Faster Complete Formulas for the GLS254 Binary Curve

Thomas Pornin

NCC Group, thomas.pornin@nccgroup.com

31 October, 2023

Abstract. GLS254 is an elliptic curve defined over a finite field of characteristic 2; it contains a 253-bit prime order subgroup, and supports an endomorphism that can be efficiently computed and helps speed up some typical operations such as multiplication of a curve element by a scalar. That curve offers on x86 and ARMv8 platforms the best known performance for elliptic curves at the 128-bit security level.
In this paper we present a number of new results related to GLS254:

- We describe new efficient and complete point doubling formulas (2M + 4S) applicable to all ordinary binary curves.
- We apply the previously described (x, s) coordinates to GLS254, enhanced with the new doubling formulas. We obtain formulas that are not only fast, but also complete, and thus allow generic constant-time usage in arbitrary cryptographic protocols.
- Our strictly constant-time implementation multiplies a point by a scalar in 31615 cycles on an x86 Coffee Lake, and 77435 cycles on an ARM Cortex-A55, improving previous records by 13% and 11.7% on these two platforms, respectively.
- We take advantage of the completeness of the formulas to define some extra operations, such as canonical encoding with (x, s) compression, constant-time hash-to-curve, and signatures. Our Schnorr signatures have size only 48 bytes, and offer good performance: signature generation in 18374 cycles, and verification in 27376 cycles, on x86; this is about four times faster than the best reported Ed25519 implementations on the same platform.
- The very fast implementations leverage the carryless multiplication opcodes offered by the target platforms. We also investigate performance on CPUs that do *not* offer such an operation, namely a 64-bit RISC-V CPU (SiFive-U74 core) and a 32-bit ARM Cortex-M4 microcontroller. While the achieved performance is substantially poorer, it is not catastrophic; on both platforms, GLS254 signatures are only about 2x to 2.5x slower than Ed25519.

1 Introduction

Binary elliptic curves are elliptic curves defined over finite fields of characteristic 2. Compared to more commonly used curves such as NIST's P-256 or Curve25519, which use prime fields, binary curves have historically suffered from three main issues that together explain their relative disuse in practical applications and in security standards:

1. Efficient implementation of operations in binary fields is challenging on software architectures that do not offer a "carryless multiplication" opcode.

2. Some theoretical results applicable to fields of large degree (much larger than used in practical curves) have lead to the suspicion that there might be some theoretical weakness in such curves.
3. Since binary fields are especially well suited to hardware implementations, several organizations filed patents on relevant implementation techniques in the late 1990s and early 2000s. As is usually observed with patents in cryptography, the main effect was a severe reduction of appetite for using or even studying such patent-encumbered curves.

Patents expire after some delay (which depends on the jurisdiction and can vary on a per-patent basis) and the third issue can be solved by using techniques old enough to no longer be patented, even if they were in the past, and simply *not* patenting any new invention. I have not filed any patent for anything described in this paper nor in the previous paper on (x, s) coordinates[32], and I am not aware of any other applicable patent on these formulas and implementation techniques.[1] (This is not legal advice.)

The theoretical issues on curve weaknesses do not apply to practical curve sizes (say, fields of about 256 bits)[28]. The main hurdle is the lingering fuzzy feeling that we, in general, do not fully grasp the security of elliptic curves on binary fields. However, it would be inordinately arrogant, even foolhardy, to believe that we know any better for curves over prime fields. In truth, we don't have a proof nor even any real understanding of why discrete logarithm is a hard problem on ordinary elliptic curves; at best, we have some notions of why index calculus methods do not work for them. In practice, for security purposes using any curve, we rely on the accumulated research efforts, i.e. how long such curves have "been around" and have not been broken yet. Binary curves have been suggested for use in cryptography for as much time as curves over prime fields, and thus should logically be considered as equally secure, *as far as we know*.

This paper explores the first issue, i.e. performance. Specifically, we are interested in obtaining good performance results for an elliptic curve at the traditional "128-bit" security level, on a variety of software platforms (with or without carryless multiplication opcodes), while preserving proper security:

– The curve should offer a prime-order group abstraction, with canonical encoding and decoding, and complete routines for adding and doubling group elements without any special case.
– The implementation should be strictly constant-time, to avoid leaking any secret information through timing-based side channels.

We do not try to tackle any side channel other than time. Timing measurements are "special" in that they can potentially be performed by remote attackers, either over fast networks, or using local facilities that do not require the attacker to be physically close to the target system (crudely said, every server in a data center has a clock and can measure time). Other side channels more or less require the attacker to install attack-specific hardware in the vicinity of the

[1]Informally, *if* there is a patent issue, then it would probably be on the definition of the GLS254 curve itself and its endomorphism, which was described in 2009[14], though not on the *use* of the endomorphism, which follows some earlier work[13] and for which relevant patents have expired. I could not find any hint of GLS curves being patented beyond the now-expired GLV patents, but I can certainly not guarantee the inexistence of applicable patents on the subject, or on anything else.

target[2]. Moreover, while most side channels are specific to the employed hardware and usage context, it is possible to prevent timing-based side channels in a relatively generic way, not tied to a specific hardware model, through the programming discipline known as "constant-time coding", which can be summarized as not using any operation with data-dependent timing characteristics; in particular, no memory access, either for data or code, should happen at an address that depends on secret values. This prevents use of both lookup tables with secret indices, and conditional jumps with secret-dependent conditions.

This work is based upon two main previous works:

– Aardal and Aranha presented in [1] a very optimized implementation of multiplication of a curve point by a secret scalar (as would be used for instance in ECDH key exchange), at the 128-bit security level[3]. Their code held the speed records (until now) on both 64-bit x86 (Intel Skylake-class CPUs) and 64-bit ARM (Cortex-A55). They define and use a specific binary curve called GLS254.
– Our previous work on binary curves[32] presented a novel representation of curve elements, that allowed building a prime order group abstraction with efficent and complete formulas, as well as canonical point encoding and decoding in a compressed format.

We merge here these two works, with a few improvements.

– In section 2, we recall the definition of the GLS254 curve parameters, and explain how the (x, s) coordinates are applied to that curve. A novel and efficient point doubling formula, applicable to all ordinary curves on binary fields, is described, and adapted for (x, s) coordinates.
– Section 3 covers some high-level cryptographic algorithms that can use GLS254. We present a custom map-to-point process for a constant-time hash-to-curve procedure; we also specify efficient signature generation and verification algorithms, which support signatures shorter than the usual Ed25519 (48 bytes instead of 64 bytes), and finally discuss key exchange and ECDH variants.
– In section 4, we describe our implementation on 64-bit x86 CPUs with the `pclmulqdq` opcode. On Intel Skylake-class CPUs, we beat the record from [1] by about 13%, down to 31615 cycles for the same "raw ECDH" operation. For signatures, we sign in 18374 cycles, and verify in 27376 cycles; the latter figure is about 4x faster than the fastest reported Ed25519 value on that platform.
– In section 5, we discuss our implementation on a 64-bit ARM CPU, specifically the ARM Cortex-A55 from an ODROID C4 single-board computer (the same hardware platform as [1]). We again break the record, by about 11.7%, down to 77435 cycles. Signature generation cost is 55526 cycles, verification is 68649 cycles.
– In section 6, we explore a different platform, namely a 64-bit RISC-V core (SiFive-U74) which does *not* offer a carryless multiplication opcode. Various methods to perform the

[2]There is no absolute rule here; modern computers can measure their own temperature and input voltage, and that can be leveraged for side channel attacks among, for instance, logically separated contexts co-located on the same machine. But these are very low bandwidth side channels and their use for attacks is still in the early theoretical, lab-demonstration stages.

[3]At the lower 112-bit security level, slightly better performance is achievable with the 233-bit K-233 Koblitz curve[32].

computations are discussed. The performance is certainly not as good as in the two previous platforms, but it is not catastrophic either; signature generation and verification are about 2x slower than Ed25519.
- A much smaller ARM Cortex-M4 microcontroller is investigated in section 7. Again, the lack of a carryless multiplication opcode degrades performance; we still achieve signature generation in 1034123 cycles, and verification in 1735470 cycles. These figures translate to about 3x the estimated cost of Ed25519 on the same platform.

Our x86, ARMv8 and RISC-V implementations are written in Rust; they are part of the `crrl` research library (version 0.8.0), available there:

https://github.com/pornin/crrl

In fact, the same source code is used for curve operations on the three platforms; only the backend implementation of finite field operations differs.

The ARM Cortex-M4 implementation is written mostly in assembly and C, and is available there:

https://github.com/pornin/gls254-cm4

A Note on Benchmarks. In this paper we present some performance measurements, expressed in clock cycles. We will try to stick to only "large" operations (such as "verifying a signature"), especially on large platforms, because the cost of a small routine – e.g. multiplication in a finite field – is not a well-defined value. For a given operation, one can try to measure *latency* (how soon the output is available after the computation has started) and *throughput* (how many independent operations can be executed in a given amount of time). Neither really matches the actual usage: the compiler will try to inline any small routine within its caller and mix the instructions with other operations, so that a given routine variant might fare worse on benchmarks, e.g. with a higher latency, but improve overall performance by having more "free slots" in which instructions for other operations can be placed. Even if the compiler does not do such inlining and mixing, modern CPUs with out-of-order execution will gleefully do it at runtime, even across function calls that seem separate in the binary, and with a pipeline depth that can range to 200 or more instructions. Under these conditions, it does not make much sense to express the cost of a small routine in isolation. It can be done in a meaningful way for assembly routines on in-order platforms; we will do it when analyzing the ARM Cortex-M4 implementation.

Benchmarks for high-level operations are performed by running the operation repeatedly and measuring the time taken by a number of successive co-dependent executions (so that what is really measured is the latency). Some "warm-up" executions are performed, in order to populate all caches (code, data, branch prediction...), then 100 runs are performed, and the cost is extracted from the time taken by the median run. For signature verification, which is not constant-time, at least 128 distinct signatures are used, to avoid caches "learning" exactly the relevant memory slots. All values are reported in clock cycles, under the (possibly audacious!) assumption that these computations all fit in L1 cache and are somewhat independent from the external RAM bandwidth. These methods are in line with what is commonly practiced when benchmarking cryptographic algorithms.

2 GLS254 in *(x,s)* Coordinates

2.1 Binary Fields

For any integer $m > 0$, $GF(2^m)$ is a finite field of characteristic 2. We recall a few properties of such fields which are relevant for implementation:

- Squaring is a field automorphism in binary fields: for any elements x and y, $(xy)^2 = x^2y^2$ and $(x+y)^2 = x^2+y^2$. Thus, squarings, square roots, and sequences of such operations, are linear and substantially less expensive to compute than multiplications. Every field element is a quaratic residue, and has a single square root.
- Inversion can be computed with a variant of Fermat's little theorem described by Itoh and Tsujii[18], which uses relatively few multiplications and some sequences of squarings, the latter being amenable to table-based implementations thanks to their linearity.
- The *trace* of an element x is:
$$\mathrm{Tr}(x) = \sum_{i=0}^{m-1} x^{2^i}$$
The trace is linear; it is always equal to 0 or 1; the trace of 0 is 0. For any field element x, $\mathrm{Tr}(x) = \mathrm{Tr}(x^2) = (\mathrm{Tr}(x))^2$. If m is odd, then $\mathrm{Tr}(1) = 1$; however, if m is even, then $\mathrm{Tr}(1) = 0$. There is always at least one field element of trace 1 with minimal Hamming weight (i.e. a single non-zero bit).
- For any field element d such that $\mathrm{Tr}(d) = 0$, there are two field elements x such that $x^2 + x = d$; if we defined as $\mathrm{QSolve}(d)$ one such value x, then the other one is $\mathrm{QSolve}(d) + 1$. When m is odd, a solution x can be efficiently found with the halftrace:
$$H(x) = \sum_{i=0}^{(m-1)/2} x^{2^{2i}}$$
Note that when $\mathrm{Tr}(d) \neq 0$, then $H(d)^2 + H(d) = d + \mathrm{Tr}(d)$. When m is even, QSolve is slightly more complicated, but can still be efficiently computed.

The GLS254 curve is defined over $GF(2^{254})$. That field is itself defined as a degree-2 extension of $GF(2^{127})$. The latter is obtained as the quotient ring of $GF(2)[z]$ (the binary polynomials in z) by the ideal generated by the irreducible polynomial $M = 1 + z^{63} + z^{127}$. The choice of the modulus M does not matter algebraically, as long as it is irreducible (all finite fields with the same cardinal are isomorphic to each other); this specific polynomial was chosen because it allows for some efficient implementations[26].

$GF(2^{254})$ is then obtained by defining a formal element u such that $u^2 + u = 1$ (there is no such element in $GF(2^{127})$); an element $x \in GF(2^{254})$ can then be uniquely written as $x = x_0 + ux_1$ for two values $(x_0, x_1) \in GF(2^{127}) \times GF(2^{127})$. There is a single element of trace 1 with minimal Hamming weight in $GF(2^{254})$; this is u itself (thus, the trace of x is equal to the least significant bit of x_1, in a classic polynomial representation).

We define the Frobenius automorphism ϕ in $GF(2^{254})$ as:

$$\phi(x) = x^{2^{127}}$$

It can be seen that $\phi(x_0 + ux_1) = x_0 + x_1 + ux_1$. Moreover, for any $x \neq 0$, then $\phi(x) \neq 0$, and:

$$x\phi(x) = x_0^2 + x_1^2 + x_0 x_1$$

which is an element of $GF(2^{127})$. This allows computing inversions in $GF(2^{254})$ with a single inversion in $GF(2^{127})$, using $1/x = \phi(x)/(x\phi(x))$.

Over $GF(2^{254})$, for $d = d_0 + ud_1$, we compute $\mathrm{QSolve}(d) = x = x_0 + ux_1$ with the following process:

1. $x_1 \leftarrow H(d_1)$
2. If $\mathrm{Tr}_{127}(x_1) \neq \mathrm{Tr}_{127}(d_0)$, then: $x_1 \leftarrow x_1 + 1$
3. $x_0 \leftarrow H(d_0 + x_1^2)$

with Tr_{127} being the trace over $GF(2^{127})$. The second step ensures that $\mathrm{Tr}_{127}(d_0 + x_1^2) = 0$, and therefore $x_0^2 + x_0 = d_0 + x_1^2$. It can be verified that this ensures that $x^2 + x = d + u\mathrm{Tr}(d)$, for all field elements d.

2.2 Curve Parameters and *(x,s)* Coordinates

We start with the GLS254 curve as used in [1] (which followed up on work presented earlier[26,27]): the curve $\mathcal{E}$ is the set of points $(x, y) \in GF(2^{254}) \times GF(2^{254})$ such that:

$$y^2 + xy = x^3 + Ax^2 + B$$

for the constants $A = u$ and $B = 1 + z^{27}$. An extra formal point, the point-at-infinity $\mathbb{O}$, does not have defined coordinates, and serves as the neutral element for the group law defined over the curve. The curve happens to contain $2r$ points in total, for a prime r close to 2^{253}:

$$r = 2^{253} + 83877821160623817322862211711964450037$$

The constant B has been chosen so that multiplication of a field element by B is inexpensive.

This curve is believed to offer "128-bit security"[4]. The fact that $GF(2^{254})$ can be defined as a degree-2 extension of $GF(2^{127})$ opens the conceptual possibility of applying the GHS attack on the curve[15]; however, [17] offers extensive arguments on why that attack can only work with some specific curve parameters. Aardal and Aranha verified that their specific choices for A and B are not in the class of curve parameters on which the GHS attack can apply.

For building arbitrary cryptography protocols, a *prime order group abstraction* is required. The GLS254 curve contains a subgroup of order r, called the points of r-torsion (i.e. exactly the points P such that $rP = \mathbb{O}$) and denoted $\mathcal{E}[r]$. It can be shown that the points in $\mathcal{E}[r]$ are exactly $\mathbb{O}$, and the curve points (x, y) such that $\mathrm{Tr}(x) = \mathrm{Tr}(A) = 1$; any point deserialization routine can thus easily verify that a provided point (x, y) is part of the curve (by applying the curve equation) and is furthermore in $\mathcal{E}[r]$ (by checking the value of $\mathrm{Tr}(x)$).

However, classic point representation and formulas for computing the group law on the curve (or its subgroup $\mathcal{E}[r]$) have some drawbacks:

[4]Technically it is "126.5-bit security", since generic discrete logarithm in a group of size r works with effort about $\sqrt{r}$, but we can argue that each group operation is more expensive to compute than a single bit operation; similarly, Curve25519 is often said to offer 128-bit security instead of "only" 126-bit security, as would be inferred from its subgroup order close to 2^{252}.

– The point-at-infinity $\mathbb{O}$ does not have defined coordinates, and must thus use a special (conventional) representation.
– The point addition formulas have some special cases in which they fail to compute the correct result. These are all cases that involve $\mathbb{O}$ as one of the operands (adding $\mathbb{O}$ to another point) or as the result (adding a point to its opposite), and also involuntary point doublings (adding a point to another point which happens to be equal to the first one). These formulas are said to be *incomplete*.

Incomplete formulas are a problem for use in cryptography because implementations must then account for the special cases in a way which does not leak information through side channels (and in particular whether any given operation turned out to be a special case). In some protocols, an *a priori* analysis can show that some operations cannot hit a special case; such an analysis is an important part of the point multiplication routine described by Aardal and Aranha. In the general case, we would like to have an efficient and complete point addition routine, which would remove any such concern and avoid the need for an analysis that can be complicated. Most preferably, the complete routine would be an application of *complete formulas*, i.e. formulas that inherently work for all cases.

(x, s) coordinates, described in [32], offer such efficient and complete formulas. The application of (x, s) coordinates to the GLS254 curve works as follows:

1. For a point $(\bar{x}, \bar{y}) \in \mathcal{E}[r]$ on the original curve (with equation $\bar{y}^2 + \bar{x}\bar{y} = \bar{x}^3 + A\bar{x}^2 + B$), define:

$$\begin{aligned} a &= A^4 \\ b &= B^2 \\ \dot{x} &= \bar{x}^4 \\ \dot{y} &= \bar{y}^4 + B^2 \end{aligned}$$

The point $(\dot{x}, \dot{y})$ is then an element of the curve defined by the (non-short) Weierstraß equation:

$$\dot{y}^2 + \dot{x}\dot{y} = \dot{x}(\dot{x}^2 + a\dot{x} + b)$$

This map is bijective and preserves the algebraic structures of the curve; it can thus be considered to be a mere change of representation while still being on the "same" curve.

2. The curve admits a single point of order 2, denoted N. We now replace the point $P = (\dot{x}, \dot{y})$ with the point $P + N = (x, y)$ such that:

$$\begin{aligned} x &= \frac{b}{\dot{x}} \\ y &= \frac{b(\dot{y} + \dot{x})}{\dot{x}^2} \end{aligned}$$

In effect, this means that we are *representing* a point $P \in \mathcal{E}[r]$ by the point $P+N \notin \mathcal{E}[r]$; for $P, Q \in \mathcal{E}[r]$, the representation of $P + Q$ is then $P + Q + N = (P + N) + (Q + N) + N$. The advantage of this convention is that the neutral element ($\mathbb{O}$) is represented by the point N, which has defined coordinates and is thus amenable to the definition of complete formulas.

3. Replace the y coordinate with s such that:

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

   Given x, the y coordinate can be recomputed from s, and vice versa.

With these conventions, we can apply the complete formulas described in [32]. The (x, s) coordinates use an extended representation as four field elements $(X{:}S{:}Z{:}T)$, with:

$$\begin{aligned} Z &\neq 0 \\ x &= \frac{\sqrt{b}X}{Z} \\ s &= \frac{\sqrt{b}S}{Z^2} \\ T &= XZ \end{aligned}$$

In the first step above, we raised the initial source coordinates to the power 4 precisely so that $\sqrt{b} = \sqrt{B^2} = B$: the formulas use multiplications by $\sqrt{b}$ and we want these multiplications to be inexpensive. With $A = u$, the parameter a is equal to $A^4 = u$ (indeed, u is a cube root of 1 in the field).

We recall in appendix A the formulas (in pseudocode) for the relevant operations on which the group abstraction operates (encoding to a compressed format and decoding, point addition and doublings, comparisons...). The complete formulas also happen to be faster than previously known incomplete formulas (e.g. in (x, λ) coordinates[26], a generic point addition has cost 11M + 2S, but in (x, s) coordinates this cost is lowered to 8M + 2S).

A conventional generator element G is chosen; any non-neutral element can be used as generator, since the group has prime order r. We reuse the same generator as in [1] (specifically, from the Sage script `sage/ec.sage`, lines 20-28, in the companion code repository for that paper), translated to (x, s) coordinates. Encoding the x and s coordinates into hexadecimal strings, we have:

$$\begin{aligned} G_x = &\quad \sqrt{b}\ \texttt{0x657CB9F79AE29894B6412F20326B8675} \\ &+u\ \sqrt{b}\ \texttt{0x14C6F62CB2E3915E3932450FF66DD010} \\ G_s = &\quad \sqrt{b}\ \texttt{0x763522ADA04300F15FADCA04023DC896} \\ &+u\ \sqrt{b}\ \texttt{0x4F69A66A2381CA6D206E4C1E9E07345A} \end{aligned}$$

In the notation above, we implicitly apply a conversion from the integer $\sum_i d_i 2^i$ to the binary polynomial $\sum_i d_i z^i$; moreover, the hexadecimal strings match the expected internal representation as "scaled affine" coordinates, i.e. extended coordinates with $Z = 1$, which is why there is an additional $\sqrt{b}$ scaling factor.

In the next two subsections we also present novel formulas for computing point doublings, and a succinct description of the efficient curve endomorphism that can be leveraged to speed some operations up.

2.3 New Doubling Formulas

For this subsection, we first switch back to the general case of a short Weierstraß curve. Suppose that the curve has equation $\bar{y}^2 + \bar{x}\bar{y} = \bar{x}^3 + A\bar{x}^2 + B$. Coordinates $\bar{x}$ and $\bar{y}$ are normally represented with some sort of fractional system (e.g. projective coordinates) so that group operations on the curve can be expressed in a few number of additions, subtractions and multiplications in the field.

In a binary field, addition and subtraction are the same operation, and since it basically boils down to a bitwise XOR, we ignore its cost when comparing formulas. We similarly ignore the cost of multiplying any field element by the constant A: for any non-zero field element C, the change of variable $y \mapsto y + Cx$ is a curve isomorphism that maps that equation to another short Weierstraß equation where the constant A is replaced with $A + C^2 + C$ (but B is unchanged). In that way, the constant A can always be arranged to be any other field element with the same trace. In particular, when m is odd, then one can always have $A = 0$ or 1; when m is even, $\mathrm{Tr}(1) = 0$, but there is always at least one field element with trace 1 and minimal Hamming weight (in $GF(2^{254})$ as defined previously, this is u) such that multiplication by A can be made as inexpensive as an addition in the field.

Formula costs are typically expressed in terms of generic multiplications of field elements (M), squarings (S), and multiplications by the constant B ($\mathrm{m_b}$). In some curves, in particular GLS254, B is chosen so that multiplication by B has low cost (much lower than that of a generic multiplication). In software architectures that do not offer a carryless multiplication opcode, squaring cost is typically around 1/10th of multiplication cost, and the number of multiplications in a formula is the dominant factor; when a carryless multiplication opcode is available, squaring cost will often be between 0.5 and 0.75 times the multiplication cost. For point doublings, the previously best known formulas have cost $2\mathrm{M} + 4\mathrm{S} + 2\mathrm{m_b}$ (from [5], as an improvement over an idea from [20]), but are applicable only to curves such that $A = 1$; when $A = 0$, the cost is $2\mathrm{M} + 5\mathrm{S} + 2\mathrm{m_b}$. As we saw, when working over a binary field $GF(2^m)$ for an even m, $\mathrm{Tr}(0) = \mathrm{Tr}(1) = 0$, and trace-1 curves cannot use either $A = 0$ nor $A = 1$; indeed, GLS254 uses $A = u$ instead.

We present here formulas which achieve cost $2\mathrm{M} + 4\mathrm{S} + 2\mathrm{m_b}$ for all ordinary binary curves with a short Weierstraß equation, regardless of the value of A. We suppose that the $(\bar{x}, \bar{y})$ coordinates use the extended representation $(\bar{X}{:}\bar{Y}{:}\bar{Z}{:}\bar{T})$ with:

$$\begin{aligned}
\bar{Z} &\neq 0 \\
\bar{x} &= \frac{\bar{X}}{\bar{Z}} \\
\bar{y} &= \frac{\bar{Y}}{\bar{Z}^2} \\
\bar{T} &= \bar{X}\bar{Z}
\end{aligned}$$

In that case, the double of point $(\bar{X}{:}\bar{Y}{:}\bar{Z}{:}\bar{T})$ can be computed as $(\bar{X}'{:}\bar{Y}'{:}\bar{Z}'{:}\bar{T}')$, with:

$$\begin{aligned}
\bar{Z}' &= \bar{T}^2 \\
\bar{D}' &= (\bar{X} + \sqrt{b}\bar{Z})^2 \\
\bar{X}' &= \bar{D}'^2 \\
\bar{T}' &= \bar{X}'\bar{Z}' \\
\bar{Y}' &= (\bar{Y}(\bar{Y} + \bar{D}' + \bar{T}) + (a + b)\bar{Z}')^2 + (a + 1)\bar{T}'
\end{aligned}$$

(We consider multiplication by $\sqrt{b}$ and by b to have the same cost $\mathrm{m_b}$; this is indeed the case for the implementations described in this paper.)

The formulas above are complete, provided that the point-at-infinity $\mathbb{O}$ is represented as a quadruplet $(X{:}0{:}0{:}0)$ for any $X \neq 0$.

In the case of GLS254, the source point is obtained in a different representation on a non-short Weierstraß equation. We can use the formulas above, provided that we add conversion steps. Suppose that we receive a point $(X{:}S{:}Z{:}T)$ ((x, s) coordinates for point $P + N$, with $P \in \mathcal{E}[r]$) and want to compute $n \geq 1$ successive doublings, i.e. obtain $2^n P + N$:

1. Convert the source value to a point $(\bar{X}{:}\bar{Y}{:}\bar{Z}{:}\bar{T})$ in the curve with short Weierstraß equation $\bar{y}^2 + \bar{x}\bar{y} = \bar{x}^2 + a\bar{x}^2 + b^2$, with the following formulas:
$$\begin{aligned}\bar{X} &= \sqrt{b}X\\ \bar{T} &= \sqrt{b}T\\ \bar{Z} &= Z\\ \bar{Y} &= \sqrt{b}S + \bar{X}^2 + a\bar{T}\end{aligned}$$
2. Apply the n doublings in the short Weierstraß extended representation, using the formulas presented above.
3. Convert back the result to our (x, s) extended representation:
$$\begin{aligned}X &= \sqrt{b}\bar{Z}\\ S &= \sqrt{b}(\bar{Y} + (a+1)\bar{T} + \bar{X}^2)\\ Z &= \bar{X}\\ T &= \sqrt{b}\bar{T}\end{aligned}$$

Note that the first step yields a representation of $P{+}N$ in the short Weierstraß curve; however, since there is at least one doubling, the point obtained at the start of the third step is in $\mathcal{E}[r]$, and the final conversion formulas include the addition of N to obtain a proper extended (x, s) representation. It is easily verified that the formulas are complete (i.e. they also work if the orignal input point is N).

The overall cost of n successive doublings in extended (x, s) coordinates is then $n(2\text{M} + 4\text{S} + 2\text{m}_\text{b}) + 2\text{S} + 6\text{m}_\text{b}$. Previously published formulas[32] computed n point doublings with cost $n(3\text{M} + 4\text{S} + \text{m}_\text{b}) + 3\text{m}_\text{b}$; these new formulas are faster when multiplication by $\sqrt{b}$ is relatively inexpensive and n is large enough, although the exact threshold depends on the implementation platform. In our code, for x86 with the `pclmulqdq` opcode, we use these new formulas when $n \geq 2$; on ARM Cortex M4, squarings and multiplications by b or $\sqrt{b}$ are fast enough that the new formulas are always better.

2.4 The GLS254 Endomorphism

GLS254 is a GLS curve[14], which means that it was defined in such a way that an efficient endomorphism can be computed on it, and the endomorphism can speed up computations. Namely, on the original curve (with the short Weierstraß equation $\bar{y}^2 + \bar{x}\bar{y} = \bar{x}^3 + A\bar{x}^2 + B$), we define a formal value α (which is not in $GF(2^{254})$ but in a degree-2 extension $GF(2^{508})$) such that $\alpha^2 + \alpha = A$; then the endomorphism is ψ such that:

$$\psi(\bar{x}, \bar{y}) = (\phi(\bar{x}), \phi(\bar{y}) + \phi(\bar{x})(\phi(\alpha) + \alpha))$$

with ϕ being the Frobenius automorphism. The construction is explained in the Galbraith-Lin-Scott paper; ψ can be verified to be an endomorphism on the curve such that, for any point P, $\psi(\psi(P)) = -P$, so that ψ really is the multiplication of the point P by a scalar which is a square root of -1 modulo r. In GLS254, applying these definitions to $A = u$, we find that $\phi(\alpha) + \alpha = u$.

Applying the formulas above through our change of variables, (x, s) coordinates, and representation of point P with $P + N$, we obtain the translated endomorphism ζ which we will use thereafter. If (x, s) is a group element (with $x = x_0 + ux_1$ and $s = s_0 + us_1$), then $\zeta(x, s) = (x', s')$, with:

$$\begin{aligned} x' &= \phi(x) &&= (x_0 + x_1) + ux_1 \\ s' &= \phi(s) + (u+1)\phi(x) &&= (s_0 + s_1 + x_0) + u(s_1 + x_0 + x_1) \end{aligned}$$

In extended $(X{:}S{:}Z{:}T)$ coordinates, the formulas for ζ become the following:

$$\begin{aligned} X' &= \phi(X) &&= (X_0 + X_1) + uX_1 \\ S' &= \phi(S) + (u+1)\phi(T) &&= (S_0 + S_1 + T_0) + u(S_1 + T_0 + T_1) \\ Z' &= \phi(Z) &&= (Z_0 + Z_1) + uZ_1 \\ T' &= \phi(T) &&= (T_0 + T_1) + uT_1 \end{aligned}$$

As can be seen, all these formulas only imply a few additions in $GF(2^{127})$ and can thus be computed very efficiently.

With the definitions above, for any group element P, $\zeta(P) = \mu P$, with:

$$\begin{aligned} \mu = {} & 108110115148377375347170251625214377052 \\ & 3874957562909877084374141370993 1738644 \end{aligned}$$

For any integer k in the 0 to $r - 1$ range, we can obtain a "split" version as a pair of signed integers (k_0, k_1) such that:

$$\begin{aligned} k &= k_0 + \mu k_1 \mod r \\ k_0^2 &< r \\ k_1^2 &< r \end{aligned}$$

The method is the same as in the original GLV paper[13]. We start with two integers e and f such that $e^2 + f^2 = r$, and $\mu = e/f \mod r$; such integers are easily found with some known methods such as Lagrange's algorithm for lattice basis reduction. We have:

$$\begin{aligned} e &= 85070591730234615854573802599387326102 \\ f &= 85070591730234615877113501116496779625 \end{aligned}$$

We then compute:

$$\begin{aligned} c &= \lfloor kf/r \rceil \\ d &= \lfloor ke/r \rceil \\ k_0 &= k - de - cf \\ k_1 &= df - ce \end{aligned}$$

Since it is guaranteed that $|k_0|$ and $|k_1|$ are lower than $\sqrt{r} \approx 2^{126.5}$, the computations of k_0 and k_1 can be performed over signed 128-bit integers, simply truncating higher bits.

The main cost is computing the rounded divisions $\lfloor ke/r \rceil$ and $\lfloor kf/r \rceil$. Since r is close to a power of two, some shortcuts are possible:

1. Write $r = 2^{253} + r_0$; the integer r_0 is somewhat lower than 2^{126}.
2. Given k, compute $g = ke + (r-1)/2$ and split it at the 253-bit mark into $g = g_0 + 2^{253} g_1$ (with $0 \leq g_0 < 2^{253}$).
3. Compute $g_1 r_0$; if that value is lower than or equal to g_0, then $\lfloor ke/r \rceil = g_1$; otherwise, $\lfloor ke/r \rceil = g_1 - 1$.

The gist of the method above is that $\lfloor ke/r \rceil = \lfloor g/r \rfloor$; then, we use 2^{-253} as an approximation of $1/r$, and adjust the result by subtracting 1 if necessary ($1/r$ is in fact slightly lower than 2^{-253}, so the simple shift by 253 bits may overestimate the quotient, but with $k < r$, that overestimate cannot exceed 1). The second rounded division ($\lfloor kf/r \rceil$) is computed in the same way. Since this entails only integer multiplications with limited ranges, an efficient and constant-time implementation can be obtained.

3 High-Level Algorithms

3.1 Hash-to-Curve

Given a generic map of field elements to curve points, such that a curve point can have only a limited number of antecedents through that map, a hash-to-curve process can be defined[8] to map some arbitrary input data to curve elements, such that curve points are selected with a distribution indistinguishable from a uniformly random choice, and their discrete logarithm with regard to the conventional generator point is unknown:

1. Hash the input with a suitable classic hash function, such that the output can be split into two sub-sequences that are both mapped to field elements with negligible selection bias.
2. Map each field element to a curve point.
3. Add the points together.

A suitable map can be obtained with some generic methods described by Shallue and van de Woestijne[35] and later optimized by Ulas[38]. We present here a variant, which can be efficiently implemented and yields an (x, s) representation of a point $P + N$ for $P \in \mathcal{E}[r]$. The input is a sequence of 256 bits.

1. The input sequence consists of bits (h_i) for $i = 0$ to 255. We first map that sequence to a field element $c \in GF(2^{254})$:

$$c = \sum_{i=0}^{126} h_i z^i + u \left(1 + \sum_{i=2}^{126} h_{i+128} z^i \right)$$

   Thus, bits h_{127}, h_{128}, h_{129} and h_{255} are ignored for obtaining c (bit h_{128} will be used in a later step; the other three bits will remain unused). Note that the choice of c implies that $\mathrm{Tr}(c) = 1$ and $\mathrm{Tr}(c/z) = 0$.

2. Compute the field elements m_i for $i = 1$ to 3:

$$
\begin{aligned}
m_1 &= c \\
m_2 &= c + z^2 \\
m_3 &= c + c^2/z^2
\end{aligned}
$$

   then, for each m_i, the value $e_i = b/m_i$. It can be verified that $\mathrm{Tr}(m_i) = 1$ for all i, and $e_1 + e_2 + e_3 = 0$.
3. Set $(m, e) = (m_i, e_i)$ for the lowest i such that $\mathrm{Tr}(e_i) = 0$. Since $e_1 + e_2 + e_3 = 0$, they cannot all three have trace 1, and the (m, e) pair is well-defined.
4. Set $d = \sqrt{m}$. Since $\mathrm{Tr}(d) = \mathrm{Tr}(m) = 1$, we can compute $w = \mathrm{QSolve}(d)$, which is then such that:

$$d = w^2 + w + a$$

   Since QSolve can return two possible solutions (w and $w + 1$), we select the one such that the least significant bit of w matches the source bit h_{128} (i.e. we forcibly set the least significant bit of w to h_{128}).
   At that point, we have $e = b/d^2$ with $\mathrm{Tr}(e) = 0$; these are the exact conditions for successful and unambiguous decoding of w into a group element, as described in [32] (section 4.3), which we then apply in the following steps.
5. Compute:

$$
\begin{aligned}
f &= \mathrm{QSolve}(e) \\
x &= df
\end{aligned}
$$

   There are two solutions to the equation $f^2 + f = e$ and it is not specified which one is returned by QSolve; we normalize it by choosing the solution that implies that $\mathrm{Tr}(x) = 0$. In other words, if $\mathrm{Tr}(x) \neq 0$, then we add d to x. Since $\mathrm{Tr}(d) = 1$, this always unambiguously selects the value of x.
6. Set $s = xw^2$.
7. Convert (x, s) to the extended coordinate system:

$$
\begin{aligned}
X &= x \\
S &= s\sqrt{b} \\
Z &= \sqrt{b} \\
T &= x\sqrt{b}
\end{aligned}
$$

   Since $\mathrm{Tr}(x) = 0$, the obtained point is necessarily equal to $P + N$ for some $P \in \mathcal{E}[r]$.

Each value w can correspond to a unique group element (this is how the compressed encoding works). For a given w, there is a unique matching d, which can itself be obtained from at most three possible values m_i. Values m_1 and m_2 each have a unique antecedent c, while m_3 can have at most two antecedents c. Since h_{129} is ignored, we obtain that the process above follows a map from $GF(2^{254})$ to the order-r group, such that any group element has at most eight antecedents. This is enough for the hash-to-curve process (with two invocations of the map).

3.2 Signatures

Schnorr signatures[34] are an efficient mechanism for building digital signatures out of a prime order group. Such a signature is, nominally, a triplet (R, c, s) where R is a group element (the *commitment*), c is the *challenge*, and s the *response*. Both s and c are scalars (integers modulo the group order r). If the public key is the group element Q, then the signature is acceptable for a message m if and only if $R = sG - cQ$ and $c = h(R, Q, m)$ for some hash function h that takes as inputs the point R, the public key Q, and the message m, and outputs a scalar. The signature is generated by first choosing a secret scalar k (that must be chosen uniformly, and must be used for one signature instance only), then computing $R = kG$, then c with the hash function h, and finally $s = k + cd$, where d is the private key such that $Q = dG$.

A Schnorr signature can be readily compressed by omitting either R or c. If c is omitted, then the verification process recomputes it from R, Q and m with the hash function h; this is method chosen by the classic scheme Ed25519. It leads to 64-byte signatures, that furthermore support the batched verification process that was the main point of the original Ed25519 paper[3]. However, instead of omitting c, one can omit R, in which case R is recomputed with $R = sG - cQ$ during the verification. This opens the possibility of using a challenge c which is shorter than the group order r. This was already noticed by Schnorr in his original paper[34]. Neven, Smart and Warinschi offer extensive analysis on why c can be made about half shorter than r while maintaining the expected security level[24]. This allows making signatures shorter (48 bytes instead of 64 bytes, at the 128-bit security level), *and* also speeds up the verification, as was remarked for jq255 curves[31]. This shorter format does not support batch verification as in Ed25519, but since it makes each verification faster, this is not necessarily a problem.

For GLS254, we apply an extra improvement. In jq255 curves, the challenge c is obtained as a 16-byte sequence (truncated BLAKE2s output), which is then interpreted as an integer in the 0 to $2^{128} - 1$ range. The analysis in [24] relies on a generic group model where information about a group element is obtained only when a computation outputs that exact group element; thus, it does *not* rely on the possible challenge values to be consecutive integers in a given range. What matters for security is that the challenge c is obtained as the output as a hash function which is secure against random-prefix preimages and second-preimages; this is achieved if we use a "secure" hash function and the truncation/conversion process yields a scalar chosen uniformly in a set of size 2^{128}.

We can thus compute c as follows:

1. Hash the public key Q, commitment R and message m together with a generic purpose hash function such as BLAKE2s, and truncate the output to 16 bytes.
2. Split that 16-byte string into two 8-byte strings, which are each interpreted as integers c_0 and c_1, respectively. These integers are in the 0 to $2^{64} - 1$ range.
3. The challenge is $c = c_0 + \mu c_1$ (with μ being the square root of -1 modulo r that corresponds to the endomorphism ζ from section 2.4).

Two distinct 16-byte hash outputs necessarily yield distinct challenge values; indeed, if $c = c'$, then $(c_0 - c'_0, c_1 - c'_1)$ is a vector in the lattice $L = \{(i, j) \in \mathbb{Z} \times \mathbb{Z} \mid i + \mu j = 0 \bmod r\}$. A shortest basis in that lattice is $((e, f), (f, -e))$ with the values e and f shown in section 2.4, where both vectors have norm $\sqrt{r} \approx 2^{126.5}$; no vector in the lattice can be shorter than that, except the null vector. However, $|c_0 - c'_0| < 2^{65}$ and $|c_1 - c'_1| < 2^{65}$, leading to a vector norm that is lower than 2^{66}, considerably smaller than $\sqrt{r}$. Therefore, if $c = c'$, then $c_0 = c'_0$ and $c_1 = c'_1$. It follows that the definition of the computation of c shown above does not alter the security of Schnorr signatures, compared to the more classic case of c being a plain 128-bit integer.

The point of this special definition of c is that it further speeds signature verification up. In Ed25519-style signatures, the verification equation is $R = sG - cQ$, which can use Straus's algorithm[36] with a window optimization. If we assume a window size of w bits, meaning that we process scalar bits by groups of w, and a "window" is a table of points iG or iQ for $1 \leq i \leq 2^{w-1}$, then for an n-bit curve, this computation needs about n point doublings and $2n/w$ extra point additions. The Antipa *et al* optimization[2] can reduce the cost by splitting the challenge c using some heuristic methods, in particular Lagrange's algorithm, which can be implemented at a relatively low cost[29]; the number of extra point additions is not changed (still $2n/w$), but the number of point doublings is halved, down to $n/2$.

Using a half-size challenge c, as suggested by Schnorr and used in jq255 curves, offers some additional speed-ups: there is no longer a need for running Lagrange's algorithm, and the number of extra point additions is lowered to $1.5n/w$; the number of point doublings is still $n/2$.

The new challenge computation process shown above, applicable to GLS254, keeps the number of extra point additions at $1.5n/w$, but again halves the number of point doublings, down to $n/4$, which yields an extra speed-up. Indeed, the recomputation of R can be written as:

$$\begin{aligned} R &= sG - cQ \\ &= (s_0 + \mu s_1)G - c_0 Q - \mu c_1 Q \\ &= s_0 G - c_0 Q + s_1 \zeta(G) - c_1 \zeta(Q) \end{aligned}$$

assuming that s was split into $s_0 + \mu s_1$ as described in section 2.4. Here, the s_0 and s_1 integers have size $n/2$ bits, but they are applied to known fixed points G and $\zeta(G)$; each of these integers can be further split into two $n/4$-bit halves, since a window over $2^{n/4}G$ can be precomputed. The computation of R can thus be reduced to a linear combination of six points with coefficients of size $n/4$ bits, hence doable in $n/4$ point doublings and $6n/4w$ extra point additions. Moreover, the application of ζ in GLS254 is fast enough that half of the windows can be omitted: for an integer i, the value $i\zeta(Q)$ can be dynamically obtained as $\zeta(iQ)$.

3.3 ECDH and raw ECDH

By "raw ECDH", we designate the core functionality of an ECDH key exchange with the specific properties used in [1]:

- An encoding of a group element is received. This is an uncompressed representation which contains the two affine coordinates, over a total of 64 bytes. This value is *public*.
- The group element is decoded, which entails verifying that the two provided coordinates match the curve equation.
- The point is multiplied by a secret scalar, and the result is encoded into a new 64-byte sequence using the same uncompressed representation.

We use this functionality for benchmarks mainly because it was used that way by Aardal and Aranha; this makes the benchmarks directly comparable with each other. Since we use (x, s) coordinates, we expect the two coordinates to be in "scaled affine" format, i.e. to be the X and S values in an extended representation $(X{:}S{:}1{:}X)$. Some notable features of raw ECDH are the following:

– The input point is public; thus, computations on that point, before using the secret scalar, do not need to be constant-time. This is leveraged in [1] in particular for building their 2-dimensional window and normalizing it to affine coordinates.
– Our implementation returns early if the input point is not a valid representation of a group element (i.e. the coordinates of a curve point $P + N$ with $P \in \mathcal{E}[r]$). The curve equation and the trace of X are thus verified.
– Our code currently accepts the neutral N as input. It could be easily rejected at negligible cost by checking that the provided X coordinate is not zero.
– The secret scalar k is provided as an array of 32 bytes, with unsigned little-endian encoding. Our Rust-based implementations verify that the scalar is in the 0 to $r-1$ range, while our implementation for ARM Cortex M4 accepts any value in the 0 to $2^{256} - 1$ range, and implicitly reduces it modulo r. In both cases, the split of k into $k_0 + \mu k_1$, to be used with the endomorphism ζ, is included in the measured performance.

For a stricter and safer key exchange in general, we also implement "ECDH" (not the raw version), along the lines of the similar functionality for the jq255 curves:

– Exchanged points use the more size-efficient 32-byte compressed format.
– The shared key is obtained through a systematic derivation with a hash function, using not only the result of the point multiplication but also the received point (peer public key) and the recipient's own public key (corresponding to the used secret scalar). The resulting key is thus bound to the used keys, and fully unbiased.
– The input point is not considered public. All processing is constant-time; if the point is incorrect, then a "shared" key is still produced (in a way which is indistinguishable from a success by outsiders). This allows use of ECDH in some protocols, such as password-based key exchanges, in which a success/failure might leak important information on low-entropy secrets.
– The neutral point is considered invalid, so that the input fully contributes to the resulting secret.

4 x86 Implementation

4.1 Test Platform

We consider here a 64-bit x86 platform, running under the Linux operating system (Ubuntu 22.04). The CPU is an Intel i5-8259U "Coffee Lake" running at 2.3 GHz; TurboBoost is disabled, and measurements are made with the timestamp counter on single-threaded code running on an otherwise idle machine.

The Coffee Lake is part of a large family of Intel cores derived from the Skylake (first deployed in 2015). The Skylake, Kaby Lake, Cannon Lake and Coffee Lake differ in processing technology, number of cores and cache sizes (especially L3 cache size), but should all have identical timing behaviour on computations such as the ones we consider here, which all fit in L1 caches for both code and data[12]. Aardal and Aranha used a Kaby Lake platform for benchmarks; we verified that their code yields the exact same measured performance on our test platform.

Our implementation is written in the Rust programming language, which is convenient for that kind of code: it offers a programming model with access to about the same CPU features as C, and through the same code generator backend (LLVM, also used by the C compiler Clang). Its syntax supports operator overloading, which is convenient for implementing operations on finite fields and curve points with notations close to the mathematical description. Our code is part of `crrl`, a growing open-source library meant for research purposes on cryptographic algorithms; the features described here are part of version 0.8.0:

https://github.com/pornin/crrl/tree/v0.8.0

The curve implementation is in the `src/gls254.rs` source file, while the finite field backends are in `src/backend/w32/` (32-bit backends) and `src/backend/w64/` (64-bit backend). For the 64-bit backend using the `pclmulqdq` opcode (through compiler intrinsics), the following compilation command is used:

```
RUSTFLAGS="-C target-cpu=skylake" cargo bench \
    --no-default-features -F gls254,gls254bench
```

The option used in `RUSTFLAGS` instructs the Rust compiler to generate and optimize code for a Skylake-class CPU; in particular, this enables the use of `pclmulqdq`, which is supported by Skylake-class CPUs but not by all CPUs that follow the standard 64-bit x86 ABI. This also enables AVX2 opcodes, which are leveraged by our code in several places. The `bench` command compiles some benchmark code (in `benches/gls254.rs`) which performs timing measurements; it also instructs the compiler to use "release mode", i.e. to optimize the output for speed. The feature flags disable compilation of anything that is not directly required for GLS254 support (the `gls254bench` feature enables the raw ECDH functions, which are nominally only for benchmarks); this greatly reduces compilation time, as `crrl` features a growing list of cryptographic algorithms, and is also a bit "heavy-handed" on function inlining, which is good for speed but bad for compilation time.

All our benchmarks use the Rust compiler version 1.72.1 (from September 13th, 2023).

The specific backend for $GF(2^{254})$ operations on x86 with `pclmulqdq` is located in file `src/backend/w64/gfb254_x86clmul.rs`. Intrinsics, not inline assembly, are used to access all the relevant CPU features.

4.2 Raw ECDH Performance

Obtained performance on raw ECDH is summarized in table 1, compared with the values reported by Aardal and Aranha[1]. Each variant name is a combination of a dimension (1DT or 2DT) and a window size (2 to 5). One-dimensional variants use a window with points iP for integers i and source point P, while two-dimensional variants use a combined window with points $iP + j\zeta(P)$ for pairs of integers (i, j). The window size is the number of integer bits that are processed per iteration; in other words, each iteration with window size n involves n point doublings, followed by either two (1DT) or one (2DT) point lookup(s) and addition.

For some reason, Aardal and Aranha use an off-by-one notation for the window size, i.e. our "2DT-2" is described by them as "2DT with $w = 3$".

Variant	This work	Previous work[1]
1DT-3	35383	-
1DT-4	31615	36480
1DT-5	31785	-
2DT-2	32583	35739
2DT-3	32275	38076

Table 1: Performance of raw ECDH on Intel x86 Skylake-class CPUs, using `pclmulqdq`. Values are in clock cycles.

We thus increase the processing speed by about 13.0% (equivalently, the computation cost is reduced by 11.5%). The gain comes from several sources, discussed below.

New Formulas. The (x, s) formulas for point addition do not, in fact, contribute much to the speed gain. Since the window points are normalized to affine in both our implementation and the previous work, the point additions in the main algorithm loop are mixed additions, for which (x, λ) coordinates offer formulas which have about the same cost as the corresponding mixed (x, s) additions. The main gain here comes from the use of the new doubling formulas (section 2.3), which favour long sequences of doublings; this is why the 1DT variants are faster than the 2DT variants in our code.

Inversions. For inversions in $GF(2^{127})$, Aardal and Aranha use two distinct implementations: both use the same addition chain for the exponent, but one optimizes some sequences of squarings with large lookup tables that allow processing the input one byte at a time, while the other one sticks to plain squarings in a loop. The former is faster but inherently not constant-time, which is why it is used only for normalizing the window (working over the input point only, which is public), while the conversion to affine of the output point (which is secret) uses the latter inversion code.

In our code, we use a single constant-time inversion routine; long sequences of squarings are optimized with tables with 128 elements which are processed bit by bit, so that constant-time discipline is maintained. AVX2 intrinsics are used to read tables efficiently; the CPU can read a full 32-byte chunk of data (i.e. two table entries) in a single opcode. Experimentally, we achieve a table processing speed of about 0.75 cycle per table entry; the table is more efficient than performing squarings for sequences of 12 or more squarings. We also use a different addition chain:

$$1 \rightarrow 2 \rightarrow 3 \rightarrow 6 \rightarrow 7 \rightarrow 14 \rightarrow 28 \rightarrow 42 \rightarrow 84 \rightarrow 126$$

This chain implies the same number (9) of field multiplications than the one used in [1], but only needs two tables (for sequences of 14 and 42 squarings), while their chain uses three or four tables (for steps with 18, 48, 30, and optionally 12 squarings); this lowers L1 cache pressure.

In total, our constant-time inversion in $GF(2^{254})$ uses about 750 cyles, which is close to the non-constant-time table-based inversion of [1] (rated at 628 cycles), and much better than their constant-time function (1965 cycles).

Squarings. A squaring in a binary field is the combination of an "expansion" step, and a reduction. The expansion is a squaring over binary polynomials: data bits are simply moved, with an extra zero inserted between any two successive bits. The expansion can be performed with `pclmulqdq`, which, in Rust syntax with compiler intrinsics, looks as follows:

```
let d0 = _mm_clmulepi64_si128(a, a, 0x00);
let d1 = _mm_clmulepi64_si128(a, a, 0x11);
```

However, it is also possible to use only bitwise operations, and the `pshufb` opcode, for the same operation:

```
let m16 = _mm_set1_epi8(0x0F);
let shk = _mm_setr_epi8(
    0x00, 0x01, 0x04, 0x05, 0x10, 0x11, 0x14, 0x15,
    0x40, 0x41, 0x44, 0x45, 0x50, 0x51, 0x54, 0x55);
let t0 = _mm_shuffle_epi8(
    shk, _mm_and_si128(a, m16));
let t1 = _mm_shuffle_epi8(
    shk, _mm_and_si128(_mm_srli_epi16(a, 4), m16));
let d0 = _mm_unpacklo_epi8(t0, t1);
let d1 = _mm_unpackhi_epi8(t0, t1);
```

This variant uses the convenient fact that while `pshufb` is nominally an opcode for shuffling bytes, it can be used as a generic parallel application of any $4 \rightarrow 8$ lookup table (this was already remarked by Hamburg for an AES implementation[16]) if the data is used as index operand instead. This code uses more instructions than the one using `pclmulqdq`, but it also has lower latency, which improves performance for long sequences of successive squarings, i.e. exactly what is used in inversions.

In $GF(2^{254})$, we obtain the reverse effect. Our code receives the two parts of the input as two 128-bit values (`__m128i`) and assembles them into a single 256-bit AVX2 value (`__m256i`). The two expansion sequences above can then be merged into a single one:

```
let a = _mm256_setr_m128i(a0, a1);

let m16 = _mm256_set1_epi8(0x0F);
let shk = _mm256_setr_epi8(
    0x00, 0x01, 0x04, 0x05, 0x10, 0x11, 0x14, 0x15,
    0x40, 0x41, 0x44, 0x45, 0x50, 0x51, 0x54, 0x55,
    0x00, 0x01, 0x04, 0x05, 0x10, 0x11, 0x14, 0x15,
    0x40, 0x41, 0x44, 0x45, 0x50, 0x51, 0x54, 0x55);
let t0 = _mm256_shuffle_epi8(
    shk, _mm256_and_si256(a, m16));
let t1 = _mm256_shuffle_epi8(
    shk, _mm256_and_si256(_mm256_srli_epi16(a, 4), m16));
let d0 = _mm256_unpacklo_epi8(t0, t1);
let d1 = _mm256_unpackhi_epi8(t0, t1);
```

This is followed by the reduction code, where we again stick to AVX2 to perform both reductions in parallel for the cost of one. In total, this implementation has a *higher* latency than the `pclmulqdq` code (mainly because the movements of data between the low and high AVX2 lanes have 3-cycle latency), but a better throughput: even though AVX2 is, in many respects, two SSE2 units running side-by-side, it can only perform one $64 \times 64 \rightarrow 128$ carryless multiplication at a time, not two. Experimentally, using this variant of squarings in $GF(2^{254})$ yielded a gain of about 1000 cycles in our code.

Window Construction. The construction of the 2DT window benefits from a few minor optimizations. Like [1], we use some specialized functions for doubling or tripling an affine point, and for adding a point P to $\zeta(P)$. However, we also scrape a few more cycles by noticing that the resulting points have related Z coordinates: when computing both $P + Q$ and $P - Q$, the two results have the same Z coordinates. Moreover, if $1/Z$ is computed, then $1/\phi(Z)$ is obtained with the Frobenius operator, which is much faster that inverting $\phi(Z)$ separately. Of course, Montgomery's trick is used to mutualize inversions, but these minor optimizations save some multiplications in $GF(2^{254})$.

4.3 High-Level Algorithm Performance

In table 2, we compare our implementation of GLS254 with the best reported Ed25519 implementation on the same platform. That implementation happens to be `crrl` itself: we compared it with OpenSSL as distributed with the operating system, curve25519-dalek version 4.0.0[10], and the C implementations in eBACS[4]; `crrl` code is slightly faster than all these, especially for signature verification, thanks to the use of the Antipa *et al* optimization[5].

[5]curve25519-dalek supports *batch verification*, which reduces the per-verification cost below the single-verification cost of `crrl`. We consider here only the stand-alone case.

Operation	**GLS254**	**Ed25519**
`decode`	1231	9483
`encode`	824	7512
`mul`	30767	103338
`mulgen`	15912	38596
`hash_to_curve`	4190	-
`load_skey`	16529	47403
`sign`	18374	49090
`verify`	27376	108719

Table 2: Performance comparison of GLS254 and Ed25519 for high-level operations on Intel Skylake. Values are in clock cycles.

The operations in table 2 are:

- `decode`: decoding of a point from its 32-byte (compressed) encoding. The decoding includes validation of the input; for GLS254 (but, crucially, not for Ed25519, which has "cofactor issues"), this includes verification that the point is part of the proper prime-order subgroup. The output of `decode` is the in-memory representation in relevant extended coordinates.
- `encode`: the reverse operation of `decode`.
- `mul`: multiplication of a (decoded) point by a scalar.
- `mulgen`: same as `mul`, when it is statically known that the point to multiply is the conventional generator, for which precomputed tables of multiples are included[6].
- `hash_to_curve`: the hash-to-curve process over a small input, which involves two executions of the map described in section 3.1.
- `load_skey`: decoding of the private key from 32 bytes, and recomputation of the public key (i.e. `mulgen`). It would of course be possible to store the encoded public key along with the private key, making this operation much faster; it has been reported, though, that allowing the private and public key to be provided separately implies a risk of key confusion, in which non-matching keys are used, with deleterious effects. Forcing the public key to be systematically recomputed is thus a safer API in general.
- `sign`: production of a signature (48 bytes) over a given short message (which can be a hash value corresponding to a larger input data), using a given private key. The private key is expected as an already loaded in-memory object (i.e. a prior call to `load_skey`). The signature is produced in encoded format (i.e. bytes).

[6]The `crrl` library deliberately limits the size of such tables, so that the compiled code is not huge, and everything remains in L1 cache with room to spare. For GLS254, four tables of 16 elements are included, for a total of 4096 bytes of data; for Ed25519, the tables use 6144 bytes, because its affine points use the "Duif representation" with three coordinates per point. Some minor speed improvements in benchmarks can be obtained with larger tables, but these would not necessarily translate to actual benefits when the code is integrated into a larger application, because use of the cryptographic primitive would kick more application data out of L1 cache.

– `verify`: verification of a signature, against a given public key and short message. The signature is provided as a sequence of bytes. The public key is expected in the already decoded in-memory format; the reported cost is thus the per-verification cost when processing multiple signatures against the same public key. To get the verification cost against an *encoded* public key, add the `decode` cost (which is quite low, for GLS254).

We see that, roughly speaking, GLS254 provides signatures which are not only smaller (48 bytes instead of 64 bytes) but also substantially faster (by a factor of 2.5x to 4x) than Ed25519 signatures. They also do not suffer from cofactor issues or non-canonical encodings since all operations in GLS254 enforce canonicality for a prime order group.

As a side note, the OpenSSL implementation of RSA, a thoroughly optimized piece of code with handmade assembly, takes about 57000 cycles for signature verification with the common 2048-bit key size and the usual $e = 65537$ public exponent. RSA is normally assumed to offer very fast signature verification. Our GLS254 signature verification happens to be twice faster! And that is using considerably shorter signatures (48 bytes instead of 256), at arguably a higher security level (RSA-2048 security is accounted at between 89 and 112 bits, depending on how CPU and RAM costs are extrapolated)[7].

5 ARMv8 Implementation

5.1 Test Platform

For the ARMv8 implementation, we use an ODROID C4 single-board computer, from Hardkernel. This is the exact same test platform as used by Aardal and Aranha. The operating system is again Linux (Ubuntu 22.04) in 64-bit mode. The CPU uses ARM Cortex-A55 cores. This is not a workstation-level system; it is more representative of relatively powerful embedded systems. In our Rust implementation, the relevant code is located in the `src/backend/w64/gfb254_arm64pmull.rs` source file. It uses the NEON intrinsics for SIMD operations, and in particular the `pmull` and `pmull2` opcodes, that compute $64 \times 64 \rightarrow 128$ carryless multiplications.

For compilation, we use the `target-cpu=cortex-a55` compiler flags. The same compiler version (Rust 1.72.1) as the x86 platform is used[8].

5.2 Raw ECDH Performance

Table 3 shows the performance we achieve on this platform; refer to section 4.2 for the variant naming scheme.

[7]Public key operations with RSA can be made much faster if using $e = 3$. However, for a variety of reasons, many of which more related to psychological impact and mythology than science, small RSA exponents are frowned upon, and the newest FIPS 186-5 standard[25] explicitly mandates that e must be at least 65537.

[8]Indeed, the very same compiler is used, with cross-compilation to the `aarch64-unknown-linux-gnu` platform; this yields much faster compilation times than running the Rust compiler on the ODROID, especially since the latter uses a not-very-fast microSD card for storage.

Variant	This work	Previous work[1]
1DT-3	84902	-
1DT-4	78303	92460
1DT-5	83900	-
2DT-2	77435	86525
2DT-3	85238	91682

Table 3: Performance of raw ECDH on ARM Cortex-A55, using NEON. Values are in clock cycles.

We again obtain a speed improvement (of about 11.7%) over the previous record. For this platform, the 2DT-2 code turns out to be slightly better than the 1DT-4 code, although the difference is slight. The optimization notes listed in section 4.2 for the x86 platform also roughly apply here, though with some variations:

– The memory bandwidth of the A55 is much lower than that of the x86, with a maximum of only 64 bits per cycle. This increases the cost of point lookups (which is why the 1DT-5 variant is substantially more expensive that 1DT-4) and also of tables for optimizing sequences of squarings in inversions. Our inversion routine thus forgoes such tables. We also use a different addition chain, though it does not make much of a difference:

$$1 \rightarrow 2 \rightarrow 3 \rightarrow 6 \rightarrow 7 \rightarrow 14 \rightarrow 21 \rightarrow 42 \rightarrow 63 \rightarrow 126$$

– There is no equivalent of the x86 `pshufb` opcode in NEON. On the other hand, the carryless multiplication opcode has high throughput (one per cycle) and very low latency (two cycles).
– NEON opcodes have in general two variants, which operate over either 64-bit or 128-bit values. The A55 can in general "pair" the 64-bit variants (i.e. execute two instructions in the same cycle, as long as they don't have a dependency relationship with each other), but not 128-bit variants. Thus, operations such as reductions in $GF(2^{127})$ can typically be expressed with either 64-bit or 128-bit opcodes, and which variant turns out to yield the best performance depends on the usage context (the 64-bit code involves more instructions but gives the compiler more options for reorganizing code). In our implementation, we use a mixture of both.

Integer Multiplications are Not Constant-Time. A disappointing feature of the ARM Cortex-A55 is that its 64-bit multiplications are *not* constant-time: their latency is 3 cycles, but only 2 cycles if one or both of the operands fits in 32 bits. A consequence is that basically none of the usual elliptic curve implementations is constant-time on this platform.

The GLS254 raw ECDH implementation from [1] is *mostly* constant-time since integer multiplications are used only in operations over the scalar; hence, only the scalar-splitting step (for use of the curve endomorphism) may potentially leak information. The probability of hitting a problematic case is very low (a random 64-bit value has only probability 2^{-32} of having its 32 top bits equal to zero) and in a key exchange setup, an active attacker targeting a given secret scalar cannot make that scalar vary. Thus, it can be argued that the non-constant-time properties of the A55 multiplications are not a practical issue in that case. A similar reasoning could be made for digital signatures, since secret scalars are not under adversarial control, and the probability of hitting a "bad case" is very low; moreover, a properly randomized signature generator cannot be made to repeat any specific per-signature secret scalar k value, preventing the attacker from repeating experiments in order to amplify the side channel signal (which is only a 1-cycle difference).

For elliptic curve implementations using prime fields, and in particular for ECDH, in which a target implementation is made to work with an attacker-provided point that can exercise the side channel, this is more problematic. A possible mitigation, at least for point coordinates, is to randomize the internal point representation (basically multiplying the Z coordinate by a random value, and adjusting the other coordinates accordingly), though its effectiveness should be assessed.

In any case, to remove any remaining uncertainty about the non-constant-time multiplications, we chose to restrict our code to only 32-bit values (and $32\times 32 \rightarrow 64$ multiplications) for all operations on scalars, when working on an ARM platform. The cost of splitting a scalar is thus raised, but the overhead remains slight when compared with full curve operations (the scalar splitting cost is still lower than 1000 cycles).

5.3 High-Level Algorithm Performance

In table 4, we compare our implementation of GLS254 with Ed25519 on the ARM Cortex-A55. Again, `crrl` turns out to be the best Ed25519 implementation *in general*, but we must note that for the X25519 function (specialized key exchange over Curve25519), there is at least the very good implementation from Lenngren[22], which leverages the NEON floating-point operations and performs the operation in less than 160000 cycles.

For comparison, curve25519-dalek generates and verifies Ed25519 signatures in about 199000 and 560000 cycles, respectively, on our test system; OpenSSL's performance is worse, at 313000 and 633000 cycles, for the same operations. eBACS does not have any A55 timings; it lists two sets of measurements for the slightly older ARM Cortex-A53, which for this code should have timings similar to that of the A55[9], amounting to about 211000 and 591000 cycles for signing and verifying, respectively.

[9]The A53, as typically found on Raspberry Pi-3 single-board computers, does *not* support the large-input carryless multiplication ocode `pmull`; GLS254 would be much slower on such an A53 CPU. However, for Ed25519, there is no use of carryless multiplications, and the integer opcodes have timings similar to that of the A55. The A55 has improved memory management and branch prediction, but this should have little impact on an Ed25519 implementation which should entirely fit in L1 cache.

Operation	GLS254	Ed25519
`decode`	3506	34514
`encode`	1746	15979
`mul`	76219	400473
`mulgen`	46295	151159
`hash_to_curve`	13243	-
`load_skey`	48651	170901
`sign`	55526	175136
`verify`	68649	384541

Table 4: Performance comparison of GLS254 and Ed25519 for high-level operations on ARM Cortex-A55. Values are in clock cycles.

We again find that GLS254 allows signature operations to be performed much faster than for Ed25519 (up to 5.6x for signature verifications). Part of the relative slowness of Ed25519 may be attributed to a lack of optimization efforts specific to that platform; however, there is no doubt that GLS254 is a much faster curve on such systems.

6 RISC-V Implementation

6.1 Test Platform

RISC-V is an open instruction set architecture that includes a barebone "full RISC" core with only a few instructions, and a large list of standardized (but optional) extensions. It is an increasingly popular ISA for all kinds of small and large embedded systems because it is, by design, a royalty-free ISA which can be used in custom hardware designs without licensing costs.

Our test system is a VisionFive 2 single-board computer (by StarFive). It uses a JH7110 CPU, itself built out of SiFive U74 cores. These cores follow the RV64I variant (64-bit, base integer computations) along with extensions M (integer multiplications and divisions), A (atomic operations on memory), F and D (single and double-precision floating point), C (compressed 16-bit instructions), Zba (accelerated address generation, mostly one-cycle instructions to compute $x + vy$ for integers x and y, and v a small power of two) and Zbb (basic bit manipulation, e.g. word rotation opcodes)[10]. The I, M, A, F and D combination is often abbreviated as "G", and, combined with the C extension, make up the `riscv64gc` architecture which is one of the main architectures supported by LLVM (and thus both the Rust and Clang compilers). Crucially, that core does *not* support the Zbc extension (nor its subset Zbkc); the Zbc extension is the one that includes carryless multiplication opcodes. This system is then a representative of what happens with GLS254 implementations when no carryless multiplication opcode is provided.

For compilation, we use the following flags:

[10] It also supports a few extensions related to control registers, which are not relevant here, except for the cycle counter, which we use for benchmarking purposes.

```
RUSTFLAGS="-C target-cpu=sifive-u74 -C target-feature=+zba,+zbb" \
    cargo bench --no-default-features -F gls254,gls254bench
```

In our code, both the Zba and Zbb opcodes provide some important performance improvements; we have to enable them explicitly because the SiFive-U74 is itself a configurable design, and any hardware vendor integrating U74 cores in a chip may or may not include either extension. The relevant source code file is `src/backend/w64/gfb254_m64.rs`.

ISA details and instruction timing characteristics are important for implementation efficiency:

– 64-bit multiplications use distinct opcodes to obtain the low (`mul`) and high (`mulhu`) halves of the 128-bit result. The CPU can issue one of either instruction at every clock cycle, and they have 3-cycle latency. They appear to be constant-time (the SiFive-U74 manual does not mention any timing variation, and we could not detect any early return with special inputs such as zero). The RISC-V ISA documents that if `mul` and `mulhu` are used in that order, with no intervening instruction in between, and working over the same source operand registers, then some RISC-V implementations may notice that they both work on the same mathematical operation and take advantage of it for better performance; this does not appear to be the case for the U74. A practical consequence is that a $64 \times 64 \longrightarrow 128$ multiplication costs twice as much as a $64 \times 64 \longrightarrow 64$ multiplication, in terms of instruction throughput at least.
– The RISC-V does not inherently support carry propagation; there is no "add-with-carry" opcode. In order to perform carry propagation (e.g. when adding big integers represented over several 64-bit limbs), the output carry value must be obtained by comparing the result with one of the operands. A complete add-with-carry operation, with both input and output carries, needs a sequence of five opcodes and a 4-cycle latency (for the output carry). This does not matter much for GLS254 (since binary fields do not use carry propagation), but it impacts curves over prime fields.
– The RISC-V has a large but not infinite number of general purpose registers: there are 32 registers, one of which being always equal to zero. Register pressure can be a bottleneck for some operations, since memory traffic to and from the stack must use instructions that cannot be assumed to execute "in parallel" at little cost, because the CPU does not have extensive superscalar abilities. Compounding the effect is that fixed integer constants typically consume extra registers: a few opcodes can use an immediate value as an operand, but only up to 12 bits in size, and larger constants must be loaded from RAM into registers, or with multi-instruction sequences with a relatively large cost.

6.2 Implementing Binary Field Operations

6.2.1 Multiplications

In the absence of efficient carryless multiplication opcodes, the implementation of operations in $GF(2^{254})$ is challenging. Multiplications are, in practice, the most expensive.

In some heavily parallel usage contexts (e.g. when trying to break discrete logarithm), it can be worthwhile to split the work into individual bit operations, amenable to bitslicing; asymptotically fast algorithms such as Cantor-Kaltofen[9] can then provide good performance. However, practical usage for a curve such as GLS254 is far for allowing parallel execution of dozens of multiplications in the field.

For “normal” curve usage, the multiplication of two field elements is normally reduced to multiplication of smaller binary polynomials through one or several layers of Karatsuba-Ofman variants[19]. A multiplication of x and y in $GF(2^{254})$ is first reduced to three multiplication in $GF(2^{127})$:

$$\begin{aligned}(x_0 + ux_1)(y_0 + uy_1) &= x_0y_0 + u^2x_1y_1 + u(x_0y_1 + x_1y_0)\\ &= x_0y_0 + (u+1)x_1y_1 + u((x_0 + y_0)(x_1 + y_1) + x_0y_0 + x_1y_1)\\ &= x_0y_0 + x_1y_1 + u((x_0 + y_0)(x_1 + y_1) + x_0y_0)\end{aligned}$$

Each individual multiplication in $GF(2^{127})$ can then be broken into three $64 \times 64 \longrightarrow 128$ multiplications on binary polynomials. Optionally, each of these can be further broken into three $32 \times 32 \longrightarrow 64$ carryless multiplications, and so on. Compared to big integer multiplication, Karatsuba-Ofman is easier on binary polynomials because there is no carry propagation to worry about, and no enlargement of values when adding them together.

A popular method for computing binary polynomial multiplications is the use of lookup tables[7] to process a multiplier a few bits at a time. Such methods are unfortunately inherently not constant-time, unless they are degraded to bit-by-bit processing, which is simple but expensive.

Another option, which is increasingly attractive on modern processors with fast constant-time multipliers, is to leverage integer multiplications. The trick is to separate data bits with enough zeros to absorb and ultimately mask out the carries. Consider polynomials $c = \sum_i c_i z^i$ and $d = \sum_i d_i z^i$, and an integer $w > 0$. We can “mask out” bits in c and d, leaving only one every w bits to a non-zero value; then:

$$\left(\sum_i c_{wi}2^{wi}\right)\left(\sum_i d_{wi}2^{wi}\right) = \sum_i \left(\sum_{j=0}^{i} c_{wj}d_{wi-wj}\right)2^{wi}$$

If the number of non-zero bits in either of the masked versions of c or d is less than 2^w, then the inner sum fits over w bits and cannot overflow into the next slot. Moreover, the least significant bit of the sum of individual bit values (here the $c_{wj}d_{wi-wj}$ single-bit products) is equal to the exclusive or, i.e. the sum in $GF(2)$, of these bits. We can thus simply compute the product *as integers* of the masked c and d, and apply the same mask on the output, to get the product of the masked c and d *as binary polynomials*.

This method was rediscovered several times. The earliest description seems to be from Knuth[21], who proposed it as an exercise (and provided the solution) in *The Art of Computer Programming*[11]. Another early publication is from Fischer and Paterson in 1974[11].

With w-bit spacing, polynomials up to $w(2^w-1)$ bits can be processed in all generality. For 64-bit polynomials, we hit a small hurdle, which is that $w = 4$ is almost, but not quite sufficient: when multiplying two 64-bit polynomials with 4-bit spacing, the maximum Hamming weight of each masked operand is 16, and one of the sums of bits implied in the product can go up to 16, which does not fit over 4 bits and may spill out in the next bit. Using 5-bit spacing would raise the number of required integer multiplications from 16 to 25, and increase

[11] Exercise 4 of section 4.6, page 363, 401, or 420 of the first, second or third edition, respectively. Solution is on page 536, 617, or 671.

register pressure. This can however be fixed by using bit reversal. For binary polynomials of up to n bits (i.e. degree up to $n-1$), define the bit-reversal function rev_n:

$$\mathrm{rev}_n\left(\sum_{i=0}^{n-1} c_i z^i\right) = \sum_{i=0}^{n-1} c_{n-1-i} z^i$$

It then holds that, for any two binary polynomials c and d of degree less than n:

$$\mathrm{rev}_{2n-1}(cd) = \mathrm{rev}_n(c)\mathrm{rev}_n(d)$$

This means that we can obtain the higher bits of a product of binary polynomials by using the lower bits of the product of the bit-reversed polynomials. This is interesting for 64-bit operands, because while 4-bit spacing is not sufficient for the full product, it is enough for the low half of the 64-bit product (the spill-in-next-slot only happens near the start of the higher half). While obtaining low and high halves separately seems to double the number of integer multiplications, this is only apparent: as noted earlier, on our RISC-V test platform, the low and high halves of an integer product are obtained from two separate opcodes, and full-width $64 \times 64 \rightarrow 128$ multiplications already cost twice as much as truncated $64 \times 64 \rightarrow 64$ multiplications. Moreover, handling low and high halves separately reduces the number of used registers at any one time, which helps performance because register pressure appears to be a bottleneck for this code.

Bit reversal itself has some overhead. The Zbb extension includes the `rev8` instruction, which reverses the order of bytes; however, the complementary `brev8` instruction, which reverse the order of bits within each byte of its input operand, is part of the Zbkb extension and the SiFive-U74 does not have it. After an initial `rev8`, we must thus finish the bit reversal with some masking and shifts. This last step can be merged with the split of the values into 4-bit spacing masked polynomials.

This implementation method, with 64-bit multiplications, 4-bit spacing and bit reversal, yielded the fastest code we could design on our test platform for GLS254. We also considered a variant, which is only slightly slower in practice:

– Handle the highest bit of one $GF(2^{127})$ operand separately; split the rest into three 42-bit words. The other operand is split into three words of size 42, 42 and 44 bits[12].
– Use a Karatsuba-Ofman variant to reduce the product to six 42×44 products[13].
– Perform each 42×44 multiplication as two 21×44 multiplications. Note that each individual 21×44 multiplication can use 3-bit spacing (since the first operand, after masking, would have at most Hamming weight 7), and its result always fits on 64 bits, thus requiring only the `mul` opcode and using a single output register.

[12]In our code, we use a slightly redundant 128-bit representation of elements of $GF(2^{127})$, which we fully reduce only when encoding. Thus, in general, values have 128 bits. Here, we apply the strict reduction on the first operand, and then split out the top bit, while the second operand is not fully reduced.

[13]Toom-3 multiplication[37] could allow reducing the $GF(2^{127})$ product to only 5 smaller multiplications, but Bodrato[6] remarked that this necessarily implies a division by an "uneasy" constant, i.e. not z^i for some integer i. Moreover, at least one of the 5 smaller multiplications would have both operands of size 43 bits or more, which would be inconvenient and induce extra costs.

The implementation of this method is included (commented out) in our source code, in case it is useful on a different target architecture in which it would turn out to be faster than the 64-bit version with bit reversal and 4-bit spacing.

6.2.2 Squarings

Squaring is a much faster operations in binary fields. Integers with w-bit spacing can again be used; in our implementation, we use $32 \rightarrow 64$ squarings with 4-bit spacing. The Rust implementation of that operation is simple:

```
#[inline(always)]
fn expand_32(x: u64) -> u64 {
    let x0 = x & 0x1111111111111111;
    let x1 = x & 0x2222222222222222;
    let x2 = x & 0x4444444444444444;
    let x3 = x & 0x8888888888888888;

    #[inline(always)]
    fn sq_lo(x: u64) -> u64 {
        x.wrapping_mul(x)
    }

    let y0 = (sq_lo(x0) ^ sq_lo(x2)) & 0x1111111111111111;
    let y1 = (sq_lo(x1) ^ sq_lo(x3)) & 0x4444444444444444;

    y0 ^ y1
}
```

Here, we use the `sq_lo()` inline function to compute a truncated integer squaring (in Rust, truncation is detected and reported as an error in debug mode, unless a “wrapping multiplication” was explicitly used). A squaring in $GF(2^{127})$ only needs four invocations of this function; since we inline it, the loading of the masking constants in registers will presumably be shared among the four invocations.

6.3 Raw ECDH and High-Level Algorithm Performance

We list in table 5 the performance of our GLS254 implementation. For Ed25519, the comparison point is (again) our own code, which was especially optimized for that specific test platform[33]. Notably, classic Ed25519 implementations (e.g. curve25519-dalek, OpenSSL...) use 51-bit limbs in a way that maximizes parallelism, with five independent addition chains on 128-bit product outputs; such code was designed initially to map to the abilities of the Intel CPUs at that time, for which multiplications and carry propagation had relatively large latency. However, on our RISC-V platform, latency is not as much a problem as instruction throughput, and carry propagation is very expensive in that respect. The Ed25519 implementation in `crrl` (in the `src/backend/w64/gf255_m51.rs` file) splits product outputs and performs additions on low and high words separately. This specific optimization effort lowered `crrl`’s Ed25519 cost to about 158000 and 304000 cycles for signing and verifying, compared to 176000 and 444000 for curve25519-dalek, and 446000 and 1022000 for OpenSSL. There does not appear to be any RISC-V measurement in eBACS[14].

Operation	**GLS254**	**Ed25519**
raw ECDH, 1DT-3	775889	-
raw ECDH, 1DT-4	670156	-
raw ECDH, 1DT-5	678120	-
raw ECDH, 2DT-2	710039	-
raw ECDH, 2DT-3	679575	-
`decode`	14385	18898
`encode`	7466	16185
`mul`	666618	333775
`mulgen`	352999	128443
`hash_to_curve`	47549	-
`load_skey`	367947	152027
`sign`	377698	158023
`verify`	636398	304249

Table 5: Performance comparison of GLS254 and Ed25519 for high-level operations on SiFive-U74. Values are in clock cycles.

As shown in the table, GLS254 signatures on that specific platform appear to about 2.1x to 2.4x slower than Ed25519. This is certainly not *good* performance, but it is arguably not catastrophic either; for instance, the SSH client and server on the test machine use OpenSSL for cryptographic operations, in particular Ed25519, and OpenSSL's code for Ed25519 appear to be even slower than our own code for GLS254. The CPU can nominally run at 1.5 GHz[15] and at that frequency, 636398 cycles are less than half a millisecond.

7 ARM Cortex-M4 Implementation

7.1 Test Platform

We use an STM32F407 microcontroller, in an STM32F407G-DISC1 evaluation board. The CPU can be run at up to 168 MHz, though at that speed accesses to Flash memory (for constant data and code) cannot be performed without extra wait states; on-board caches are then used. To avoid cache effects, we run our test code at 24 MHz, and caches are disabled.

[14]The list of "computers" includes two RISC-V system using SiFive-U54 cores, but they do not appear in the list of signature-related results.

[15]For some unknown reason, our test machine seems to remain stuck at 1 GHz; this is probably some incompatibility between the specific Linux kernel version that we use and the board circuitry that allows dynamic frequency scaling. In any case, this does not impact our benchmarks, since we use the cycle counter register, and the whole computation fits in L1 cache.

Our code is written in assembly and C and is available there:

`https://github.com/pornin/gls254-cm4`

The assembly code implements the computations over $GF(2^{127})$ and $GF(2^{254})$, and the core elliptic curve operations (additions, sequences of doublings…). The C code supports scalars and high-level algorithms such as multiplication of a point by a scalar; that C code is compiled with optimization flags `-Os` (to optimize for binary size rather than speed), but this does not matter much for performance since almost all the computation time is spent in the assembly code, which is not impacted by compiler optimization levels.

Timings on an ARM Cortex-M4 are usually rather straightforward: the CPU can issue at most one instruction per cycle, and the output of the instruction is available at the next cycle. However, a few things are worth noting:

– Memory accesses have an extra latency. A load operation for an aligned 32-bit word takes 2 cycles. There are opcodes that can read n values in $n + 1$ cycles (`ldrd`, `ldm`, `pop`,…). Single-word stores can be performed in a single cycle, because the write is done in the background over the next cycle as well; but that holds only if the next instruction is *not* doing memory accesses, otherwise the CPU stalls for an extra cycle. Opcodes that write multiple words (`strd`, `stm`, `push`,…) need $n + 1$ cycles for n words.
– Our M4 is technically a "M4F", i.e. it has a floating-point unit. We do not use the floating-point operations (which are single-precision only), but we use the floating-point registers as a temporary data storage space which is somewhat faster than the stack (we can read two 32-bit words, or write two 32-bit words, into the floating-point registers in only 2 cycles). This yields an overall speed improvement of about 3%.
– The M4 core design does not include caches. However, that design from ARM is integrated by the hardware vendor into a microcontroller, which includes some RAM, Flash, and possibly caches and other elements. When the CPU issues read or write accesses, they go through an interconnection matrix that decides what element the access targets, and arbitrates between accesses from several sources (e.g. some peripheral might be able to do DMA independently of the M4 core). Caches may be applied at that level. Thus, any memory access may incur delays that depend on the accessed address, in ways which are *not* documented in the Cortex-M4 manual, since these delays come from circuit elements which are outside of the M4 core.
  For instance, it appears that some addresses in the Flash space trigger an extra load delay of a few cycles (they apparently correspond to the boundaries of Flash cells). Similarly, the M4 performs instruction fetches and data accesses on separate busses, but they both go through the interconnection matrix and may end up in the same space (e.g. the Flash cells), in which they may conflict and induce extra stalls. We noticed that, in particular, some code sequences with 32-bit instructions which are not 32-bit aligned may suffer from extra delays, even though that should not matter for the M4 core itself.

A noteworthy element of our code is that it does not use the register `r9`. The standard ABI for ARMv7-M (AAPCS32) states that the role of this register is platform-specific. It might be a plain data register, fully usable by application code, but it may also be reserved by the implementation for use by, for instance, position-independent code or thread-local storage. Notably, it might be reserved *at all times* in case it is to be used by asynchronously-invoked code (e.g. signal handlers), so that in some operating systems it must never be modified, and saving/restoring it in a given function is not sufficient[16]. In our test platform with no actual operating system, and disabled interrupts, we could certainly use `r9`, but, for maximum portability, we refrain from ever touching that register.

7.2 Raw ECDH and High-Level Algorithm Performance

Table 6 lists performance achieved by our implementation. We did not find any optimized implementation of Ed25519 for the M4; the best reported X25519 code on the M4 is from Lenngren[23], who obtained a cost of 476275 cycles on the M4F (548873 cycles if not using the floating-point registers[17]); X25519 would functionally be close to raw ECDH on GLS254. We estimate that an optimized Ed25519 implementation might hope for signing in about 350000 cycles, and verifying in 650000 cycles (using the Antipa *et al* optimization).

Operation	**Time (cycles)**
raw ECDH, 1DT-4	1673324
`decode`	21939
`mul`	1651923
`hash_to_curve`	92369
`load_skey`	1016515
`sign`	1034123
`verify`	1735470

Table 6: Performance of GLS254 on ARM Cortex-M4.

GLS254 signatures are about three times slower than Ed25519 on the M4. Again, this is not very good performance, but it is not atrocious either; at 24 MHz, signature verification takes about 72 milliseconds.

Since the M4 uses an in-order pipeline with very few latency effects, we can meaningfully define and measure the performance of individual operations. Our code uses, inside the assembly code, an internal ABI that differs from the standard ABI in that we do not save any register value (of course, appropriate register saving and restoring instructions are included at the interface between C and assembly). We count in the cost of a function all the instructions that constitute that function along the internal ABI, including the return statement (`bx lr` or `pop { pc }`) but excluding the call opcode itself (`bl`), which is accounted in the cost of the caller. Under these rules, we obtain the following elementary costs:

- Squaring in $GF(2^{127})$: 66 cycles.
- Multiplication in $GF(2^{127})$: 598 cycles.

[16]It seems that early incarnations of Apple's iOS had the requirement that `r9` never be modified, even transiently.

[17]Running Lenngren's code on our test platform yields a higher value, at 563310 cycles, possibly due to one of these undocumented alignment or Flash stall effects.

- Squaring in $GF(2^{254})$: 183 cycles.
- Multiplication in $GF(2^{254})$: 1875 cycles.
- Inversion in $GF(2^{254})$: 14554 cycles.

A complete raw ECDH execution with the 1DT-4 strategy implies a grand total of 728 multiplications, 693 squarings and 2 inversions in $GF(2^{254})$. These operations thus account for 81.57%, 7.58% and 1.74% of the total cost, respectively. Everything else, including the scalar splitting, constant-time window lookups, and general loop management, make up for 9.11% of the total cost only. This highlights how much the bottleneck is in the multiplications in $GF(2^{254})$.

In our code (file `gls254-cm4.s`), the macro `MM32` implements a sequence of 51 instructions which computes a $32 \times 32 \longrightarrow 64$ multiplication of binary polynomials. This sequence is executed 19872 times in raw ECDH, totalling to 60.6% of the total cost. This is the primary target for optimization. It uses 4-bit spacing with `umull` and `umlal` opcodes; one interesting point is that with 4-bit spacing, masked values have Hamming weight at most 8, and a maximum value of 8 can be obtained only in a single 4-bit slot in the 64-bit output. We can thus add *as integers* two such 64-bit outputs provided that they are such that the up-to-8 value is not the same slot for both. This is valuable, because `umlal` performs a multiplication and a 64-bit addition in one cycle in total, whereas a multiplication followed by a two-register XOR would require three cycles (one `umull` and two `eor`).

An additional important issue is that the `MM32` sequence consumes all registers (one register holds the mask constant `0x11111111` and is preserved; all other registers are modified). This implies in return a lot of data movement between registers and the stack (or the floating-point registers), which impacts performance.

For X25519, Lenngren achieves a multiplication in $GF(2^{255}-19)$ in as little as 173 cycles, more than 10 times faster than our code in $GF(2^{254})$; for squarings, the ratio is less dire (106 vs 183 cycles). The overall performance ratio is “only” 3.5x for raw ECDH thanks to the fact that GLS254 operations use fewer operations than Curve25519 for point doublings, and also thanks to the important savings due to the curve endomorphism.

8 Conclusion

We presented optimized formulas and implementations of the GLS254 curve. The salient points are the following:

- Our formulas support a convenient prime-order group abstraction, amenable to building arbitrary cryptographic protocols. In particular, the formulas are complete (no special case), can be implemented in constant-time code, and encoding/decoding is canonical in a verifiable way. There is no cofactor issue.
- The ease and safety resulting from the formula completeness does not come at a performance cost. In fact, the formulas make the whole code slightly *faster* than previous implementations.
- On small architectures without a carryless multiplication opcode, the performance is still lagging, compared with curves using prime-order fields, in particular the jq255 curves, and Curve25519. The important optimization target for such platforms is the implementation of binary polynomial multiplication. The achieved performance is not abysmal.

Elliptic curve cryptography is a slightly endangered field – the current mood is at the move to "post-quantum" cryptographic algorithms, which should (hopefully) resist breaking attempts by quantum computers; such machines, if they ever exist in a practical way, should break classical asymmetric algorithms such as ECDH and EC-based Schnorr signatures with relative ease. It is conceivable that attempts at building quantum computers never overcome their current technological hurdles. In any case, some deployed systems have put strong bets on quantum computers never becoming a reality (many blockchain-based systems, for instance, have committed assets measured in the trillions of US dollars to the idea that elliptic curve cryptography will remain unbreakable). GLS254 represents the current state-of-the-art in achievable performance of pre-quantum asymmetric cryptography; it is a good candidate for specialty pre-quantum deployments that need the high performance, and the short 48-byte signatures are convenient for bandwidth-constraind usage. As for the post-quantum world, the GLS254 performance can serve as a decent target to achieve; some lattice-based systems already rival such speeds, though with substantially larger public keys and signatures.

Acknowledgements

We thank Tjaden Hess and Samuel Neves for help with the Knuth and Fischer-Paterson references about carryless multiplication implementation tricks.

References

1. M. Aardal and D. Aranha, *2DT-GLS: Faster and exception-free scalar multiplication in the GLS254 binary curve*, https://eprint.iacr.org/2022/748
2. A. Antipa, D. Brown, R. Gallant, R. Lambert, R. Struik and S. Vanstone, *Accelerated Verification of ECDSA signatures*, Selected Areas in Cryptography - SAC 2005, Lecture Notes in Computer Science, vol. 3897, pp. 307-318, 2005.
3. 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.
4. D. Bernstein and T. Lange, *eBACS: ECRYPT Benchmarking of Cryptographic Systems*, https://bench.cr.yp.to (accessed 4 August 2022).
5. D. Bernstein, T. Lange and R. Rezaeian Farashahi, *Binary Edwards Curves*, Cryptographic Hardware and Embedded Systems - CHES 2008, Lecture Notes in Computer Science, vol. 5154, pp. 244-265, 2008.
6. M. Bodrato, *Towards Optimal Toom-Cook Multiplication for Univariate and Multivariate Polynomials in Characteristic 2 and 0*, International Workshop on the Arithmetic of Finite Fields - WAIFI 2007, Lecture Notes in Computer Science, vol. 4547, pp. 116-133, 2007.
7. R. Brent, P. Gaudry, E. Thomé and P. Zimmerman, *Faster multiplication in GF(2)[x]*, Algorithmic Number Theory - ANTS-VIII, Lecture Notes in Computer Science, vol. 5011, pp. 153-166, 2008.
8. É. 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.
9. D. Cantor and E. Kaltofen, *On fast multiplication of polynomials over arbitrary algebras*, Acta Informatica, vol. 28, issue 7, pp. 693-701, 1991.

10. I. Lovecruft and H. de Valence, *Dalek cryptography*,
`https://github.com/dalek-cryptography`
11. M. Fischer and M. Paterson, *String-Matching and Other Products*, MAC Technical Memorandum 41, Massachusetts Institute of Technology, 1974.
12. A. Fog, *The microarchitecture of Intel, AMD, and VIA CPUs*,
`https://www.agner.org/optimize/microarchitecture.pdf`
13. 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. 2139, pp. 190-200, 2001.
14. S. Galbraith, X. Lin and M. Scott, *Endomorphisms for Faster Elliptic Curve Cryptography on a Large Class of Curves*, Advances in Cryptology - EUROCRYPT 2009, Lecture Notes in Compute Science, vol. 5479, pp. 518-535, 2009.
15. 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.
16. M. Hamburg, *Accelerating AES with Vector Permute Instructions*, Cryptographic Hardware and Embedded Systems - CHES 2009, Lecture Notes in Computer Science, vol. 5747, pp. 18-32, 2009.
17. D. Hankerson, K. Karabina and A. Menezes, *Analyzing the Galbraith-Lin-Scott Point Multiplication Method for Elliptic Curves over Binary Fields*, IEEE Transactions on Computers, vol. 58, issue 10, pp. 1411-1420, 2009.
18. 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.
19. А. Карацуба and Ю. Офман, *Умножение Многозначных Чисел На Автоматах*, Доклады Академи и наук СССР, vol. 145, issue 2, pp. 293-294, 1962.
20. K. Kim and S. Kim, *A New Method for Speeding Up Arithmetic on Elliptic Curves over Binary Fields*,
`https://eprint.iacr.org/2007/181`
21. D. Knuth, *The Art of Computer Programming, volume 2: Seminumerical Algorithms*, Addison-Wesley, 1969.
22. E. Lenngren, *X25519 for ARM AArch64*,
`https://github.com/Emill/X25519-AArch64`
23. E. Lenngren, *X25519 for ARM Cortex-M4 and other ARM processors*,
`https://github.com/Emill/X25519-Cortex-M4`
24. G. Neven, N. P. Smart and B.Warinschi, *Hash function requirements for Schnorr signatures*, Journal of Mathematical Cryptology, vol. 3, issue 1, pp. 69-87, 2009.
25. Information Technology Laboratory, *Digital Signature Standard (DSS)*, National Institute of Standard and Technology, FIPS 186-5, 2023.
26. T. Oliveira, J. López-Hernández, D. Aranha and F. Rodríguez-Henríquez, *Two is the fastest prime: lambda coordinates for binary elliptic curves*, Journal of Cryptographic Engineering, vol. 4, issue 1, pp. 3-17, 2014.
27. T. Oliveira, J. López-Hernández, D. Aranha and F. Rodríguez-Henríquez, *Improving the performance of the GLS254*, Presented at the CHES 2016 Rump Session.
28. C. Petit and J.-J. Quisquater, *On Polynomial Systems Arising from a Weil Descent*, Advances in Cryptology - ASIACRYPT 2012, Lecture Notes in Computer Science, vol. 7658, pp. 451-466, 2012.
29. T. Pornin, *Optimized Lattice Basis Reduction In Dimension 2, and Fast Schnorr and EdDSA Signature Verification*,
`https://eprint.iacr.org/2020/454`
30. T. Pornin, *Double-Odd Elliptic Curves*,
`https://eprint.iacr.org/2020/1558`

31. T. Pornin, *Double-Odd Jacobi Quartic*, https://eprint.iacr.org/2022/1052
32. T. Pornin, *Efficient and Complete Formulas for Binary Curves*, https://eprint.iacr.org/2022/1325
33. T. Pornin, *On Multiplications with Unsaturated Limbs*, https://research.nccgroup.com/2023/09/18/on-multiplications-with-unsaturated-limbs/
34. C. Schnorr, *Efficient identification and signatures for smart cards*, Advances in Cryptology - CRYPTO '89, Lecture Notes in Computer Science, vol. 435, pp. 239-252, 1990.
35. 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.
36. E. Straus, *Addition chains of vectors (problem 5125)*, American Mathematical Monthly, vol. 70, pp. 806-808, 1964.
37. А. Тоом, *О сложности схемы из функциональных злементов, реализирующей умножение целых чисел*, Доклады Академи и наук СССР, vol. 153, issue 3, pp. 496-498, 1963.
38. 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.

A Curve Formulas

We recall here the relevant formulas, in pseudocode (Python syntax). Points are in $(X{:}S{:}Z{:}T)$ coordinates. Fixed constants are $a = u$ (a), $a^2 = u + 1$ (aa), $b = 1 + z^{54}$ (b) and $\sqrt{b} = 1 + z^{27}$ (sqrt_b).

A.1 Point Addition

Generic point addition, from [32] (section 5.1).

```
# Addition in the group, XSZT coordinates.
def point_add(P1, P2):
    (X1, S1, Z1, T1) = P1
    (X2, S2, Z2, T2) = P2
    X1X2 = X1*X2
    S1S2 = S1*S2
    Z1Z2 = Z1*Z2           # for mixed addition (Z2 == 1), Z1Z2 = Z1
    T1T2 = T1*T2
    D = (S1 + T1)*(S2 + T2)
    E = aa*T1T2            # aa == a^2; this disappears if a == 0
    F = X1X2**2
    G = Z1Z2**2
    X3 = D + S1S2
    S3 = sqrt_b*(G*(S1S2 + E) + F*(D + E))
    Z3 = sqrt_b*(F + G)
    T3 = X3*Z3
    return (X3, S3, Z3, T3)
```

Addition cost is $8\text{M}+2\text{S}+2\text{m}_\text{b}$ in all generality. For curves with $a = 0$, one multiplication is saved (we do not need T1T2 at all), but GLS254 uses $a = u$.

Mixed addition applies when one of the operands (e.g. P2) is in “scaled affine” coordinates, i.e. Z2 is equal to 1 (and T2 is then equal to X2). In that case, the computation of Z1Z2 is trivial, and the cost is $7\text{M} + 2\text{S} + 2\text{m}_\text{b}$.

Affine addition corresponds to the case of both operands being in scaled affine coordinates (Z1 and Z2 are both 1). This allows further optimizations: T1T2 is equal to X1X2, Z1Z2 is equal to 1, and G is equal to 1. The overall cost is then $5\text{M} + 1\text{S} + 2\text{m}_\text{b}$.

A.2 Point Negation and Subtraction

Negation is simply adding X to S. Subtraction is achieved by negation followed by addition.

```
# Negate a point.
def point_negate(P1):
    (X1, S1, Z1, T1) = P1
    return (X1, S1 + X1, Z1, T1)
```

A.3 Point Doubling

Successive point doublings, as described in section 2.3 of this paper. A single doubling is obtained by setting $n = 1$.

```
# n successive doublings, with n >= 1.
def point_xdouble(P1, n):
    (X1, S1, Z1, T1) = P1
    X = sqrt_b*X1
    T = sqrt_b*T1
    Z = Z1
    Y = sqrt_b*S1 + X**2 + a*T
    for i in range(0, n):
        D = (X + sqrt_b*Z)**2
        Z = T**2
        X = D**2
        E = D + T
        T = X*Z
        Y = (Y*(Y + E) + (a + b)*Z)**2 + (a + 1)*T
    X3 = sqrt_b*Z
    S3 = sqrt_b*(Y + (a + 1)*T + X**2)
    Z3 = X
    T3 = sqrt_b*T
    return (X3, S3, Z3, T3)
```

Cost for n doublings is $n(2\text{M} + 4\text{S} + 2\text{m}_b) + 2\text{S} + 6\text{m}_b$.

A.4 Comparisons

Two group elements can be compared with each other with cost 2M ([32], section 5.5)

```
# Point equality check
def point_equals(P1, P2):
    (X1, S1, Z1, T1) = P1
    (X2, S2, Z2, T2) = P2
    return S1*T2 == S2*T1
```

When testing whether a given point is the group neutral element N, it suffices to check whether its `X` coordinate is zero: N is the only such group element.

A.5 Encode/Decode

Encoding maps a group element to a field element w. The encoding is canonical: a given w can have a single antecedent, which the decoding process retrieves. This follows [32], section 4.3; the encoding of the neutral element can use $w_0 = 0$ (no other group element in GLS254 maps to zero).

```
# Point encoding.
def point_encode(P):
    (X, S, Z, T) = P
    if X == 0:
        return 0
    else:
        return sqrt(S / T)
```

Note: inversion in the field is normally implemented with a single inversion in $GF(2^{127})$, which itself uses Itoh-Tsujii[18]. A natural consequence of that algorithm is that the inverse of zero is (formally) defined as zero. In this specific case, this is convenient: it allows simply doing the division of S by T and not having any special case for the neutral N, since we want to obtain a zero result for N.

Decoding is somewhat more complicated:

```
# Point decoding.
def point_decode(w):
    if w == 0:
        return N
    d = w**2 + w + a
    e = b/(d**2)
    if trace(e) == 1:
        raise Exception("invalid input")
    f = qsolve(e)
    x = d*f
    if trace(x) != 0:
        x += d
    s = x*(w**2)
    X = x
    S = sqrt_b*s
    Z = sqrt_b
    T = sqrt_b*s
    return (X, S, Z, T)
```

A.6 Map to Curve

The following pseudocode implements the process described in section 3.1.

```
# Mapping 32 bytes into a group element.
def point_map(hh):
    # Set from specific bits in the input.
    h = copy_of(hh)
    h[127] = 0
    h[128] = 1
    h[129] = 0
    h[255] = 0
    c = bits_to_field(h)

    m1 = c
    m2 = c + z**2
    m3 = c + (c/z)**2
    e1 = b/m1
    e2 = b/m2
    e3 = b/m3
    if trace(e1) == 0:
        (m, e) = (m1, e1)
    elif trace(e2) == 0:
        (m, e) = (m2, e2)
    else:
        (m, e) = (m3, e3)
    d = sqrt(m)
    w = qsolve(d)
    w[0] = hh[128]
    f = qsolve(e)
    x = d*f
    if trace(x) == 1:
        x += d
    s = x*(w**2)
    return (x, sqrt_b*s, sqrt_b, sqrt_b*x)
```

B High-Level Specifications

High-level operations are key pair generation, public-key encoding and decoding, hash-to-curve, key exchange, and signatures. For all these operations, we reuse the same rules are the double-odd Jacobi quartics jq255e and jq255s[30,31], and specified as part of the C2SP project:

https://c2sp.org/jq255

The following adaptations are used:

– A group element is encoded into a field element with the process described in section A.5. A field element is then encoded into bytes using a little-endian convention:
  - An element $k \in GF(2^{127})$, with $k = \sum_{i=0}^{126} k_i z^i$, is encoded into 16 bytes v_0 to v_{15}, with $v_j = \sum_{i=0}^{7} k_{i+8j} 2^i$. Note that $0 \leq v_{15} < 128$ (the top bit of byte v_{15} is zero).

  - An element $x = x_0 + ux_1 \in GF(2^{254})$ is encoded into 32 bytes by concatenating the encodings of x_0 and x_1, in that order.
  - Upon decoding, it is verified that the top bits of bytes v_{15} and v_{31} of the input are both zero. If either of these bits is set, then the input is rejected as invalid.
- When generating or verifying a signature, the "challenge" value c is computed as a 16-byte sequence `cv`. In jq255e and jq255s, `cv` is interpreted as an integer c (in the 0 to $2^{128} - 1$ range) with the unsigned little-endian convention; for GLS254, `cv` is split into two 8-byte halves, which are both turned into integers (c_0 for the first half, c_1 for the second half) with the unsigned little-endian convention. The challenge value is then $c = c_0 + \mu c_1$, as described in section 3.2.

The rest of the jq255 specification is used "as is".