Efficient software implementation of curve-based cryptography

Summer School on real-world crypto and privacy

Diego F. Aranha

Department of Computer Science – Aarhus University

Contains joint work with Thomaz Oliveira, Julio López, Francisco Rodríguez-Henríquez, Marius A. Aardal

Agenda

1. A motivating example
2. Introduction to elliptic curves
3. **Binary fields** and their arithmetic
4. **Binary elliptic curves** with some notable examples
5. Scalar multiplication algorithms
6. Speed records :)

Definition

Techniques for making cryptographic systems more robust, efficient and secure in practice.

Efficiency in terms of resources is crucial!

Definition

Techniques for making cryptographic systems more robust, efficient and secure in practice.

Efficiency in terms of resources is crucial! **Improvements** come from:

Algorithmic advances

- New choices of parameters
- New mathematical representations
- Faster explicit formulas
- Faster high-level algorithms

Technological advances

- New instructions
- Vector instruction sets
- Multiple cores available
- Higher arithmetic density

Focus: Most efficient/elegant techniques arise from **interplay** between the two!

Figure 1: How can Alice send a message to Bob in a public channel?

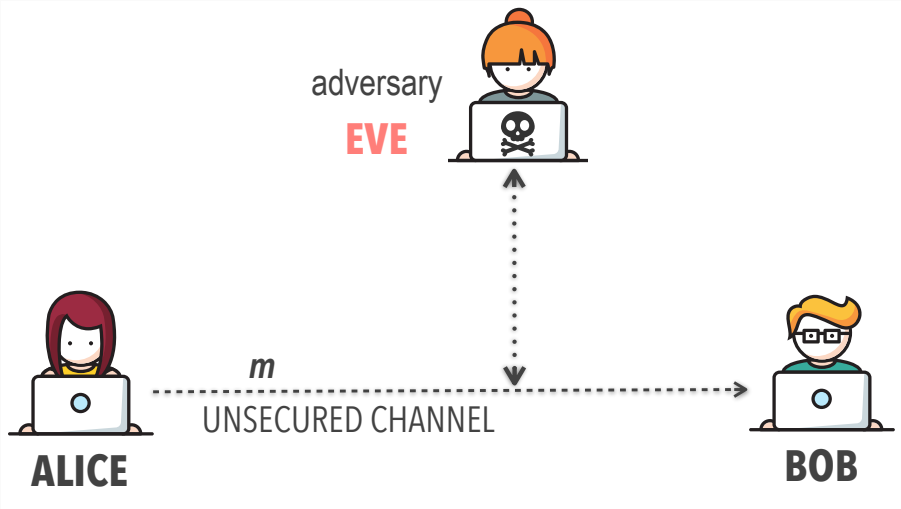
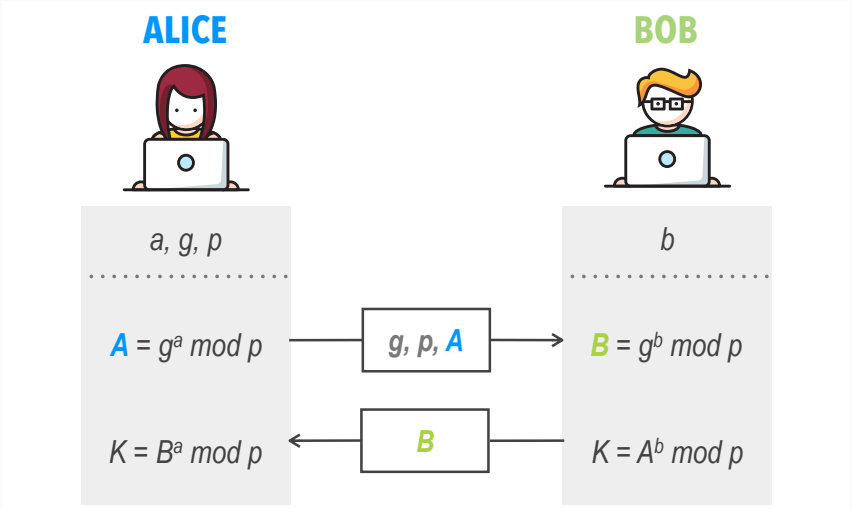


Figure 2: Group-based cryptography – Diffie-Hellman protocol!



Elliptic curves are an efficient way of instantiating cryptographic groups:

- Underlying problem **conjectured** to be fully exponential
- Small parameters, leading to **fast and compact** implementations
- **Standardized** in major protocols (TLS/SSL, SSH, Signal)

	Security level		
Algorithm	80	128	256
Finite Fields (or RSA)	1024	3072	15360
Elliptic curves (ECC)	**160**	**256**	**512**

Table 1: Key sizes in bits for public-key cryptosystems.

An **elliptic curve** is the set of solutions $(x, y) \in \mathbb{F}_q \times \mathbb{F}_q$ that satisfy the Weierstrass equation:

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

where $a_i \in \mathbb{F}_q$ with $\Delta \neq 0$, and a **point at infinity** ∞.

$E_1 : y^2 = x^3 + ax + b$ over $\mathbb{F}_p$ $\qquad$ $E_2 : y^2 + xy = x^3 + ax^2 + b$ over $\mathbb{F}_{2^m}$

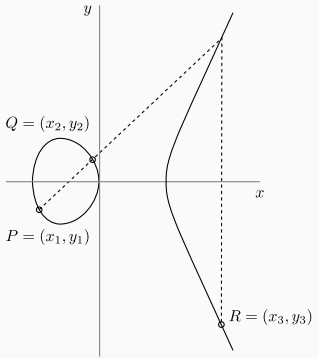


Figure 3: Point addition $R = P \oplus Q$

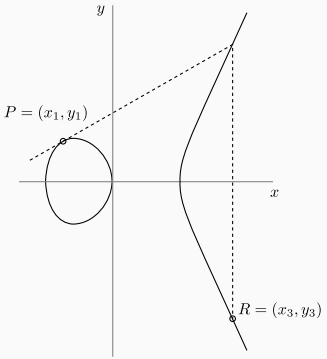


Figure 4: Point doubling $R = 2P$

Group law: Points form an additive group under the operation $\oplus$ (chord and tangent) of **order** n with ∞ as the identity.

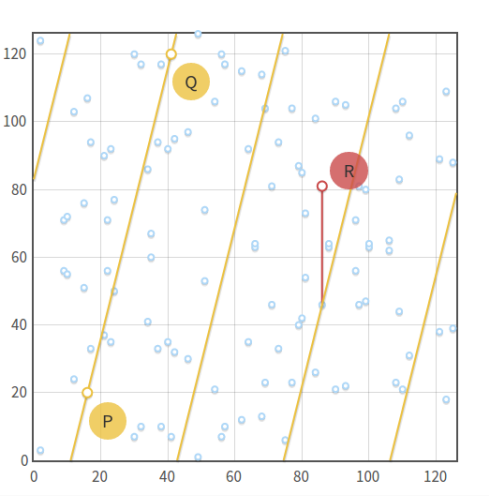


Figure 5: Point addition over the curve $y^2 = x^3 - x + 3 \pmod{127}$.

Let a point P in an elliptic curve and an integer k, the operation $[k]P$, called **scalar multiplication**, is defined as:

$$Q = [k]P = \underbrace{P \oplus P \oplus \ldots \oplus P}_{k-1 \text{ times}}.$$

Let a point P in an elliptic curve and an integer k, the operation $[k]P$, called **scalar multiplication**, is defined as:

$$Q = [k]P = \underbrace{P \oplus P \oplus \ldots \oplus P}_{k-1 \text{ times}}.$$

Assumption: Recover k given Q and P is **hard**! (ECDLP problem)

Generation of key pairs in ECC encode the problem: $P_{pub} = [sk]G$.

The order n of the curve is the number of points that satisfy the curve equation. The **Hasse condition** states that $n = q + 1 - t$, $|t| \leq 2\sqrt{q}$. The curve is **supersingular** when the field *characteristic* divides t.

The order n of the curve is the number of points that satisfy the curve equation. The **Hasse condition** states that $n = q + 1 - t$, $|t| \leq 2\sqrt{q}$. The curve is **supersingular** when the field *characteristic* divides t.

The **order** or P is the smallest integer r such that $[r]P = \infty$. We have $r|n$.

Security: We pick the order n to be a large prime or near-prime (small cofactor).

Scalar multiplication is critical for performance/security of ECC.

Algorithm 1 ECDSA signature generation

Input: Signing key $sk \in \mathbb{Z}_q$, message $m \in \{0,1\}^*$, group order n, base point G, and cryptographic hash function $H : \{0,1\}^* \to \mathbb{Z}_n$.
Output: A valid signature (r, s)

1: $k \leftarrow_{\$} \mathbb{Z}_n^*$
2: $R = (r_x, r_y) \leftarrow [k]G$
3: $r \leftarrow r_x \mod n$
4: $s \leftarrow (H(m) + r \cdot sk)/k \mod n$
5: **return** (r, s)

Critical: It should be implemented carefully to avoid leaking k.

Binary fields

A **finite field** $\mathbb{F}_{p^m}$ consists of all polynomials with coefficients in $\mathbb{Z}_p$, prime p, modulo an irreducible degree-m polynomial $f(z)$.

Prime p is the *characteristic* of the field and m is the *extension degree*.

A **binary field** $\mathbb{F}_{2^m}$ is the special case $p = 2$, formed by polynomials with **binary** coefficients.

Example – the Field $\mathbb{F}_{2^8}$

Irreducible polynomial:

$f(z) = z^8 + z^4 + z^3 + z + 1$ = `1 0001 1011`

Representation:

$a(z) = z^7 + z^3 + 1$ = `1000 1001` = `0x89`

$b(z) = z^6 + z^5 + z^2$ = `0110 0100` = `0x64`

Example – the Field $\mathbb{F}_{2^8}$

Irreducible polynomial:

$f(z) = z^8 + z^4 + z^3 + z + 1$ = `1 0001 1011`

Representation:

$a(z) = z^7 + z^3 + 1$ = `1000 1001` = `0x89`

$b(z) = z^6 + z^5 + z^2$ = `0110 0100` = `0x64`

Addition:

$a(z) + b(z) = z^7 + z^6 + z^5 + z^3 + z^2 + 1$ = `1110 1101` = `0xED`

Note: $a(z) + a(z) = 2 \cdot a(z) = 0, \forall a \in \mathbb{F}_{2^m}$

Irreducible polynomial:

$f(z) = z^8 + z^4 + z^3 + z + 1$ = `1 0001 1011`

Representation:

$a(z) = z^7 + z^3 + 1$ = `1000 1001` = `0x89`

$b(z) = z^6 + z^5 + z^2$ = `0110 0100` = `0x64`

Multiplication:

$a(z) \times b(z) = z^{13} + z^{12} + z^8 + z^6 + z^2 \bmod f(z) = z^7 + z^5 + z^4 + z^3 + z^2 + 1$ = `0xBD`

Multiplication by z:

$z \times b(z) = z^7 + z^6 + z^3$ = `1100 1000` = `0xC8` = $b \ll 1$

Binary fields

Binary fields ($\mathbb{F}_{2^m}$) are omnipresent in Cryptography:

- Efficient Curve-based Cryptography (NIST and other curves)
- Post-quantum Cryptography (BIKE, HQC and SABER key exchange protocols, Rainbow† and GeMSS signature scheme)
- Block ciphers (the previous field is used to define **substitution boxes** in AES)

Binary fields

Binary fields ($\mathbb{F}_{2^m}$) are omnipresent in Cryptography:

- Efficient Curve-based Cryptography (NIST and other curves)
- Post-quantum Cryptography (BIKE, HQC and SABER key exchange protocols, Rainbow† and GeMSS signature scheme)
- Block ciphers (the previous field is used to define **substitution boxes** in AES)

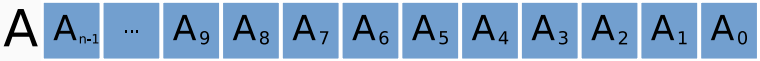
Many algorithms/optimizations already described in the literature:

- Arithmetic is faster than large **prime fields** in dedicated hardware (*normal* basis)
- Growing support in contemporary processors (`CLMUL` extension in Intel and `PMULL` in ARMv8 CPUs)

Note: Intuitively, binary fields should be faster than prime fields!

For a field $\mathbb{F}_{2^m}$, in practice we will employ [ALH10]:

- Irreducible polynomial: $f(z)$ (sparse – trinomial or pentanomial)
- Polynomial basis: $a(z) \in \mathbb{F}_{2^m} = \sum_{i=0}^{m-1} a_i z^i$.
- Software representation: vector of $n = \lceil m/w \rceil$ words in a w-bit processor.
- Graphical representation:



Arithmetic: addition is trivial, look next at squaring/multiplication/reduction/inversion.

$$a(z) = \sum_{i=0}^{m} a_i z^i = a_{m-1} + \cdots + a_2 z^2 + a_1 z + a_0$$

$$a(z)^2 = \sum_{i=0}^{m-1} a_i z^{2i} = a_{m-1} z^{2m-2} + \cdots + a_2 z^4 + a_1 z^2 + a_0$$

Example:

$$a(z) = (a_{m-1}, a_{m-2}, \ldots, a_2, a_1, a_0)$$

$$a(z)^2 = (a_{m-1}, 0, a_{m-2}, 0, \ldots, 0, a_2, 0, a_1, 0, a_0)$$

Since squaring is a linear operation (Frobenius):

$$(a(z) + b(z))^2 = a(z)^2 + b(z)^2.$$

Note: This property is also called *Freshman's dream*.

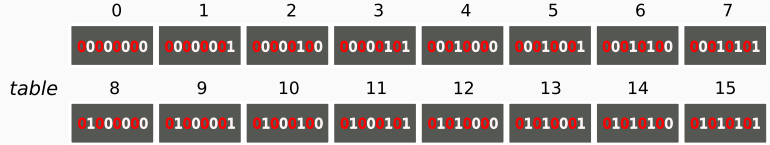
Since squaring is a linear operation (Frobenius):

$$(a(z) + b(z))^2 = a(z)^2 + b(z)^2.$$

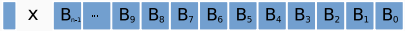
Note: This property is also called *Freshman's dream*.

Now we can compute $a(z)^2$ and $b(z)^2$ with a lookup table.

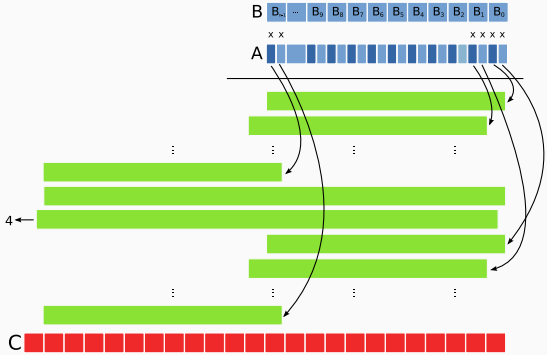
For $u = (u_3, u_2, u_1, u_0)$, use $table(u) = (0, u_3, 0, u_2, 0, u_1, 0, u_0)$:



We can compute $u \cdot b(z)$ using shifts and additions [LD00].



If $a(z)$ is divided into 4-bit polynomials, compute $a(z) \cdot b(z)$ by:



Algorithm by Fong et al. [FHLM04]:

$$\sqrt{a} = a^{2^{m-1}} = \sum_{i=0}^{m-1} \left(a_i z^i\right)^{2^m-1} = \sum_{i=0}^{m-1} a_i \left(z^{2^m-1}\right)^i$$

Algorithm by Fong et al. [FHLM04]:

$$\begin{aligned}\sqrt{a} = a^{2^{m-1}} &= \sum_{i=0}^{m-1} \left(a_i z^i\right)^{2^m-1} = \sum_{i=0}^{m-1} a_i \left(z^{2^m-1}\right)^i \\ &= \sum_{i \text{ even}} a_i z^{\frac{i}{2}} + \sqrt{z} \sum_{i \text{ odd}} a_i z^{\frac{i-1}{2}}\end{aligned}$$

Algorithm by Fong et al. [FHLM04]:

$$\begin{aligned}\sqrt{a} = a^{2^{m-1}} &= \sum_{i=0}^{m-1} \left(a_i z^i\right)^{2^m-1} = \sum_{i=0}^{m-1} a_i \left(z^{2^m-1}\right)^i \\ &= \sum_{i \text{ even}} a_i z^{\frac{i}{2}} + \sqrt{z} \sum_{i \text{ odd}} a_i z^{\frac{i-1}{2}} \\ &= a_{\text{even}} + \sqrt{z} \cdot a_{\text{odd}}\end{aligned}$$

Algorithm by Fong et al. [FHLM04]:

$$\begin{aligned}\sqrt{a} = a^{2^{m-1}} &= \sum_{i=0}^{m-1} \left(a_i z^i\right)^{2^m-1} = \sum_{i=0}^{m-1} a_i \left(z^{2^m-1}\right)^i \\ &= \sum_{i \text{ even}} a_i z^{\frac{i}{2}} + \sqrt{z} \sum_{i \text{ odd}} a_i z^{\frac{i-1}{2}} \\ &= a_{\text{even}} + \sqrt{z} \cdot a_{\text{odd}}\end{aligned}$$

We can decompose $a(z)$ into bits with odd/even indexes using tables.

Since square-root is also a linear operation:

$$\sqrt{a(z) + b(z)} = \sqrt{a(z)} + \sqrt{b(z)}$$

Note: For efficiency, choose $f(z)$ such that $\sqrt{z}$ is also sparse (faster *multiplication*).

We need to reduce squaring/multiplication results $c(z)$ in $\mathbb{F}_{2^m}$ modulo $f(z) = z^m + r(z)$.

We can process multiple bits at a time based on the observation:

$$\begin{aligned} c(z) &= c_{2m-2}z^{2m-2} + \cdots + c_m z^m + c_{m-1}z^{m-1} + \ldots c_1 z + c_0 \\ &= (c_{2m-2}z^{m-2} + \cdots + c_m)r(z) + c_{m-1}z^{m-1} + \ldots c_1 z + c_0. \end{aligned}$$

Note: We need to multiply by $r(z)$, that is why we use sparse $f(z)$.

In order to compute k consecutive squarings efficiently, precompute a table T of $16\lceil \frac{m}{4} \rceil$ field elements such that

$$T[j, i_0 + 2i_1 + 4i_2 + 8i_3] = (i_0 z^{4j} + i_1 z^{4j+1} + i_2 z^{4j+2} + i_3 z^{4j+3})^{2^k}$$

Then we can compute a^{2^k} as:

$$a^{2^k} = \sum_{j=0}^{\lceil \frac{m}{4} \rceil} T[j, \lfloor a/2^{4j} \rfloor \bmod 2^4].$$

For some $a(z)$, we want to compute $b(z)$ such that $a(z)b(z) = 1 \bmod f(z)$.

We have two main options:

- Implement Extended Euclidean Algorithm for polynomials to compute:

$$a(z)b(z) + f(z)c(z) = gcd(a(z), f(z)) = 1$$

  An efficient constant-time version can be found at [BY19].

- Implement Itoh-Tsuji algorithm with precomputed 2^i powers [IT88]:

$$a(z)^{-1} = a(z)^{2^m-2} = a(z)^{(2^{m-1}-1)2}$$

Binary curves

Let's start from the Weierstrass equation:

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6.$$

If $a_1 = 0$, the curve is *supersingular* (and in this case **dangerous**!).

Let's start from the Weierstrass equation:

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6.$$

If $a_1 = 0$, the curve is *supersingular* (and in this case **dangerous**!).

Otherwise, then the change of variables $x \to a_1^2x + (a_3/a_1), y \to a_1^3y + (a_1^2a_4 + a_3^2)/a_1^3$ leads to the equation of the **ordinary** curve:

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

When $a \in \{0, 1\}$ and $b = 1$, the curve is called a **Koblitz curve**.

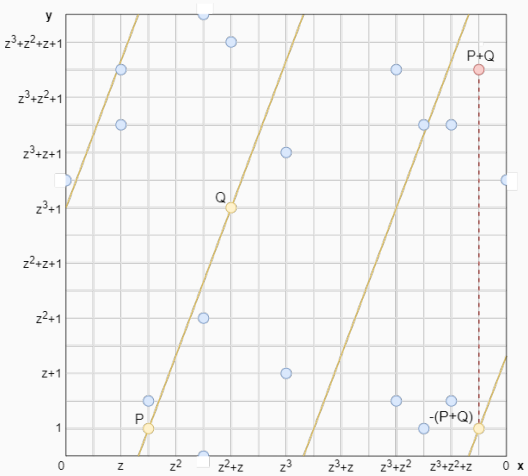


Figure 6: $E : y^2 + xy = x^3 + x^2 + z^3$ over $\mathbb{F}_{2^4}$ with reduction polynomial $f(z) = z^4 + z + 1$.

We can obtain $2P = (x_3, y_3)$ for $P = (x_1, y_1)$ with $\lambda = x_1 + y_1/x_1$ as:

$$x_3 = \lambda^2 + \lambda + b = x_1^2 + b/x_1^2, \qquad y_3 = x_1^2 + \lambda(x_3 + 1)$$

We can obtain $2P = (x_3, y_3)$ for $P = (x_1, y_1)$ with $\lambda = x_1 + y_1/x_1$ as:

$$x_3 = \lambda^2 + \lambda + b = x_1^2 + b/x_1^2, \qquad y_3 = x_1^2 + \lambda(x_3 + 1)$$

We can obtain $P + Q = (x_3, y_3)$ for $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ with $\lambda = (y_1 + y_2)/(x_1 + x_2)$:

$$x_3 = \lambda^2 + \lambda + x_1 + x_2 + a, \qquad y_3 = \lambda(x_1 + x_3) + x_3 + y_1$$

We can obtain $2P = (x_3, y_3)$ for $P = (x_1, y_1)$ with $\lambda = x_1 + y_1/x_1$ as:

$$x_3 = \lambda^2 + \lambda + b = x_1^2 + b/x_1^2, \qquad y_3 = x_1^2 + \lambda(x_3 + 1)$$

We can obtain $P + Q = (x_3, y_3)$ for $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ with $\lambda = (y_1 + y_2)/(x_1 + x_2)$:

$$x_3 = \lambda^2 + \lambda + x_1 + x_2 + a, \qquad y_3 = \lambda(x_1 + x_3) + x_3 + y_1$$

Coordinate system: For efficiency, we represent a point in **affine coordinates** (x, y) using **projective coordinates** (X, Y, Z) such that $x = X/Z^c$ and $y = Y/Z^d$.

Most popular are homogeneous projective ($c = d = 1$), Jacobian ($c = 2, d = 3$) and López-Dahab ($c = 1, d = 2$).

Algorithm 2 Double-and-add scalar multiplication

Input: $k = \sum_{i=0}^{t-1} k_i 2^i$, $P \in E(\mathbb{F}_{2^m})$ of order r.
Output: $[k]P$.

1: $Q \leftarrow \infty$
2: **for** $i = t - 1$ **downto** 0 **do**
3: $\quad Q \leftarrow 2Q$
4: $\quad$ **if** $k_i = 1$ **then**
5: $\quad\quad Q \leftarrow Q \oplus P$
6: $\quad$ **end if**
7: **end for**
8: **return** Q

Koblitz curves have the **Frobenius automorphism** τ on $E(\mathbb{F}_{2^m})$ given by $\tau(x, y) = (x^2, y^2)$.

The characteristic polynomial of the Frobenius is $\tau^2 - \mu\tau + 2$, for $\mu = (-1)^{1-a}$.

It is possible to transform a scalar $k \in \mathbb{Z}_n$ in the ring $\mathbb{Z}[\tau]$ for $\tau = \frac{\mu \pm \sqrt{-7}}{2}$, and point doublings can be replaced by applications of τ [Sol00].

Important: τ is faster to evaluate than point doubling!

Algorithm 3 Left-to-right τ-and-add scalar multiplication

Input: $k \in \mathbb{Z}$, $P \in E(\mathbb{F}_{2^m})$ of order r.
Output: $[k]P$.

1: Obtain the representation $TNAF(k) = \sum_{i=0}^{t-1} u_i \tau^i$, where $u_i \in \{-1, 0, 1\}$
2: $Q \leftarrow \infty$
3: **for** $i = t - 1$ **downto** 0 **do**
4: $\quad Q \leftarrow \tau Q$
5: $\quad$ **if** $u_i = 1$ **then**
6: $\quad\quad Q \leftarrow Q \oplus P$
7: $\quad$ **else if** $u_i = -1$ **then**
8: $\quad\quad Q \leftarrow Q \oplus (-P)$
9: $\quad$ **end if**
10: **end for**
11: **return** Q

ECC standards were pioneered by Certicom, as part of the SECG curves (`https://www.secg.org/sec2-v2.pdf`).

NIST initially standardized a number of such elliptic curves at different security levels:

- **Prime curves**: P192, P224, P256, P384, P521
- **Binary curves**: $m = 163, 233, 283, 409, 571$ (one random and one Koblitz curve)

ECC standards were pioneered by Certicom, as part of the SECG curves (`https://www.secg.org/sec2-v2.pdf`).

NIST initially standardized a number of such elliptic curves at different security levels:

- **Prime curves**: P192, P224, P256, P384, P521
- **Binary curves**: $m = 163, 233, 283, 409, 571$ (one random and one Koblitz curve)

Security: To prevent *descent attacks* (Weil, GHS) in ECC, choose m to be prime.

Famous examples of fast prime curves are **Curve25519** [Ber06] and Four$\mathbb{Q}$ [CL15]. Another SECG curve that became famous (not selected by NIST) is `secp256k1` used in Bitcoin.

Security aspects

Prime curves are certainly more **conservative**, with a *boring* security track record.

What is known about the security of binary curves:

- ECDLP **is still hard** for concrete parameters.
- **Quantum attacks** require less *qbits* (faster quantum gates)
- Asymptotically, there are faster than generic attacks (under some heuristic assumptions) that require **impractical storage**

Security aspects

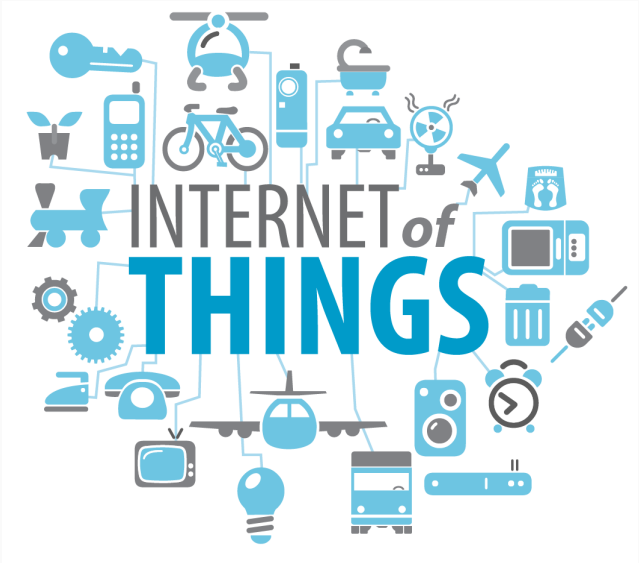
Prime curves are certainly more **conservative**, with a *boring* security track record.

What is known about the security of binary curves:

- ECDLP **is still hard** for concrete parameters.
- **Quantum attacks** require less *qbits* (faster quantum gates)
- Asymptotically, there are faster than generic attacks (under some heuristic assumptions) that require **impractical storage**

However, the third above and low industry adoption was sufficient for NIST to **deprecate all binary curves** in its revision towards postquantum cryptography (NIST SP800-186).

Note: There is still interest in research (GLS254) and bleeding-edge applications.


INTERNET of
THINGS

Side-channel attacks gather leakage during the **execution** of an implementation of a cryptographic algorithm to compromise its security properties (increasing intrusiveness):

- **Timing:** variance in execution time
- **Power:** variance in energy consumption
- **Electromagnetic and acoustic:** emanations from a device
- **Reminiscence:** recovery of stored data from RAM or Flash
- **Fault injection:** corruption of execution flow

Side-channel attacks gather leakage during the **execution** of an implementation of a cryptographic algorithm to compromise its security properties (increasing intrusiveness):

- **Timing:** variance in execution time
- **Power:** variance in energy consumption
- **Electromagnetic and acoustic:** emanations from a device
- **Reminiscence:** recovery of stored data from RAM or Flash
- **Fault injection:** corruption of execution flow

Defense: Employ **regular** implementation in energy/instruction flow:

- No branches depending on secret data (no exceptional cases)
- No memory access depending on secret data

Modern CPUs have instructions (`cflush`) that can reveal **secrets** through cache data eviction, i.e. Flush+Reload [YF14].

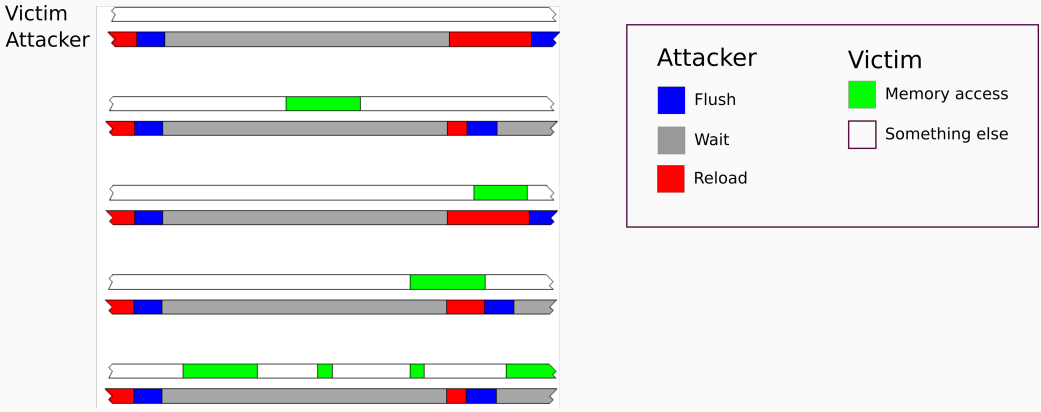


Figure 7: Picture from the original paper at [YF14]

Implementing the GLS254 curve

Motivation

For years, binary curves were considered **inefficient** for software implementation, mainly because of the absence of a carry-less native multiplier.

For years, binary curves were considered **inefficient** for software implementation, mainly because of the absence of a carry-less native multiplier.

However, in the end of the last decade we had significant **technological** and **algorithmic** improvements.

For years, binary curves were considered **inefficient** for software implementation, mainly because of the absence of a carry-less native multiplier.

However, in the end of the last decade we had significant **technological** and **algorithmic** improvements.

Technological

- A native carry-less multiplier (`PCLMULQDQ`) was released in 2010, by Intel, in its Westmere family and in 2011, by AMD, in the Bulldozer architecture.
- Also, in 2010, ARM announced the NEON instruction set, which included the carry-less instruction `VMUL.P` (8 simultaneous 8-bit multiplications) and later `PMULL` in ARMv8.

For years, binary curves were considered **inefficient** for software implementation, mainly because of the absence of a carry-less native multiplier.

However, in the end of the last decade we had significant **technological** and **algorithmic** improvements.

Technological

- A native carry-less multiplier (`PCLMULQDQ`) was released in 2010, by Intel, in its Westmere family and in 2011, by AMD, in the Bulldozer architecture.
- Also, in 2010, ARM announced the NEON instruction set, which included the carry-less instruction `VMUL.P` (8 simultaneous 8-bit multiplications) and later `PMULL` in ARMv8.

Algorithmic

- In 2009, Galbraith, Lin, Scott constructed efficiently computable endomorphisms for a large family of elliptic curves defined over $\mathbb{F}_{q^2}$.
- In the same year, Hankerson, Karabina and Menezes studied the application of the GLS curves over binary fields.

For years, binary curves were considered **inefficient** for software implementation, mainly because of the absence of a carry-less native multiplier.

However, in the end of the last decade we had significant **technological** and **algorithmic** improvements.

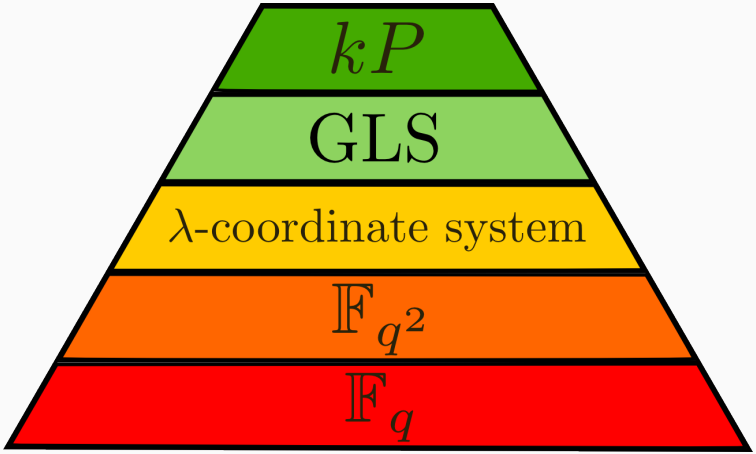
Technological

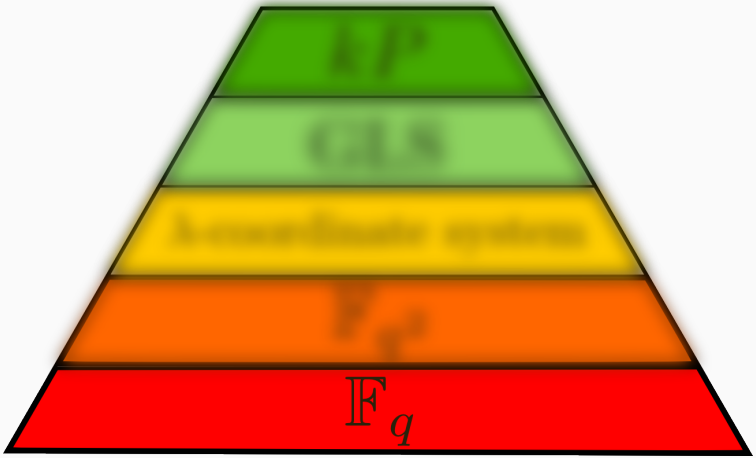
- A native carry-less multiplier (`PCLMULQDQ`) was released in 2010, by Intel, in its Westmere family and in 2011, by AMD, in the Bulldozer architecture.
- Also, in 2010, ARM announced the NEON instruction set, which included the carry-less instruction `VMUL.P` (8 simultaneous 8-bit multiplications) and later `PMULL` in ARMv8.

Algorithmic

- In 2009, Galbraith, Lin, Scott constructed efficiently computable endomorphisms for a large family of elliptic curves defined over $\mathbb{F}_{q^2}$.
- In the same year, Hankerson, Karabina and Menezes studied the application of the GLS curves over binary fields.

As a consequence, binary elliptic curves became highly competitive with prime curves.


kP
GLS
λ-coordinate system
$\mathbb{F}_{q^2}$
$\mathbb{F}_q$



Base field arithmetic

$\mathbb{F}_q$: Binary extension field of order $q = 2^m$.
Constructed by a polynomial $f(x)$ of degree m irreducible over $\mathbb{F}_2$.

$\mathbb{F}_{q^2}$: Quadratic extension of a binary field.
Constructed by a polynomial $g(u)$ of degree 2 irreducible over $\mathbb{F}_q$.

The elements of $\mathbb{F}_q$ are represented in polynomial base as $\sum_{i=0}^{m-1} a_i x^i$, with $a_i \in \mathbb{F}_2$.

The elements of $\mathbb{F}_{q^2}$ are represented as $a + bu$, with $a, b \in \mathbb{F}_q$.

$\mathbb{F}_q$: Binary extension field of order $q = 2^m$.
Constructed by a polynomial $f(x)$ of degree m irreducible over $\mathbb{F}_2$.

$\mathbb{F}_{q^2}$: Quadratic extension of a binary field.
Constructed by a polynomial $g(u)$ of degree 2 irreducible over $\mathbb{F}_q$.

The elements of $\mathbb{F}_q$ are represented in polynomial base as $\sum_{i=0}^{m-1} a_i x^i$, with $a_i \in \mathbb{F}_2$.

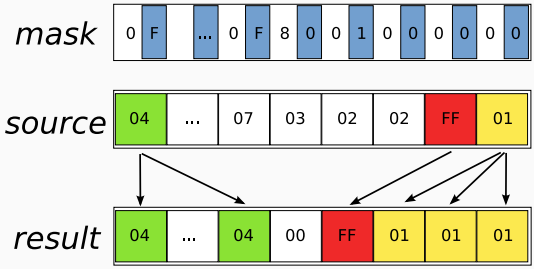
The elements of $\mathbb{F}_{q^2}$ are represented as $a + bu$, with $a, b \in \mathbb{F}_q$.

A **careful** selection of $f(x)$ and $g(u)$ is important for an efficient implementation.

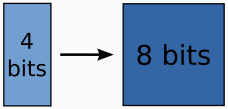
Our choices:

- $\mathbb{F}_{2^{127}} = \mathbb{F}_2[x]/(x^{127} + x^{63} + 1)$
- $\mathbb{F}_{2^{254}} = \mathbb{F}_{2^{127}}[u]/(u^2 + u + 1)$

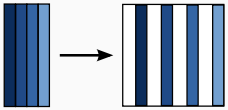
PSHUFB instruction (`_mm_shuffle_epi8`):



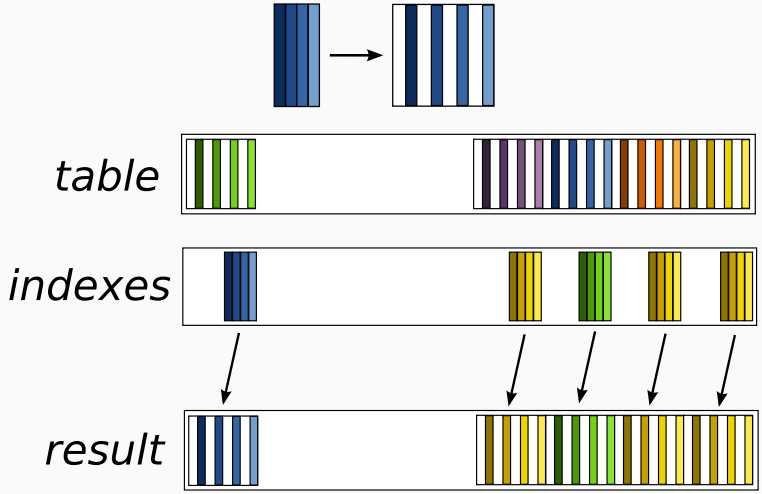
Real power: We can implement **in parallel** any function:



Example: Bit manipulation



Example: Bit manipulation



Given $a, b \in \mathbb{F}_q$, calculate $c = a \cdot b \mod f(x)$.

Given $a, b \in \mathbb{F}_q$, calculate $c = a \cdot b \mod f(x)$.

The 127-bit elements in $\mathbb{F}_{2^{127}}$ can be packed into two 64-bit words.

Given $a, b \in \mathbb{F}_q$, calculate $c = a \cdot b \mod f(x)$.

The 127-bit elements in $\mathbb{F}_{2^{127}}$ can be packed into two 64-bit words.

Polynomial multiplication can be performed using the Karatsuba method.

$$\begin{aligned} a \cdot b &= (a_1 x^{64} + a_0) \cdot (b_1 x^{64} + b_0) \\ &= (a_1 \cdot b_1) x^{128} + [(a_1 + a_0) \cdot (b_1 + b_0) + a_1 \cdot b_1 + a_0 \cdot b_0] x^{64} + a_0 \cdot b_0 \end{aligned}$$

Given $a, b \in \mathbb{F}_q$, calculate $c = a \cdot b \mod f(x)$.

The 127-bit elements in $\mathbb{F}_{2^{127}}$ can be packed into two 64-bit words.

Polynomial multiplication can be performed using the Karatsuba method.

$$\begin{aligned} a \cdot b &= (a_1 x^{64} + a_0) \cdot (b_1 x^{64} + b_0) \\ &= (a_1 \cdot b_1) x^{128} + [(a_1 + a_0) \cdot (b_1 + b_0) + a_1 \cdot b_1 + a_0 \cdot b_0] x^{64} + a_0 \cdot b_0 \end{aligned}$$

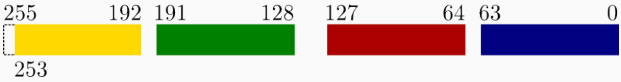
In $\mathbb{F}_{2^{127}}$, this operation can be implemented with three carry-less multiplication instructions.

Modular reduction can be efficiently computed due to the special form of the trinomial $f(x) = x^{127} + x^{63} + 1$.

Modular reduction can be efficiently computed due to the special form of the trinomial $f(x) = x^{127} + x^{63} + 1$.

After one polynomial multiplication in $\mathbb{F}_{2^{127}}$ we have a polynomial of degree 253.

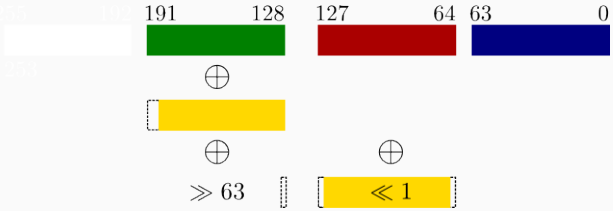
$$x^{192+i} \equiv x^{128+i} + x^{65+i},\ i \in \{0, \ldots, 61\}$$



Modular reduction can be efficiently computed due to the special form of the trinomial $f(x) = x^{127} + x^{63} + 1$.

After one polynomial multiplication in $\mathbb{F}_{2^{127}}$ we have a polynomial of degree 253.

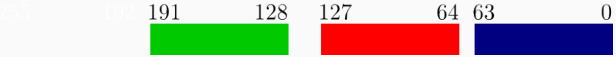
$$x^{192+i} \equiv x^{128+i} + x^{65+i},\ i \in \{0, \ldots, 61\}$$



Modular reduction can be efficiently computed due to the special form of the trinomial $f(x) = x^{127} + x^{63} + 1$.

After one polynomial multiplication in $\mathbb{F}_{2^{127}}$ we have a polynomial of degree 253.

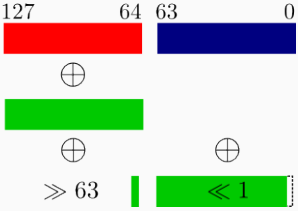
$$x^{128+i} \equiv x^{64+i} + x^{1+i},\ i \in \{0, \ldots, 63\}$$



Modular reduction can be efficiently computed due to the special form of the trinomial $f(x) = x^{127} + x^{63} + 1$.

After one polynomial multiplication in $\mathbb{F}_{2^{127}}$ we have a polynomial of degree 253.

$$x^{128+i} \equiv x^{64+i} + x^{1+i},\ i \in \{0, \ldots, 63\}$$



Modular reduction can be efficiently computed due to the special form of the trinomial $f(x) = x^{127} + x^{63} + 1$.

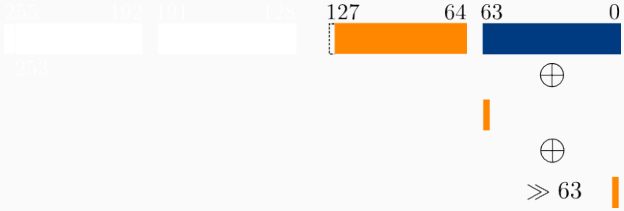
After one polynomial multiplication in $\mathbb{F}_{2^{127}}$ we have a polynomial of degree 253.

$$x^{127} \equiv x^{63} + 1$$

Modular reduction can be efficiently computed due to the special form of the trinomial $f(x) = x^{127} + x^{63} + 1$.

After one polynomial multiplication in $\mathbb{F}_{2^{127}}$ we have a polynomial of degree 253.

$$x^{127} \equiv x^{63} + 1$$



Modular reduction can be efficiently computed due to the special form of the trinomial $f(x) = x^{127} + x^{63} + 1$.

After one polynomial multiplication in $\mathbb{F}_{2^{127}}$ we have a polynomial of degree 253.

$$x^{127} = x^{63} + 1$$

Modular reduction can be efficiently computed due to the special form of the trinomial $f(x) = x^{127} + x^{63} + 1$.

After one polynomial multiplication in $\mathbb{F}_{2^{127}}$ we have a polynomial of degree 253.

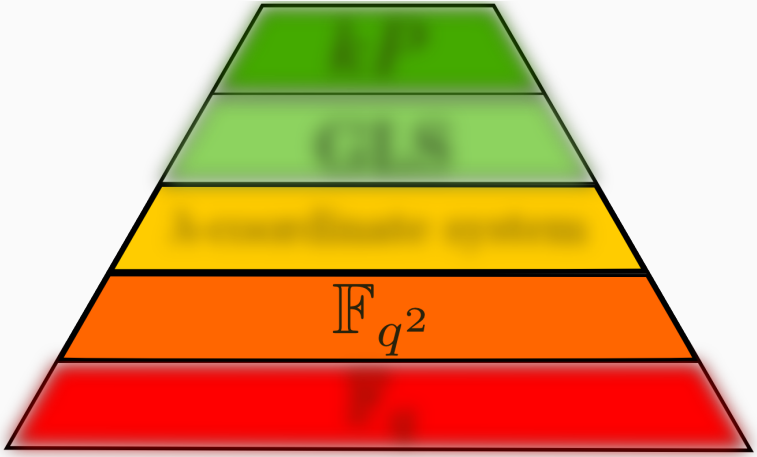
The reduction can be performed in eleven instructions.

Modular reduction can be efficiently computed due to the special form of the trinomial $f(x) = x^{127} + x^{63} + 1$.

After one polynomial multiplication in $\mathbb{F}_{2^{127}}$ we have a polynomial of degree 253.

After squaring: Taking advantage of the sparsity of the polynomial square operation, the result of this operation can be reduced using just six instructions.

Improvement: We actually consider arithmetic modulo $z \cdot f(z)$ and leave 128-bit results **unreduced**.



Quadratic extension field arithmetic

Taking advantage of the irreducible polynomial $g(u) = u^2 + u + 1$, all the field arithmetic in the quadratic extension $\mathbb{F}_{q^2}$ can be performed efficiently.

Taking advantage of the irreducible polynomial $g(u) = u^2 + u + 1$, all the field arithmetic in the quadratic extension $\mathbb{F}_{q^2}$ can be performed efficiently.

Multiplication:
$a \cdot b = (a_0 + a_1 u) \cdot (b_0 + b_1 u) = (a_0 \cdot b_0 + a_1 \cdot b_1) + ((a_0 + a_1) \cdot (b_0 + b_1) + a_0 \cdot b_0)u$ with $a_0, a_1, b_0, b_1 \in \mathbb{F}_q$.

Taking advantage of the irreducible polynomial $g(u) = u^2 + u + 1$, all the field arithmetic in the quadratic extension $\mathbb{F}_{q^2}$ can be performed efficiently.

Multiplication:
$a \cdot b = (a_0 + a_1 u) \cdot (b_0 + b_1 u) = (a_0 \cdot b_0 + a_1 \cdot b_1) + ((a_0 + a_1) \cdot (b_0 + b_1) + a_0 \cdot b_0)u$ with $a_0, a_1, b_0, b_1 \in \mathbb{F}_q$.

Squaring: $a^2 = (a_0 + a_1 u)^2 = a_0^2 + a_1^2(u + 1) = a_0^2 + a_1^2 + a_1^2 u$.

Taking advantage of the irreducible polynomial $g(u) = u^2 + u + 1$, all the field arithmetic in the quadratic extension $\mathbb{F}_{q^2}$ can be performed efficiently.

Multiplication:
$a \cdot b = (a_0 + a_1 u) \cdot (b_0 + b_1 u) = (a_0 \cdot b_0 + a_1 \cdot b_1) + ((a_0 + a_1) \cdot (b_0 + b_1) + a_0 \cdot b_0)u$ with $a_0, a_1, b_0, b_1 \in \mathbb{F}_q$.

Squaring: $a^2 = (a_0 + a_1 u)^2 = a_0^2 + a_1^2(u + 1) = a_0^2 + a_1^2 + a_1^2 u$.

Square-root: $\sqrt{a} = \sqrt{a_0 + a_1 u} = \sqrt{a_0 + a_1} + \sqrt{a_1}u$.

Taking advantage of the irreducible polynomial $g(u) = u^2 + u + 1$, all the field arithmetic in the quadratic extension $\mathbb{F}_{q^2}$ can be performed efficiently.

Multiplication:
$a \cdot b = (a_0 + a_1 u) \cdot (b_0 + b_1 u) = (a_0 \cdot b_0 + a_1 \cdot b_1) + ((a_0 + a_1) \cdot (b_0 + b_1) + a_0 \cdot b_0)u$ with $a_0, a_1, b_0, b_1 \in \mathbb{F}_q$.

Squaring: $a^2 = (a_0 + a_1 u)^2 = a_0^2 + a_1^2(u + 1) = a_0^2 + a_1^2 + a_1^2 u$.

Square-root: $\sqrt{a} = \sqrt{a_0 + a_1 u} = \sqrt{a_0 + a_1} + \sqrt{a_1}u$.

Inverse: $a \cdot c = (a_0 + a_1 u) \cdot (c_0 + c_1 u) = 1$. Compute $t = a_0 \cdot a_1 + {a_0}^2 + {a_1}^2$, $c_0 = (a_0 + a_1) \cdot t^{-1}$ and $c_1 = a_1 \cdot t^{-1}$.

$\mathbb{F}_{q^2}$	Multiplication	Square-Root	Squaring	Inversion
$\mathbb{F}_q$	3 mult + 4 add	2 sqrt + add	2 sqr + add	inv + 3 mult + 2 sqr + 3 add

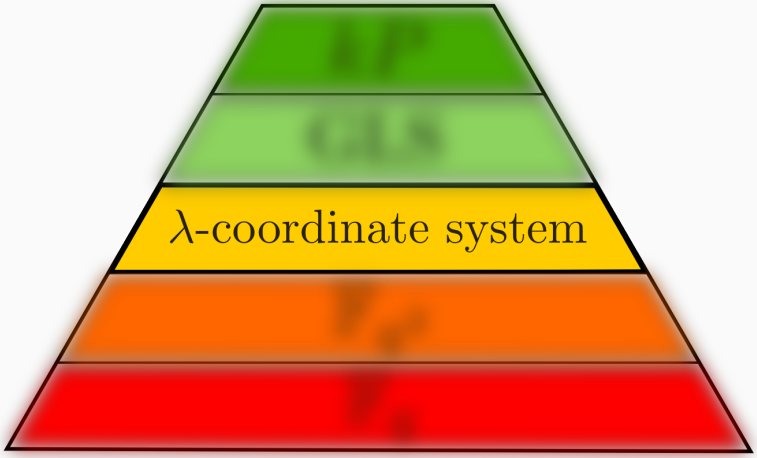
Table 2: Timings on an Intel Sandy Bridge for the field arithmetic.

Field operation	$\mathbb{F}_q$		$\mathbb{F}_{q^2}$	
	cycles	op/M[1]	cycles	op/M
Multiplication	42	1.00	94	1.00
Mod. Reduction[2]	6	0.14	11	0.12
Square root	8	0.19	15	0.16
Squaring	9	0.21	13	0.14
Multi-Squaring	55	1.31	n/a[3]	n/a
Inversion	2295	54.64	2700	29.34
Inversion (table)	765	18.21	969	10.30

[1] Ratio to multiplication.

[2] This cost is included in the timings of all operations that require modular reduction.

[3] Multi-Squaring is used for the table-based inversion.



Elliptic curve arithmetic

Let $E_{a,b}/\mathbb{F}_q : y^2 + xy = x^3 + ax^2 + b$ be a Weierstrass binary ordinary elliptic curve over $\mathbb{F}_q$.

Let $E_{a,b}/\mathbb{F}_q : y^2 + xy = x^3 + ax^2 + b$ be a Weierstrass binary ordinary elliptic curve over $\mathbb{F}_q$.

λ-affine representation: Given a point $P = (x, y) \in E_{a,b}(\mathbb{F}_q)$ with $x \neq 0$, represent $P = (x, \lambda)$, where $\lambda = x + \frac{y}{x}$:

$$(\lambda^2 + \lambda + a) \cdot x^2 = x^4 + b.$$

Let $E_{a,b}/\mathbb{F}_q : y^2 + xy = x^3 + ax^2 + b$ be a Weierstrass binary ordinary elliptic curve over $\mathbb{F}_q$.

λ-affine representation: Given a point $P = (x, y) \in E_{a,b}(\mathbb{F}_q)$ with $x \neq 0$, represent $P = (x, \lambda)$, where $\lambda = x + \frac{y}{x}$:

$$(\lambda^2 + \lambda + a) \cdot x^2 = x^4 + b.$$

We must have efficient formulas for point **addition** and doubling.

Let $E_{a,b}/\mathbb{F}_q : y^2 + xy = x^3 + ax^2 + b$ be a Weierstrass binary ordinary elliptic curve over $\mathbb{F}_q$.

λ-affine representation: Given a point $P = (x, y) \in E_{a,b}(\mathbb{F}_q)$ with $x \neq 0$, represent $P = (x, \lambda)$, where $\lambda = x + \frac{y}{x}$:

$$(\lambda^2 + \lambda + a) \cdot x^2 = x^4 + b.$$

We must have efficient formulas for point **addition** and doubling.

λ-projective point: $P = (X, L, Z)$ corresponds to the λ-affine point $(X/Z, L/Z)$. The lambda-projective form of the Weierstrass equation is:

$$(L^2 + LZ + a \cdot Z^2) \cdot X^2 = X^4 + b \cdot Z^4.$$

Let $P = (X_P, L_P, Z_P)$ be a point in $E_{a,b}(\mathbb{F}_q)$. Then the formula for $2P = (X_{2P}, L_{2P}, Z_{2P})$ using the λ-projective representation is given by

$$\begin{aligned} T &= L_P^2 + (L_P \cdot Z_P) + a \cdot Z_P^2 \\ X_{2P} &= T^2 \\ Z_{2P} &= T \cdot Z_P^2 \\ L_{2P} &= (X_P \cdot Z_P)^2 + X_{2P} + T \cdot (L_P \cdot Z_P) + Z_{2P}. \end{aligned}$$

Four multiplications, one multiplication by the a-coefficient and four squarings.

Let $P = (X_P, L_P, Z_P)$ be a point in $E_{a,b}(\mathbb{F}_q)$. Then the formula for $2P = (X_{2P}, L_{2P}, Z_{2P})$ using the λ-projective representation is given by

$$\begin{aligned} T &= L_P^2 + (L_P \cdot Z_P) + a \cdot Z_P^2 \\ X_{2P} &= T^2 \\ Z_{2P} &= T \cdot Z_P^2 \\ L_{2P} &= (X_P \cdot Z_P)^2 + X_{2P} + T \cdot (L_P \cdot Z_P) + Z_{2P}. \end{aligned}$$

Four multiplications, one multiplication by the a-coefficient and four squarings.

If the multiplication by the b-coefficient is fast, there is an alternative formula.

$$L_{2P} = (L_P + X_P)^2 \cdot ((L_P + X_P)^2 + T + Z_P^2) + (a^2 + b) \cdot Z_P^4 + X_{2P} + (a + 1) \cdot Z_{2P}.$$

Three multiplications, one multiplication by the a-coefficient, one multiplication by the b-coefficient and four squarings.

Let $P = (X_P, L_P, Z_P)$ and $Q = (X_Q, L_Q, Z_Q)$ be points in $E_{a,b}(\mathbb{F}_q)$ with $P \neq \pm Q$. Then the addition $P + Q = (X_{P+Q}, L_{P+Q}, Z_{P+Q})$ can be computed by the formulas

$$\begin{aligned} A &= L_P \cdot Z_Q + L_Q \cdot Z_P \\ B &= (X_P \cdot Z_Q + X_Q \cdot Z_P)^2 \\ X_{P+Q} &= A \cdot (X_P \cdot Z_Q) \cdot (X_Q \cdot Z_P) \cdot A \\ L_{P+Q} &= (A \cdot (X_Q \cdot Z_P) + B)^2 + (A \cdot B \cdot Z_Q) \cdot (L_P + Z_P) \\ Z_{P+Q} &= (A \cdot B \cdot Z_Q) \cdot Z_P. \end{aligned}$$

Eleven multiplications and two squarings.

Let $P = (X_P, L_P, Z_P)$ and $Q = (X_Q, L_Q, Z_Q)$ be points in $E_{a,b}(\mathbb{F}_q)$ with $P \neq \pm Q$. Then the addition $P + Q = (X_{P+Q}, L_{P+Q}, Z_{P+Q})$ can be computed by the formulas

$$\begin{aligned} A &= L_P \cdot Z_Q + L_Q \cdot Z_P \\ B &= (X_P \cdot Z_Q + X_Q \cdot Z_P)^2 \\ X_{P+Q} &= A \cdot (X_P \cdot Z_Q) \cdot (X_Q \cdot Z_P) \cdot A \\ L_{P+Q} &= (A \cdot (X_Q \cdot Z_P) + B)^2 + (A \cdot B \cdot Z_Q) \cdot (L_P + Z_P) \\ Z_{P+Q} &= (A \cdot B \cdot Z_Q) \cdot Z_P. \end{aligned}$$

For $Z_Q = 1$ (mixed addition).

Let $P = (X_P, L_P, Z_P)$ and $Q = (X_Q, L_Q, Z_Q)$ be points in $E_{a,b}(\mathbb{F}_q)$ with $P \neq \pm Q$. Then the addition $P + Q = (X_{P+Q}, L_{P+Q}, Z_{P+Q})$ can be computed by the formulas

$$\begin{aligned} A &= L_P + L_Q \cdot Z_P \\ B &= (X_P + X_Q \cdot Z_P)^2 \\ X_{P+Q} &= A \cdot X_P \cdot (X_Q \cdot Z_P) \cdot A \\ L_{P+Q} &= (A \cdot (X_Q \cdot Z_P) + B)^2 + (A \cdot B) \cdot (L_P + Z_P) \\ Z_{P+Q} &= (A \cdot B) \cdot Z_P. \end{aligned}$$

Eight multiplications and two squarings.

Let $P = (x_P, \lambda_P)$ and $Q = (X_Q, L_Q, Z_Q)$ be points in the curve $E_{a,b}(\mathbb{F}_q)$. Then the operation $2Q + P = (X_{2Q+P}, L_{2Q+P}, Z_{2Q+P})$ can be computed as follows:

$$
\begin{aligned}
T &= L_Q^2 + L_Q \cdot Z_Q + a \cdot Z_Q^2 \\
A &= X_Q^2 \cdot Z_Q^2 + T \cdot (L_Q^2 + (a + 1 + \lambda_P) \cdot Z_Q^2) \\
B &= (x_P \cdot Z_Q^2 + T)^2 \\
X_{2Q+P} &= (x_P \cdot Z_Q^2) \cdot A^2 \\
Z_{2Q+P} &= (A \cdot B \cdot Z_Q^2) \\
L_{2Q+P} &= T \cdot (A + B)^2 + (\lambda_P + 1) \cdot Z_{2Q+P}.
\end{aligned}
$$

Ten multiplications, one multiplication by the a-constant and six squarings.

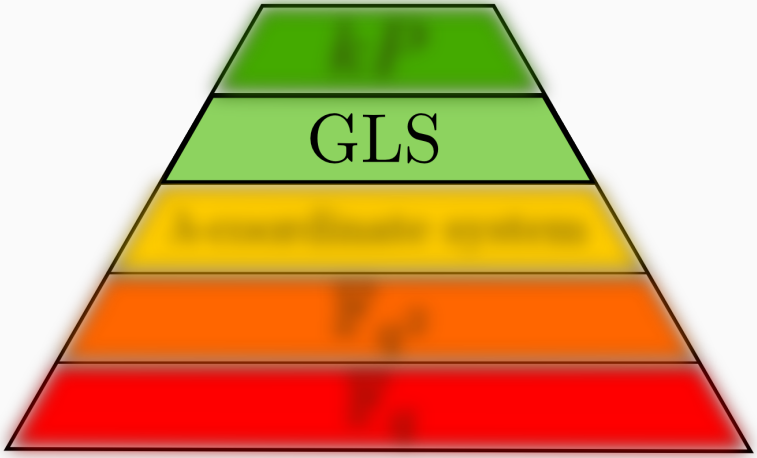
Two multiplications are saved against computing first a doubling followed by a point addition ($R = 2P$, $R = R + Q$).

Let $\tilde{m}, \tilde{s}, \tilde{m}_a$ be the cost of multiplication, squaring and multiplication by a-coefficient, respectively.

We choose $b = z^{27} + 1$ to be short.

	Coordinate systems		
	Lopez-Dahab	Lambda	
Full-addition	$13\tilde{m} + 4\tilde{s}$	$11\tilde{m} + 2\tilde{s}$	$-2\tilde{m} - 2\tilde{s}$
Mixed-addition	$8\tilde{m} + \tilde{m}_a + 5\tilde{s}$	$8\tilde{m} + 2\tilde{s}$	$-\tilde{m}_a - 3\tilde{s}$
Doubling	$3\tilde{m} + \tilde{m}_a + \tilde{m}_b + 5\tilde{s}$	$4\tilde{m} + \tilde{m}_a + 4\tilde{s}$ $3\tilde{m}+\tilde{m}_a+\tilde{m}_b+4\tilde{s}$	$+\tilde{m} - \tilde{m}_b - \tilde{s}$ $-\tilde{s}$
Doubling and addition	$11\tilde{m}+2\tilde{m}_a+\tilde{m}_b+10\tilde{s}$*	$10\tilde{m} + \tilde{m}_a + 6\tilde{s}$	$-\tilde{m} - \tilde{m}_a - \tilde{m}_b - 4\tilde{s}$

*When compared to LD-doubling + LD-mixed-addition.



GLS Binary Curves

The GLS (Galbraith, Lin and Scott) curves $E/\mathbb{F}_{q^2}$ is a large family of elliptic curves defined over $\mathbb{F}_{q^2}$ that admit **efficiently** computable endomorphisms [GLS11].

The GLS (Galbraith, Lin and Scott) curves $E/\mathbb{F}_{q^2}$ is a large family of elliptic curves defined over $\mathbb{F}_{q^2}$ that admit **efficiently** computable endomorphisms [GLS11].

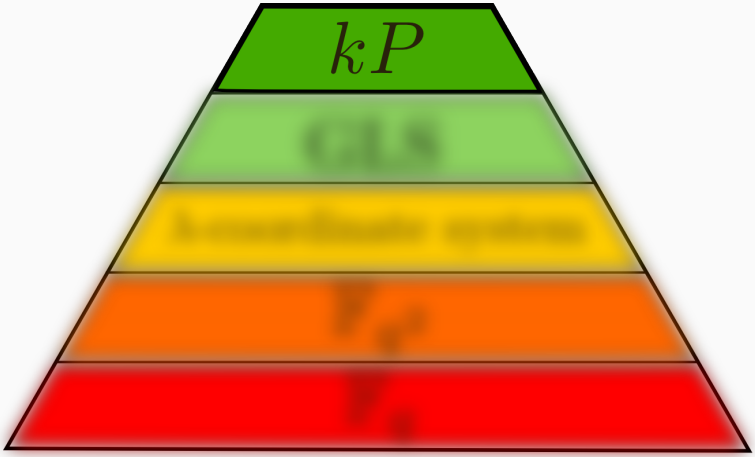
Hankerson et al. [HKM09] generalized the idea to binary curves.

The GLS curves have an **efficient endomorphism** $\psi : E \to E$ for $\lambda^2 = -1 \pmod{r}$:

$$[k]P = [k_0]P + [k_1]\psi(P)$$

$$k \equiv k_0 + k_1\lambda \pmod{r}$$

Advantage: We can convert k to (k_0, k_1) with each having half the length of k.



Scalar multiplication

Algorithm 4 Left-to-right Binary

Input: $P \in E$, $k = (k_{t-1}, \ldots, k_0)$
Output: $Q = [k]P$

1: $R \leftarrow \infty$
2: **for** $i \leftarrow t-1$ **downto** 0 **do**
3: $\quad R \leftarrow 2R$
4: $\quad$ **if** $k_i = 1$ **then**
5: $\quad\quad R \leftarrow R \oplus P$
6: $\quad$ **end if**
7: **end for**
8: **return** R

For security:

- Fix number of iterations
- Remove branch
- Point addition in constant time.
- Coordinate systems

Algorithm 5 Left-to-right Binary

Input: $P \in E$, $k = (k_{t-1}, \ldots, k_0)$

Output: $Q = [k]P$

1: $R \leftarrow \infty$
2: **for** $i \leftarrow t-1$ **downto** 0 **do**
3: $\quad R \leftarrow 2R$
4: $\quad R \leftarrow R \oplus P$ **if** $k_i = 1$
5: **end for**
6: **return** R

Ideas applied:

- Double-and-always-add
- Conditional copy (**compilers....**)
- Performance loss due to extra additions

Algorithm 6 Montgomery ladder [Mon87]

Input: $P \in E$, $k = (1, k_{t-2}, \ldots, k_1, k_0)$

Output: $Q = [k]P$

1: $R_0 \leftarrow P$, $R_1 \leftarrow [2]P$
2: **for** $i \leftarrow t-2$ **downto** 0 **do**
3: **if** $k_i \leftarrow 1$ **then**
4: $R_0 \leftarrow R_0 \oplus R_1$; $R_1 \leftarrow [2]R_1$
5: **else**
6: $R_1 \leftarrow R_0 \oplus R_1$; $R_0 \leftarrow [2]R_0$
7: **end if**
8: **end for**
9: **return** $Q = R_0$

For constant-time:

- Fixed number of iterations
- Accumulators R_i in the same order.
- Group law is implemented in constant time.

Algorithm 7 Left-to-right Montgomery ladder

Input: P, $k = (1, k_{t-2}, \ldots, k_1, k_0)$

Output: $Q = [k]P$

1: $k' \leftarrow$ Select $(k + r, k + 2r)$
2: $R_0 \leftarrow P$, $R_1 \leftarrow [2]P$
3: **for** $i \leftarrow \lg(r) - 1$ **downto** 0 **do**
4: Swap (R_0, R_1) if $k'_i = 0$
5: $R_0 \leftarrow R_0 \oplus R_1$; $R_1 \leftarrow [2]R_1$
6: Swap (R_0, R_1) if $k'_i = 0$
7: **end for**
8: **return** $Q = R_0$

For constant-time:

- Fixed iterations by **adding 1 or 2 multiples** of r.
- Replace branch with **conditional swap** (ideally implemented in ASM).
- **Careful** implementation of group law!

Important: **Extremely** suitable to Curve25519 in XZ coordinates for both performance/correctness.

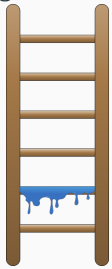
Algorithm 8 Left-to-right Montgomery ladder

Input: $P \in E$, $k = (1, k_{t-2}, \ldots, k_1, k_0)$

Output: $Q = [k]P$

1: $k' \leftarrow$ Select $(k + r, k + 2r)$
2: $R_0 \leftarrow P$, $R_1 \leftarrow [2]P$
3: **for** $i \leftarrow \lg(r) - 1$ **downto** 0 **do**
4: Swap (R_0, R_1) if $k'_i = 0$
5: $R_0 \leftarrow R_0 \oplus R_1$; $R_1 \leftarrow 2R_1$
6: Swap (R_0, R_1) if $k'_i = 0$
7: **end for**
8: **return** $Q = R_0$

LadderLeak: Breaking ECDSA With Less Than One Bit Of Nonce Leakage [ANT+20].



Problem: There are too many point additions!

Problem: There are too many point additions!

First, we decompose scalar in constant-time as $k = k_0 + k_1\lambda \pmod{r}$.

We can also convert integers to **non-zero** signed digits in the set $\{1, 3, \ldots, 2^{w-1})$ separated by $(w-1)$ spaces:

$$k_0 \to (d_\ell, 0, 0, d_{\ell-1}, 0, 0, d_{\ell-2}, 0, 0, \ldots, d_1, 0, 0, d_0)$$
$$k_1 \to (e_\ell, 0, 0, e_{\ell-1}, 0, 0, e_{\ell-2}, 0, 0, \ldots, d_1, 0, 0, e_0)$$

Advantage: Representation is **regular** and produces a **fixed-length** expansion [JT09].

Algorithm 9 Left-to-right w-ary

Input: $P, k_0 = \sum_{i=0}^{\ell} d_i \cdot 2^i$, $k_1 = \sum_{i=0}^{\ell} e_i \cdot 2^i$
Output: $Q = [k]P$
1: $R \leftarrow \infty$
2: $T[j] = [j]P$, odd $j \in [1, 2^{w-1})$
3: **for** $i \leftarrow \lceil \frac{\ell}{w-1} \rceil$ **downto** 0 **do**
4: $\quad R \leftarrow 2^{w-2}R$
5: $\quad Q \leftarrow \pm T[d_i], S\pm \leftarrow T[e_i]$
6: $\quad R \leftarrow 2R \oplus Q \oplus \psi(S)$
7: **end for**
8: **return** R

Ideas applied:

- Fixed length $\ell \approx \log(q)$
- Half of the point doublings
- **Linear pass** in T
- Fewer point additions $\lceil \frac{2}{w-1} \rceil$

Discovery: Computing $2R \oplus Q \oplus \psi(S)$ atomically is **faster** [OLAR14]!

The last step in the loop is computing $2Q \oplus \pm T[i] \oplus \psi(\pm T[j])$ for positive odd $i, j \in [1, 2^{w-1})$.

We can reduce the number of point additions in the loop by building instead a 2D table

$$T[i, j] = T[i] \oplus \psi(T[j]) = [i]P \oplus [j]\psi(P)$$

.

Important: This does not sound very smart...

The last step in the loop is computing $2Q \oplus \pm T[i] \oplus \psi(\pm T[j])$ for positive odd $i, j \in [1, 2^{w-1})$.

We can reduce the number of point additions in the loop by building instead a 2D table

$$T[i, j] = T[i] \oplus \psi(T[j]) = [i]P \oplus [j]\psi(P)$$

.

Important: This does not sound very smart... many more points to **compute/store**!

The last step in the loop is computing $2Q \oplus \pm T[i] \oplus \psi(\pm T[j])$ for positive odd $i, j \in [1, 2^{w-1})$.

We can reduce the number of point additions in the loop by building instead a 2D table

$$T[i, j] = T[i] \oplus \psi(T[j]) = [i]P \oplus [j]\psi(P)$$

.

Important: This does not sound very smart... many more points to **compute/store**!

We can reduce the table size by using ψ:

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

From positive i, j we can now compute all $\pm[i]P \oplus \pm[j]\psi(P)$ using ψ and **negation** maps.

More: We also found relatively straight-forward formulas for **faster** precomputation:

- Faster formulas for computing $3P$
- A faster formula for point **doubling-and-addition**
- A faster formula for computing $P + \psi(P)$
- Faster formulas for computing $P \pm Q$

Algorithm 10 Left-to-right w-ary 2D

Input: $P, k_0 = \sum_{i=0}^{\ell} d_i \cdot 2^i$, $k_1 = \sum_{i=0}^{\ell} e_i \cdot 2^i$
Output: $Q = [k]P$

1: $R \leftarrow \infty$
2: $T[i,j] = [i]P + [j]\psi(P)$, odd $i, j \in [1, 2^{w-1})$
3: **for** $i \leftarrow \lceil \frac{\ell}{w-1} \rceil$ **downto** 0 **do**
4: $\quad R \leftarrow 2^{w-2}R$
5: $\quad Q \leftarrow \pm T[d_i, e_i]$
6: $\quad R \leftarrow (2R \oplus Q) \vee (2R \oplus \psi(Q))$
7: **end for**
8: **return** R

Ideas applied:

- Heavier precomputation
- Conditional ψ to compress table
- Single **Linear pass** in T
- Fewer point additions $\lceil \frac{1}{w-1} \rceil$

Table 3: Constant-time variable base scalar multiplication benchmarks that are mostly performed on an Arm Quad-Core Cortex-A55 2.0 GHz. Memory is measured in terms of the number of elliptic curve points stored in the online precomputed table.

Implementation	Algorithm	Memory	Cycles
Lenngren [Len] (Cortex-A55)	Curve25519	0	157,182
Longa [Lon16] (Cortex-A55)	Four$\mathbb{Q}$	8	191,184
Longa [Lon16] (Cortex-A15)	Four$\mathbb{Q}$	8	132,000
This work (Cortex-A55)	GLS254 1D $w = 5$	8	92,460
	GLS254 2D $w = 3$	4	**86,525**
	GLS254 2D $w = 4$	16	91,682

Table 4: Constant-time variable base scalar multiplication benchmarks for 64-bit Intel Core i7 4770 Haswell at 3.4GHz, and Core i7 7700 Kaby Lake at 3.6GHz, both with TurboBoost disabled. Memory is measured in terms of the number of elliptic curve points stored in the precomputed table.

Implementation	Algorithm	Memory	Cycles
Longa et al. [CL15] (Haswell)	Four$\mathbb{Q}$	8	56,000
Longa et al. [CL15] (Kaby Lake)	Four$\mathbb{Q}$	8	47,052
Oliveira et al. [OLAR14] (Haswell)	GLS254 1D $w = 5$	8	48,301
Oliveira et al. [OLAR16] (Skylake)	GLS254 1D $w = 5$	8	38,044

Table 4: Constant-time variable base scalar multiplication benchmarks for 64-bit Intel Core i7 4770 Haswell at 3.4GHz, and Core i7 7700 Kaby Lake at 3.6GHz, both with TurboBoost disabled. Memory is measured in terms of the number of elliptic curve points stored in the precomputed table.

Implementation	Algorithm	Memory	Cycles
Longa et al. [CL15] (Haswell)	Four$\mathbb{Q}$	8	56,000
Longa et al. [CL15] (Kaby Lake)	Four$\mathbb{Q}$	8	47,052
Oliveira et al. [OLAR14] (Haswell)	GLS254 1D $w = 5$	8	48,301
Oliveira et al. [OLAR16] (Skylake)	GLS254 1D $w = 5$	8	38,044
This work (Haswell)	GLS254 1D $w = 5$	8	45,966
	GLS254 2D $w = 3$	4	45,253
	GLS254 2D $w = 4$	16	47,184

Table 4: Constant-time variable base scalar multiplication benchmarks for 64-bit Intel Core i7 4770 Haswell at 3.4GHz, and Core i7 7700 Kaby Lake at 3.6GHz, both with TurboBoost disabled. Memory is measured in terms of the number of elliptic curve points stored in the precomputed table.

Implementation	Algorithm	Memory	Cycles
Longa et al. [CL15] (Haswell)	Four$\mathbb{Q}$	8	56,000
Longa et al. [CL15] (Kaby Lake)	Four$\mathbb{Q}$	8	47,052
Oliveira et al. [OLAR14] (Haswell)	GLS254 1D $w = 5$	8	48,301
Oliveira et al. [OLAR16] (Skylake)	GLS254 1D $w = 5$	8	38,044
This work (Haswell)	GLS254 1D $w = 5$	8	45,966
	GLS254 2D $w = 3$	4	45,253
	GLS254 2D $w = 4$	16	47,184
This work (Kaby Lake)	GLS254 1D $w = 5$	8	36,480
	GLS254 2D $w = 3$	4	**35,739**
	GLS254 2D $w = 4$	16	38,076

Out admittedly not quite readable code is available at:

`https://github.com/dfaranha/gls254`

The ePrint by Aardal & Aranha will be out soon! It contains a proof that the Algorithm 9 and 10 do not incur exceptions (except last iteration).

There is still plenty to do:

- Adapt the 2D idea back to prime curves
- Implement in 32-bit ARMv7 with NEON instructions
- Accelerate implementation with AVX-512 instructions
- Benchmark in newer machines (**24,000** cycles in Ice Lake...)

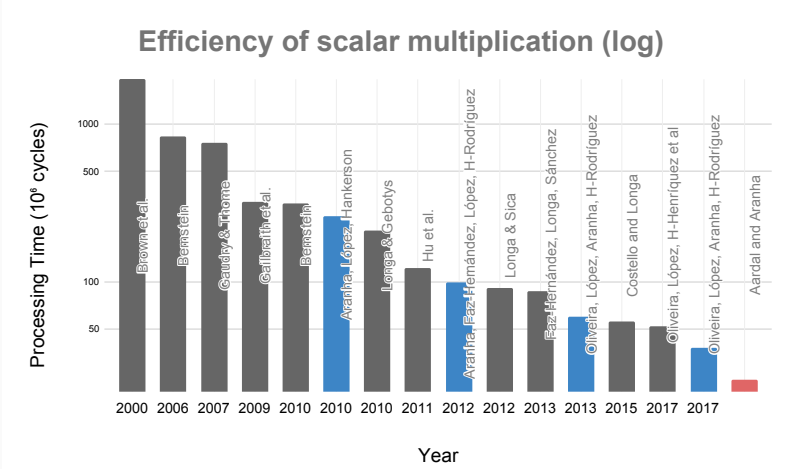


Figure 8: Results at 128-bit security on Intel CPUs from collaborations and current record.

References I

- Diego F. Aranha, Julio César López-Hernández, and Darrel Hankerson, *Efficient software implementation of binary field arithmetic using vector instruction sets*, LATINCRYPT, Lecture Notes in Computer Science, vol. 6212, Springer, 2010, pp. 144–161.
- Diego F. Aranha, Felipe Rodrigues Novaes, Akira Takahashi, Mehdi Tibouchi, and Yuval Yarom, *Ladderleak: Breaking ECDSA with less than one bit of nonce leakage*, CCS, ACM, 2020, pp. 225–242.
- Daniel J. Bernstein, *Curve25519: New diffie-hellman speed records*, Public Key Cryptography, Lecture Notes in Computer Science, vol. 3958, Springer, 2006, pp. 207–228.
- Daniel J. Bernstein and Bo-Yin Yang, *Fast constant-time gcd computation and modular inversion*, IACR Trans. Cryptogr. Hardw. Embed. Syst. **2019** (2019), no. 3, 340–398.

References II

- Craig Costello and Patrick Longa, *Four$\mathbb{Q}$: four-dimensional decompositions on a $\mathbb{Q}$-curve over the Mersenne prime*, IACR Cryptol. ePrint Arch. (2015), 565.
- Kenny Fong, Darrel Hankerson, Julio César López-Hernández, and Alfred Menezes, *Field inversion and point halving revisited*, IEEE Trans. Computers **53** (2004), no. 8, 1047–1059.
- Steven D. Galbraith, Xibin Lin, and Michael Scott, *Endomorphisms for faster elliptic curve cryptography on a large class of curves*, J. Cryptol. **24** (2011), no. 3, 446–469.
- Darrel Hankerson, Koray Karabina, and Alfred Menezes, *Analyzing the galbraith-lin-scott point multiplication method for elliptic curves over binary fields*, IEEE Trans. Computers **58** (2009), no. 10, 1411–1420.
- Toshiya Itoh and Shigeo Tsujii, *A fast algorithm for computing multiplicative inverses in gf (2m) using normal bases*, Information and computation **78** (1988), no. 3, 171–177.

- Marc Joye and Michael Tunstall, *Exponent recoding and regular exponentiation algorithms*, AFRICACRYPT, LNCS, vol. 5580, Springer, 2009, pp. 334–349.
- Julio César López-Hernández and Ricardo Dahab, *High-speed software multiplication in f_{2^m}*, INDOCRYPT, Lecture Notes in Computer Science, vol. 1977, Springer, 2000, pp. 203–212.
- Emil Lenngren, *AArch64 optimized implementation for X25519*, `https://github.com/Emill/X25519-AArch64`.
- Patrick Longa, *Four$\mathbb{Q}$neon: Faster elliptic curve scalar multiplications on ARM processors*, SAC, LNCS, vol. 10532, Springer, 2016, pp. 501–519.
- Peter L Montgomery, *Speeding the pollard and elliptic curve methods of factorization*, Mathematics of computation **48** (1987), no. 177, 243–264.

- Thomaz Oliveira, Julio César López-Hernández, Diego F. Aranha, and Francisco Rodríguez-Henríquez, *Two is the fastest prime: lambda coordinates for binary elliptic curves*, J. Cryptogr. Eng. **4** (2014), no. 1, 3–17.
- _____, *Improving the performance of the gls254. (2016)*, Presentation at CHES 2016 rump session (2016).
- Jerome A. Solinas, *Efficient arithmetic on koblitz curves*, Des. Codes Cryptogr. **19** (2000), no. 2/3, 195–249.
- Yuval Yarom and Katrina Falkner, *FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack*, USENIX Security Symposium, USENIX Association, 2014, pp. 719–732.

For some choice of a' it can be shown that $E' = E_{a',b}$ are isomorphic to E over $\mathbb{F}_{q^4}[s]/(s^2 + s = a + a')$ (**quadratic twist**) under a *twisting* isomorphism

$$\phi : (x, y) \mapsto (x, y + sx).$$

We will see later how to accelerate scalar multiplication using endomorphisms.

Let $\pi : E \to E$ be the Frobenius map, $(x, y) \mapsto (x^q, y^q)$. Then the endomorphism $\psi : E' \to E'$ is defined as $\psi = \phi\pi\phi^{-1}$ and represented by the formula

$$\psi : (x, y) \mapsto (x^q, y^q + x^q(s^q + s)).$$

Let $\pi : E \to E$ be the Frobenius map, $(x, y) \mapsto (x^q, y^q)$. Then the endomorphism $\psi : E' \to E'$ is defined as $\psi = \phi\pi\phi^{-1}$ and represented by the formula

$$\psi : (x, y) \mapsto (x^q, y^q + x^q(s^q + s)).$$

For our choice of elliptic curve E defined over the quadratic field $\mathbb{F}_{q^2} \cong \mathbb{F}_{2^{127}}[u]/(u^2 + u + 1)$ we have,

$$\psi(P) = \psi(x_0 + x_1 u, y_0 + y_1 u) \mapsto ((x_0 + x_1) + x_1 u, (y_0 + y_1 + 1) + (y_1 + 1)u)$$

Let $\pi : E \to E$ be the Frobenius map, $(x, y) \mapsto (x^q, y^q)$. Then the endomorphism $\psi : E' \to E'$ is defined as $\psi = \phi\pi\phi^{-1}$ and represented by the formula

$$\psi : (x, y) \mapsto (x^q, y^q + x^q(s^q + s)).$$

For our choice of elliptic curve E defined over the quadratic field $\mathbb{F}_{q^2} \cong \mathbb{F}_{2^{127}}[u]/(u^2 + u + 1)$ we have,

$$\psi(P) = \psi(x_0 + x_1 u, y_0 + y_1 u) \mapsto ((x_0 + x_1) + x_1 u, (y_0 + y_1 + 1) + (y_1 + 1)u)$$

Lambda Coordinates Aftermath

For points in λ-affine representation, the endomorphism is computed as $\psi(x_0 + x_1 u, \lambda_0 + \lambda_1 u) \mapsto ((x_0 + x_1) + x_1 u, (\lambda_0 + \lambda_1) + (\lambda_1 + 1)u)$.