}$$

Proposition 2. *Let $P = (x_P, \lambda_P)$ and $Q = (X_Q, \Lambda_Q, Z_Q)$ be points on $E'_{a,b}(\mathbb{F}_{q^2})$ with $P \neq \pm Q$. Then $P + Q$ and $P - Q$ can be simultaneously computed in λ-projective coordinates as follows.*

$$\begin{aligned}
A &= \lambda_P \cdot Z_Q + \Lambda_Q\\
B &= (x_P \cdot Z_Q + X_Q)^2\\
C &= (x_P \cdot Z_Q) \cdot X_Q\\
X_{P+Q} &= A^2 \cdot C\\
Z_{P+Q} &= A \cdot B \cdot Z_Q\\
\Lambda_{P+Q} &= (A \cdot X_Q + B)^2 + Z_{P+Q} \cdot (\lambda_P + 1)\\
X_{P-Q} &= X_{P+Q} + C \cdot Z_Q^2\\
Z_{P-Q} &= Z_{P+Q} + B \cdot Z_Q^2\\
\Lambda_{P-Q} &= \Lambda_{P+Q} + (X_Q \cdot Z_Q)^2 + (B \cdot Z_Q^2) \cdot (\lambda_P + 1)
\end{aligned}$$

Proposition 3. *Let $P = (X, \Lambda, Z)$ on $E'_{a,b}(\mathbb{F}_{q^2})$. For $c \in \mathbb{F}_{q^2}$, let c_0, c_1 denote its coefficients in $\mathbb{F}_q$ such that $c = c_0 + c_1 u$. Then $\psi(P)$ can be computed in λ-projective coordinates as follows.*

$$\begin{aligned}
X_{\psi(P)} &= X + X_1 \\
\Lambda_{\psi(P)} &= \Lambda + \Lambda_1 + Z_1 + Z_0 u \\
Z_{\psi(P)} &= Z + Z_1
\end{aligned}$$

Proposition 4. *Let $P = (x, \lambda)$ on $E'_{a,b}(\mathbb{F}_{q^2})$. For $c \in \mathbb{F}_{q^2}$, let c_0, c_1 denote its coefficients in $\mathbb{F}_q$ such that $c = c_0 + c_1 u$. Then $P + \psi(P)$ can be computed in λ-projective coordinates as follows.*

$$\begin{aligned}
A &= (x_0 \cdot \lambda_1 + x_1) + (x_1 \cdot \lambda_1 + x_0 + x_1)u \\
B &= x_1 \cdot \lambda_1 + x_1 u \\
X_{P+\psi(P)} &= A \cdot (A + B) \\
Z_{P+\psi(P)} &= (x_1^2 \cdot \lambda_1) + x_1^2 u \\
\Lambda_{P+\psi(P)} &= (A + B + x_1^2)^2 + Z_{P+\psi(P)} \cdot (\lambda_P + 1)
\end{aligned}$$

Finally, we also include how to compute $2Q+P$ and $2Q+P_1+P_2$ for respectively the 1D and 2D variant in the last iteration of the scalar multiplication loop, using complete formulas. Our approach was to compute them non-atomically from left to right. We do a normal doubling which is exception-free and use Algorithm 9 for the complete mixed additions. A complete mixed addition costs $11\tilde{m} + 5\tilde{s}$, which is slightly more than a full addition. To make Algorithm 9 run in constant time, CMOV instructions are used for the conditional assignments.

Algorithm 9: Complete mixed addition

```
Input: $P = (x_P, \lambda_P)$ in $\lambda$-affine coordinates, $Q = (X_Q, \Lambda_Q, Z_Q)$ in
       $\lambda$-projective coordinates.
Output: $P + Q$ in $\lambda$-projective coordinates.
1  $R \leftarrow P + Q$ using the incomplete formula.
2  $R_D \leftarrow 2P$
3  $X_P \leftarrow x_P \cdot Z_Q$
4  $\Lambda_P \leftarrow \lambda_P \cdot Z_Q$
5  if $Z_Q = 0$ then
6      $R \leftarrow P$
7  if $X_P = X_Q$ and $\Lambda_P = \Lambda_Q$ then
8      $R \leftarrow R_D$
9  if $X_P = X_Q$ and $\Lambda_P = \Lambda_Q + Z_Q$ then
10     $R \leftarrow \mathcal{O}$
11 return $R$
```

C Timings for field arithmetic on 64-bit Intel platforms

The best timings in the literature for binary field arithmetic on Intel platforms can be found for Sky Lake [31]. We update the timings below with by benchmarking in more modern platforms, including optimizations proposed in [30].

Table 7. Benchmarks (in clock cycles) of the field arithmetic on an Intel Core i7 4770 Haswell CPU running at 3.40GHz, with HyperThreading and TurboBoost disabled. The cost of reduction is included in the cost of the multiplication and squaring. Base field reduction is mod $zf(z)$. Op/m_b, Op/$\widetilde{m}$ denotes the cost relative to respectively base/extension field multiplication.

Field op.	$\mathbb{F}_{2^{127}}$		$\mathbb{F}_{2^{254}}$	
	Cycles	Op/m_b	Cycles	Op/$\widetilde{m}$
Multiplication	24	1.00	40	1.00
Reduction	9	0.37	6	0.15
Squaring	15	0.63	20	0.50
Inversion (ct.)	2219	92.46	2251	56.28
(non-ct.)	602	25.08	668	16.70

Table 8. Benchmarks (in clock cycles) of the field arithmetic on an Intel Core i7 7700 Kaby Lake CPU running at 3.6GHz, with HyperThreading and TurboBoost disabled. The cost of reduction is included in the cost of the multiplication and squaring. Base field reduction is mod $zf(z)$. Op/m_b, Op/$\widetilde{m}$ denotes the cost relative to respectively base/extension field multiplication.

Field op.	$\mathbb{F}_{2^{127}}$		$\mathbb{F}_{2^{254}}$	
	Cycles	Op/m_b	Cycles	Op/$\widetilde{m}$
Multiplication	24	1.00	30	1.00
Reduction	9	0.38	6	0.20
Squaring	15	0.63	18	0.60
Inversion (ct.)	1912	79.67	1965	65.50
(non-ct.)	573	23.88	628	20.93