Experimenting with Faster Elliptic Curves in Rust

Diego F. Aranha[1,2]

[1]Dept. of CS and DIGIT, Aarhus University, Denmark

[2]Concordium Blockchain Research Center (COBRA)

COBRA
CONCORDIUM BLOCKCHAIN
RESEARCH CENTER AARHUS

This talk

Improving implementations of elliptic curves in Rust

1. A quick background on **curves and pairings**
2. Faster algorithms for constant-time scalar multiplication
3. Some experimental results and comparison to C

Background and definitions

Background

Cryptography

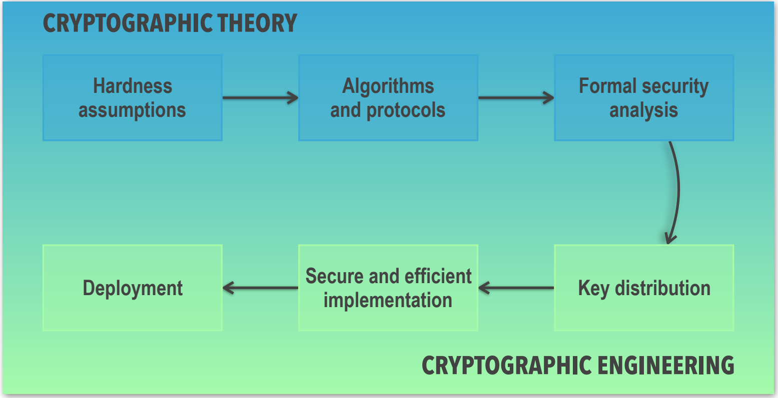
Practice and study of techniques for secure communication in the presence of **adversaries**.

Security goals: confidentiality, origin authentication, data integrity, non-repudiation.

Cryptography is everywhere. Modern banking, e-commerce, multimedia, voting and messaging systems **all** use cryptography.

Motivation

Figure 1: The lifetime of an application of cryptography:



Groups

Definition

A *group* $\mathbb{G}$ is a set equipped with a *binary operation* with the following properties: closure, identity, inverse, associativity.

When $\mathbb{G}$ is a finite group, we denote the number of elements the *order* of $\mathbb{G}$. A group is said to be *abelian* if it is **commutative**.

Examples: Rubik cube, $(\mathbb{Z}, +)$, $(\mathbb{R}^*, \times)$.

Fields

Definition

A *field* $\mathbb{F}$ is composed of elements and with two binary operations (addition and multiplication).

Both operations respect the usual properties, and they also distribute with each other.

Examples: $\mathbb{R}, \mathbb{C}$.

In cryptography we care about **prime fields** $\mathbb{F}_p$ formed by the integers $\{0, 1, \ldots, p-1\}$, with prime p and operations $\bmod p$.

Elliptic curves

Elliptic curves provide efficient public key cryptography:

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

	Security level		
Algorithm	80	128	256
Integer Factoring (RSA)	1024	3072	15360
Elliptic curves (ECC)	**160**	**256**	**512**

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

Elliptic curves

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

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

where $a, b \in \mathbb{F}_p$ and a **point at infinity** ∞.

Elliptic curves

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

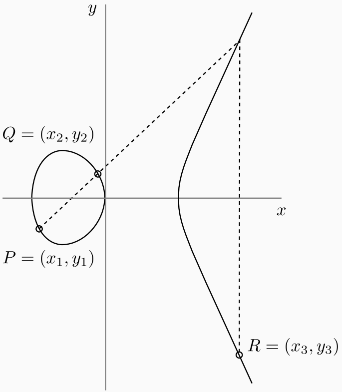
$$E : y^2 = x^3 + ax + b$$

where $a, b \in \mathbb{F}_p$ and a **point at infinity** ∞.

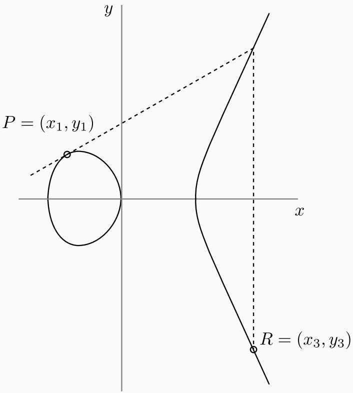
Group law: Points under the operation $\oplus$ (chord and tangent) forms an additive group of order q with ∞ as the identity.

Coordinate system: We represent a point in **affine coordinates** (x, y) using **projective** (X, Y, Z) such that $x = X/Z, y = Y/Z$.

Elliptic curves



(a) Point addition $R = P \oplus Q$



(b) Point doubling $R = [2]P$

Elliptic curves

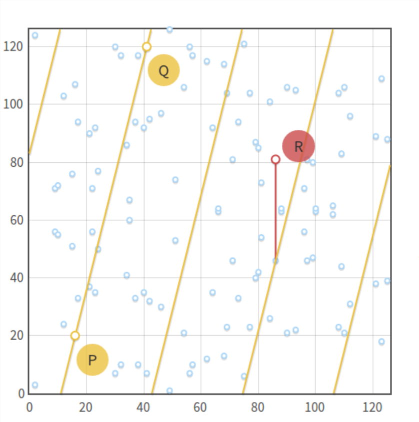


Figure 3: Point addition over the curve $y^2 = x^3 - x + 3$ over $GF(127)$, with $P = (16, 20)$ and $Q = (41, 120)$. Note how line $y = 4x + 83$ behaves $\bmod p$.

Elliptic curves

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

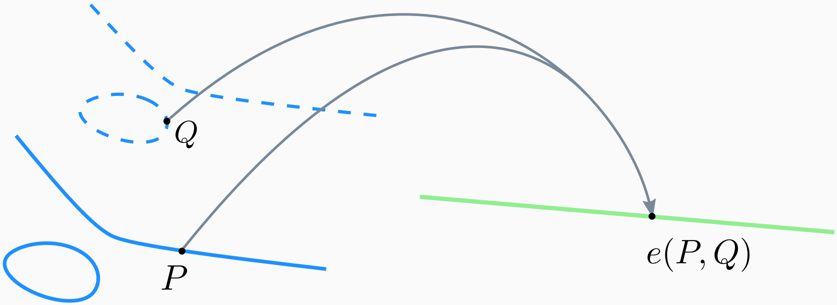
$$[k]P = \underbrace{P \oplus P \oplus \ldots \oplus P}_{k \text{ times}}.$$

Assumption: Recover k from $(P, [k]P)$ is **hard**!

Example: Public key in ECC is defined as $Q = [sk]G$ for fixed G.

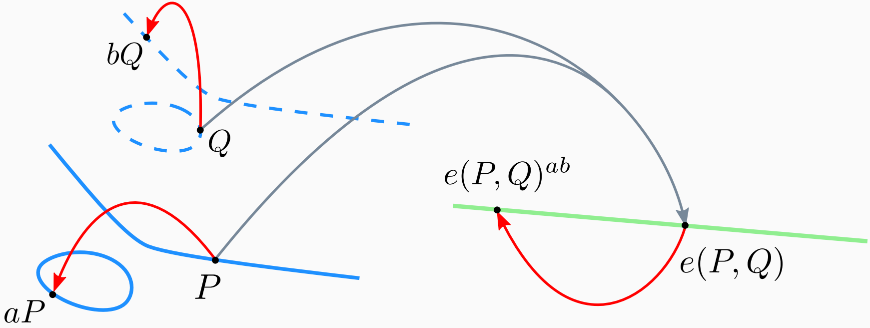
Bilinear pairings

$$e(P+R,Q) = e(P,Q)\cdot e(R,Q) \text{ and } e(P,Q+S) = e(P,Q)\cdot e(P,S)$$



Bilinear pairings

Inputs come from curves $\mathbb{G}_1$ and $\mathbb{G}_2$.



Applications: most zero-knowledge proofs needs pairings.

Pairing implementations

Protocols

$e(P, Q)$

$E(\mathbb{F}_q)$ $E(\mathbb{F}_{q^k})$

$\mathbb{F}_{q^k}$

$\mathbb{F}_q$

Low-level backend

Side-channel analysis

Side-channel analysis

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

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

Note: Increasing order of intrusiveness.

Side-channel attacks

Timing attacks

If the execution time varies with bits from the key, timing information can be used to recover parts of the key.

```
int pwdcmp(const void *str, const void *pwd, size_t size) {
    int v = 0;
    char *_a = (char *)str, *_b = (char *)pwd;

    while(size-- > 0 && v == 0)
        v = *(a++) - *(b++);
    return v;
}
```

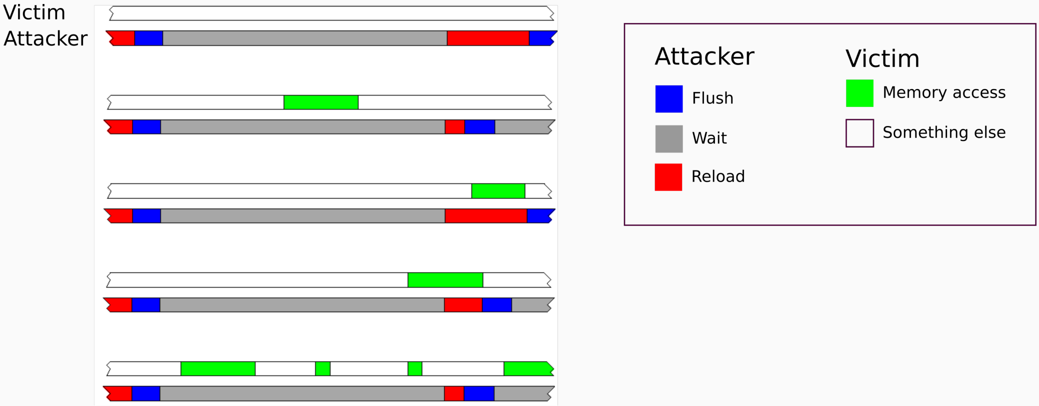
Secure implementation

```c
int util_cmp_const(const void *str, const void *pwd, size_t size) {
    char *_a = (const char *) str, *_b = (const char *) pwd;
    unsigned char result = 0;
    size_t i;

    for (i = 0; i < size; i++)
      result |= _a[i] ^ _b[i];

    return result; /* returns 0 if equal, nonzero otherwise. */
}
```

Important: Noise is not enough to prevent leakage!

Cache-timing attacks

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



Secure scalar multiplication

Timing attacks in scalar multiplication

Algorithm 1 Left-to-right Binary

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

Output: $Q = [k]P$

$R \leftarrow \infty$

for $i \leftarrow t - 1$ **downto** 0 **do**

  $R \leftarrow 2R$

  if $k_i = 1$ **then**

    $R \leftarrow R \oplus P$

return R

For security:

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

Constant-time scalar multiplication

Algorithm 2 Left-to-right Binary

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

Output: $Q = [k]P$

$R \leftarrow \infty$

for $i \leftarrow t-1$ **downto** 0 **do**

  $R \leftarrow 2R$

  $R \leftarrow R \oplus P$ **if** $k_i = 1$

return R

Ideas applied:

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

Constant-time scalar multiplication

We can **recode** integers in a different representation, where **non-zero odd** digits are separated by w spaces:

$$k = (d_\ell, 0, 0, d_{\ell-1}, 0, 0, d_{\ell-2}, 0, 0, \ldots, d_1, 0, 0, d_0)$$

Advantage: Representation is **regular**! The JT algorithm does this and produces a **fixed-length** expansion ($\ell = \lceil \frac{len(q)}{w-1} \rceil$).

Constant-time scalar multiplication

Algorithm 3 Left-to-right w-ary

Input: $P = (x, y)$, $k = \sum_{i=0}^{\ell} d_i \cdot 2^i$

Output: $Q = [k]P$

1: $R \leftarrow \infty$

2: $T_j = [j]P, j \in [1, 2^{w-1})$

3: **for** $i \leftarrow \ell$ **downto** 0 **do**

4: $\quad R \leftarrow 2^{w-1}R$

5: $\quad Q \leftarrow T_{d_i} \leftarrow \{T\}$

6: $\quad R \leftarrow R \oplus Q$

7: **return** R

Ideas applied:

- Fixed length ℓ
- **Linear pass** in T
- Fewer point additions ($\lceil \frac{1}{w-1} \rceil$)

The role of endomorphisms

Many curves in cryptography have an extra **efficient map** $\psi(P) = [\lambda]P$ for $\lambda \approx \sqrt{q}$.

$$[k]P = [k_1]P + [k_2]\psi(P)$$

Advantage: We can convert k in (k_1, k_2) with each having half the length of k.

Constant-time scalar multiplication

Algorithm 4 LtR w-ary with ψ

Input: $P = (x, y)$, $k = \sum_{i=0}^{\ell} d_i \cdot 2^i$

Output: $Q = [k]P$

$R \leftarrow \infty$, $k = k_1 + \lambda \cdot k_2 \bmod r$

$T_j = [j]P, j \in \{1, 2^{w-1})$

for $i \leftarrow \lceil \ell/2 \rceil$ **downto** 0 **do**

  $R \leftarrow 2^{w-1}R$

  $Q_1, Q_2 \leftarrow T_{d_{1,i}}, T_{d_{2,i}} \leftarrow \{T\}$

  $R \leftarrow R \oplus Q_1 \oplus \psi(Q_2)$

return R

Ideas applied:

- **Endomorphism**
- Fewer point doublings
- For $w = 5$, we now have $\approx 1/2$ point doublings and $\approx 1/4$ point additions

Experimental setup

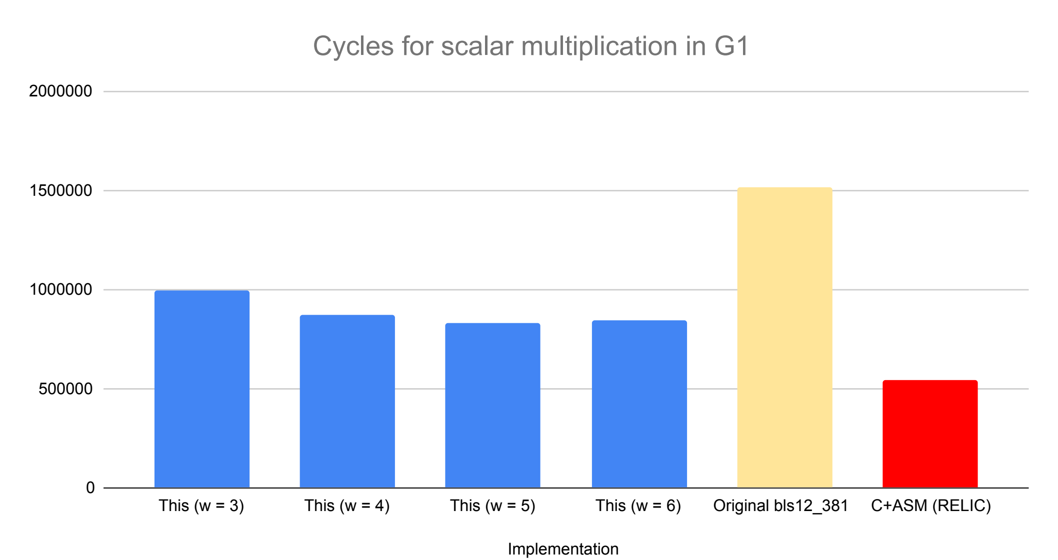
Target platforms:

- Intel Haswell Core i7-4770K (at 3.4GHz)
- TurboBoost **disabled** for reducing noise

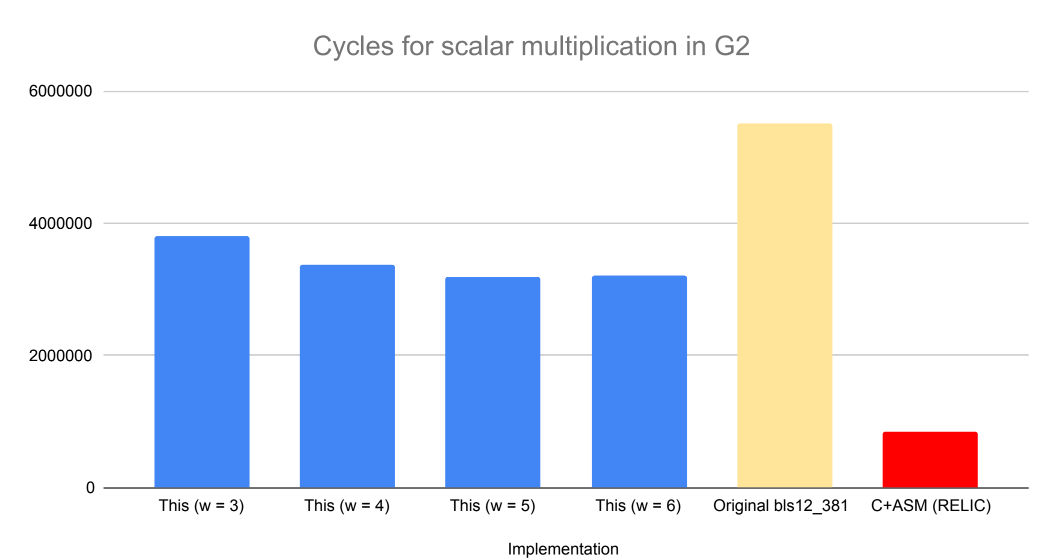
Tooling:

- `bls12_381` for curve arithmetic
- `ff` for finite field arithmetic
- `criterion` for benchmarking
- `subtle` for sensitive code
- Rust 1.60 nightly version

Experimental Results I – Curve $\mathbb{G}_1$



Experimental Results II – Curve $\mathbb{G}_2$



Main takeaways

- (Probably not **very** idiomatic) Code available at `https://github.com/dfaranha/bls12_381`
- Finally getting competitive with hand-optimized C for public-key crypto.
- Good idea of how efficient algorithm looks, now **formalize** it.
- Join the **Rust Cryptography Interest Group** to contribute!

Questions?

D. F. Aranha

`dfaranha@cs.au.dk`

`@dfaranha`