Fast, Safe, Pure-Rust Elliptic Curve Cryptography

Isis Lovecruft / Henry de Valence

RustConf 2017

Introductions

Introductions

- This talk is for those with light familiarity with Rust.

Introductions

- This talk is for those with light familiarity with Rust.
- It *isn't* aimed at cryptographers.

Introductions

- This talk is for those with light familiarity with Rust.
- It *isn't* aimed at cryptographers.
- Don't worry! We'll cover the basic terminology, and — with a tad of high-school level algebra — you should be able to follow along just fine.

Introductions

- This talk is for those with light familiarity with Rust.
- It *isn't* aimed at cryptographers.
- Don't worry! We'll cover the basic terminology, and — with a tad of high-school level algebra — you should be able to follow along just fine.
- If not, still don't worry! All questions are welcome, and if you're shy please feel free to talk to us privately afterwards, either in person or online.

Overview

What is `curve25519-dalek`?

Implementing low-level arithmetic in Rust

Rust features we love, and features we want to improve

Implementing crypto with `-dalek`

What is `curve25519-dalek`?

Anatomy of an elliptic curve cryptography implementation

<table>
<tr><td colspan="2">Applications</td></tr>
<tr><td>Protocol</td><td>Protocol-specific library</td></tr>
<tr><td>Group</td><td rowspan="3"><code>curve25519-dalek</code></td></tr>
<tr><td>Elliptic Curve</td></tr>
<tr><td>Finite Field</td></tr>
<tr><td colspan="2">CPU</td></tr>
</table>

Anatomy of an elliptic curve cryptography implementation

Applications	
Protocol	Protocol-specific library
Group	`curve25519-dalek`
Elliptic Curve	
Finite Field	
CPU	

Protocol: a specific cryptographic operation, such as a signature, a zero-knowledge proof, etc.

Anatomy of an elliptic curve cryptography implementation

Applications	
Protocol	Protocol-specific library
Group	`curve25519-dalek`
Elliptic Curve	
Finite Field	
CPU	

Protocol: a specific cryptographic operation, such as a signature, a zero-knowledge proof, etc.

Group: an abstract mathematical structure (like a trait) implemented concretely by an...

Anatomy of an elliptic curve cryptography implementation

<table>
<tr><td colspan="2">Applications</td></tr>
<tr><td>Protocol</td><td>Protocol-specific library</td></tr>
<tr><td>Group</td><td rowspan="3"><code>curve25519-dalek</code></td></tr>
<tr><td>Elliptic Curve</td></tr>
<tr><td>Finite Field</td></tr>
<tr><td colspan="2">CPU</td></tr>
</table>

Protocol: a specific cryptographic operation, such as a signature, a zero-knowledge proof, etc.

Group: an abstract mathematical structure (like a trait) implemented concretely by an...

Elliptic Curve: a set of points satisfying certain equations defined over a...

Anatomy of an elliptic curve cryptography implementation

<table>
<tr><td colspan="2">Applications</td></tr>
<tr><td>Protocol</td><td>Protocol-specific library</td></tr>
<tr><td>Group</td><td rowspan="3"><code>curve25519-dalek</code></td></tr>
<tr><td>Elliptic Curve</td></tr>
<tr><td>Finite Field</td></tr>
<tr><td colspan="2">CPU</td></tr>
</table>

Protocol: a specific cryptographic operation, such as a signature, a zero-knowledge proof, etc.

Group: an abstract mathematical structure (like a trait) implemented concretely by an...

Elliptic Curve: a set of points satisfying certain equations defined over a...

Finite Field: usually, integers modulo a prime p.

Anatomy of an elliptic curve cryptography implementation

Applications	
Protocol	Protocol-specific library
Group	`curve25519-dalek`
Elliptic Curve	
Finite Field	
CPU	

Protocol: a specific cryptographic operation, such as a signature, a zero-knowledge proof, etc.

Group: an abstract mathematical structure (like a trait) implemented concretely by an...

Elliptic Curve: a set of points satisfying certain equations defined over a...

Finite Field: usually, integers modulo a prime p.

Our implementation was originally based on Adam Langley's `ed25519` Go code, which was in turn based on the reference `ref10` implementation.

Historical Implementations

In order to talk about what `curve25519-dalek` is, and why we made it, it's important to revisit other elliptic curve libraries, their designs, and common problems.

Historical Implementations: Part I

Other elliptic curve libraries tend to have no separation between implementations of the field, curve, and group, and the protocols sitting on top of them.

Historical Implementations: Part I

Other elliptic curve libraries tend to have no separation between implementations of the field, curve, and group, and the protocols sitting on top of them.

This causes several immediate issues:

Historical Implementations: Part I

Other elliptic curve libraries tend to have no separation between implementations of the field, curve, and group, and the protocols sitting on top of them.

This causes several immediate issues:

- Idiosyncracies in the lower-level pieces of the implementation carry over into idiosyncracies in the protocol.

Historical Implementations: Part I

Other elliptic curve libraries tend to have no separation between implementations of the field, curve, and group, and the protocols sitting on top of them.

This causes several immediate issues:

- Idiosyncracies in the lower-level pieces of the implementation carry over into idiosyncracies in the protocol.
- Assumptions about how these lower-level pieces will be used aren't necessarily correct if someone wanted to reuse the code to implement a different protocol.

Historical Implementations: Part I

Other elliptic curve libraries tend to have no separation between implementations of the field, curve, and group, and the protocols sitting on top of them.

This causes several immediate issues:

- Idiosyncracies in the lower-level pieces of the implementation carry over into idiosyncracies in the protocol.
- Assumptions about how these lower-level pieces will be used aren't necessarily correct if someone wanted to reuse the code to implement a different protocol.
- Excessive copy-pasta with minor tweaks by other cryptographers (worsened by the fact that some cryptographers think that releasing unsigned tarballs of their implementations *inside* another tarball of a benchmarking suite is somehow an appropriate software distribution mechanism).

Historical Implementations: Part I (cont.)

This leads to large, monolithic codebases which are idiosyncratic, incompatible with one another, and highly specialised to perform only the single protocol they implement (usually, a signature scheme or Diffie-Hellman key exchange).

Historical Implementations: Part II

And there's worse.

Historical Implementations: Part II

And there's worse.

In major, widely-used, cryptographic libraries:

- Using C pointer arithmetic *to index an array*. In C, array indexing works both ways, e.g. `a[5] == 5[a]`. In this case they were doing `a[p+5]` (`== a+p[5] == 5[a+p]`).

Historical Implementations: Part II

And there's worse.

In major, widely-used, cryptographic libraries:

- Using C pointer arithmetic *to index an array*. In C, array indexing works both ways, e.g. `a[5] == 5[a]`. In this case they were doing `a[p+5]` (`== a+p[5] == 5[a+p]`).
- Overflowing signed integers in C and expecting the behaviour to be sane/similar across platforms and varying compilers.

Historical Implementations: Part II

And there's worse.

In major, widely-used, cryptographic libraries:

- Using C pointer arithmetic *to index an array*. In C, array indexing works both ways, e.g. `a[5] == 5[a]`. In this case they were doing `a[p+5]` (`== a+p[5] == 5[a+p]`).
- Overflowing signed integers in C and expecting the behaviour to be sane/similar across platforms and varying compilers.
- Using untyped integer arrays (e.g. `[u8; 32]`) as canonical, external representation for mathematically fundamentally incompatible types (e.g. points and numbers)

Historical Implementations: Part II

And there's worse.

In major, widely-used, cryptographic libraries:

- Using C pointer arithmetic *to index an array*. In C, array indexing works both ways, e.g. `a[5] == 5[a]`. In this case they were doing `a[p+5]` (`== a+p[5] == 5[a+p]`).
- Overflowing signed integers in C and expecting the behaviour to be sane/similar across platforms and varying compilers.
- Using untyped integer arrays (e.g. `[u8; 32]`) as canonical, external representation for mathematically fundamentally incompatible types (e.g. points and numbers)
- Using pointer arithmetic to determine both the size and location of a write buffer.

Historical Implementations: Part II

And there's worse.

In major, widely-used, cryptographic libraries:

- Using C pointer arithmetic *to index an array*. In C, array indexing works both ways, e.g. `a[5] == 5[a]`. In this case they were doing `a[p+5]` (`== a+p[5] == 5[a+p]`).
- Overflowing signed integers in C and expecting the behaviour to be sane/similar across platforms and varying compilers.
- Using untyped integer arrays (e.g. `[u8; 32]`) as canonical, external representation for mathematically fundamentally incompatible types (e.g. points and numbers)
- Using pointer arithmetic to determine both the size and location of a write buffer.
- *I can keep going.*

Design Goals of `curve25519-dalek`

- Usability

Design Goals of `curve25519-dalek`

- Usability
- Versatility

Design Goals of `curve25519-dalek`

- Usability
- Versatility
- Safety

Design Goals of `curve25519-dalek`

- Usability
- Versatility
- Safety
    - Memory Safety

Design Goals of `curve25519-dalek`

- Usability
- Versatility
- Safety
    - Memory Safety
    - Type Safety

Design Goals of `curve25519-dalek`

- Usability
- Versatility
- Safety
  - Memory Safety
  - Type Safety
  - Overflow/Underflow Detection

Design Goals of `curve25519-dalek`

- Usability
- Versatility
- Safety
  - Memory Safety
  - Type Safety
  - Overflow/Underflow Detection
- Readability

Design Goals of `curve25519-dalek`

- Usability
- Versatility
- Safety
    - Memory Safety
    - Type Safety
    - Overflow/Underflow Detection
- Readability
    - Explicitness

Design Goals of `curve25519-dalek`

- Usability
- Versatility
- Safety
  - Memory Safety
  - Type Safety
  - Overflow/Underflow Detection
- Readability …which implies
  - Explicitness

Design Goals of `curve25519-dalek`

- Usability
- Versatility
- Safety
    - Memory Safety
    - Type Safety
    - Overflow/Underflow Detection
- Readability …which implies
    - Explicitness
- Auditability

Design Goals of `curve25519-dalek`

- Usability
- Versatility
- Safety
  - Memory Safety
  - Type Safety
  - Overflow/Underflow Detection
- Readability …which implies
  - Explicitness
- Auditability

These are all things we would get from a higher-level, memory-safe, strongly-typed, polymorphic programming language,

Design Goals of `curve25519-dalek`

- Usability
- Versatility
- Safety
    - Memory Safety
    - Type Safety
    - Overflow/Underflow Detection
- Readability ...which implies
    - Explicitness
- Auditability

These are all things we would get from a higher-level, memory-safe, strongly-typed, polymorphic programming language,

a.k.a

Design Goals of `curve25519-dalek`

- Usability
- Versatility
- Safety
    - Memory Safety
    - Type Safety
    - Overflow/Underflow Detection
- Readability ...which implies
    - Explicitness
- Auditability

These are all things we would get from a higher-level, memory-safe, strongly-typed, polymorphic programming language,

a.k.a Rust.

Implementing low-level arithmetic in Rust

Example: implementing multiplication in $\mathbb{F}_p$, $p = 2^{255} - 19$

Let's jump down to the lowest abstraction layer: using primitive types to implement field arithmetic.

Example: implementing multiplication in $\mathbb{F}_p$, $p = 2^{255} - 19$

Let's jump down to the lowest abstraction layer: using primitive types to implement field arithmetic.

Specifically: how can we implement multiplication of two integers modulo $p = 2^{255} - 19$, using only the primitive operations provided by the CPU?

Example: implementing multiplication in $\mathbb{F}_p$, $p = 2^{255} - 19$

Let's jump down to the lowest abstraction layer: using primitive types to implement field arithmetic.

Specifically: how can we implement multiplication of two integers modulo $p = 2^{255} - 19$, using only the primitive operations provided by the CPU?

Two questions:

Example: implementing multiplication in $\mathbb{F}_p$, $p = 2^{255} - 19$

Let's jump down to the lowest abstraction layer: using primitive types to implement field arithmetic.

Specifically: how can we implement multiplication of two integers modulo $p = 2^{255} - 19$, using only the primitive operations provided by the CPU?

Two questions:

- What are the primitive operations?

Example: implementing multiplication in $\mathbb{F}_p$, $p = 2^{255} - 19$

Let's jump down to the lowest abstraction layer: using primitive types to implement field arithmetic.

Specifically: how can we implement multiplication of two integers modulo $p = 2^{255} - 19$, using only the primitive operations provided by the CPU?

Two questions:

- What are the primitive operations?
- What does multiplication in $\mathbb{F}_p$ look like?

Multiplication modes

Primitive types have a fixed size: `u8`, `i8`, …, `u64`, `i64`, etc., but numbers get bigger when you multiply them. What happens?

Multiplication modes

Primitive types have a fixed size: `u8`, `i8`, …, `u64`, `i64`, etc., but numbers get bigger when you multiply them. What happens?

Multiplication modes

Primitive types have a fixed size: `u8`, `i8`, ..., `u64`, `i64`, etc., but numbers get bigger when you multiply them. What happens?

1. Error on overflow (debug): `8u8 * 40u8 == panic!()`

Multiplication modes

Primitive types have a fixed size: `u8`, `i8`, ..., `u64`, `i64`, etc., but numbers get bigger when you multiply them. What happens?

1. Error on overflow (debug): `8u8 * 40u8 == panic!()`
2. Wrapping arithmetic (release): `8u8 * 40u8 == 64u8`

Multiplication modes

Primitive types have a fixed size: `u8`, `i8`, ..., `u64`, `i64`, etc., but numbers get bigger when you multiply them. What happens?

1. Error on overflow (debug): `8u8 * 40u8 == panic!()`
2. Wrapping arithmetic (release): `8u8 * 40u8 == 64u8`
3. Saturating arithmetic: `8u8 * 40u8 == 255u8`

Multiplication modes

Primitive types have a fixed size: `u8`, `i8`, ..., `u64`, `i64`, etc., but numbers get bigger when you multiply them. What happens?

1. Error on overflow (debug): `8u8 * 40u8 == panic!()`
2. Wrapping arithmetic (release): `8u8 * 40u8 == 64u8`
3. Saturating arithmetic: `8u8 * 40u8 == 255u8`
4. Widening arithmetic: `8u8 * 40u8 == 320u16`

Multiplication modes

Primitive types have a fixed size: `u8`, `i8`, ..., `u64`, `i64`, etc., but numbers get bigger when you multiply them. What happens?

1. Error on overflow (debug): `8u8 * 40u8 == panic!()`
2. Wrapping arithmetic (release): `8u8 * 40u8 == 64u8`
3. Saturating arithmetic: `8u8 * 40u8 == 255u8`
4. Widening arithmetic: `8u8 * 40u8 == 320u16`

Rust has intrinsics for 1, 2, and 3, and we can get 4 by writing

`(x as T) * (y as T)`,

where `T` is the next-wider type.

Lowering widening multiplication to assembly on x86-64

```rust
#![feature(i128_type)]
// Test to see how rustc / LLVM lowers a widening mul on x64
pub fn widening_mul(x: u64, y: u64) -> u128 {
  (x as u128) * (y as u128)
}  // made with https://rust.godbolt.org
```

rustc nightly (Editor #1, Compiler #1)

rustc nightly | -O

.LX0: | .text | // | \s+ | Intel

```asm
example::widening_mul:
  push rbp
  mov rbp, rsp
  mov rax, rsi
  mul rdi
  pop rbp
  ret
```

rustc 1.21.0-nightly (a7e0d3a81 2017-08-11)- *3600ms*

rustc nightly (Editor #1, Compiler #2)

rustc nightly | C target_cpu=haswell

.LX0: | .text | // | \s+ | Intel

```asm
example::widening_mul:
  push rbp
  mov rbp, rsp
  mov rdx, rsi
  mulx rdx, rax, rdi
  pop rbp
  ret
```

rustc 1.21.0-nightly (a7e0d3a81 2017-08-11)- *900ms*

Radix-2^{51} representation

The Ed25519 paper suggests using a "radix-2^{51}" representation.

Radix-2^{51} representation

The Ed25519 paper suggests using a "radix-2^{51}" representation.

What does this mean? It means we write numbers x, y as

$$x = x_0 + x_1 2^{51} + x_2 2^{102} + x_3 2^{153} + x_4 2^{204} \qquad 0 \le x_i \le 2^{51}$$
$$y = y_0 + y_1 2^{51} + y_2 2^{102} + y_3 2^{153} + y_4 2^{204} \qquad 0 \le y_i \le 2^{51}$$

Radix-2^{51} representation

The Ed25519 paper suggests using a "radix-2^{51}" representation.

What does this mean? It means we write numbers x, y as

$$x = x_0 + x_1 2^{51} + x_2 2^{102} + x_3 2^{153} + x_4 2^{204} \qquad 0 \leq x_i \leq 2^{51}$$
$$y = y_0 + y_1 2^{51} + y_2 2^{102} + y_3 2^{153} + y_4 2^{204} \qquad 0 \leq y_i \leq 2^{51}$$

Since $2^{51} < 2^{64}$, we can write this as

```
struct FieldElement64([u64;5])
```

and use the widening multiplication

```
(x[i] as u128) * (y[j] as u128)
```

Multiplication, part I

How do we multiply? Set $z = xy$. Then we can write down the coefficients of $z = z_0 + z_1 2^{51} + z_2 2^{102} + \dots$

Multiplication, part I

How do we multiply? Set $z = xy$. Then we can write down the coefficients of $z = z_0 + z_1 2^{51} + z_2 2^{102} + \dots$

$$z_0 = x_0 y_0 \qquad 1$$

Multiplication, part I

How do we multiply? Set $z = xy$. Then we can write down the coefficients of $z = z_0 + z_1 2^{51} + z_2 2^{102} + \dots$

$$z_0 = x_0 y_0 \qquad 1$$

$$z_1 = x_0 y_1 + x_1 y_0 \qquad 2^{51}$$

Multiplication, part I

How do we multiply? Set $z = xy$. Then we can write down the coefficients of $z = z_0 + z_1 2^{51} + z_2 2^{102} + \ldots$

$$
\begin{aligned}
z_0 &= x_0 y_0 && 1 \\
z_1 &= x_0 y_1 + x_1 y_0 && 2^{51} \\
z_2 &= x_0 y_2 + x_1 y_1 + x_2 y_0 && 2^{102}
\end{aligned}
$$

Multiplication, part I

How do we multiply? Set $z = xy$. Then we can write down the coefficients of $z = z_0 + z_1 2^{51} + z_2 2^{102} + \dots$

$$
\begin{aligned}
z_0 &= x_0 y_0 && 1 \\
z_1 &= x_0 y_1 + x_1 y_0 && 2^{51} \\
z_2 &= x_0 y_2 + x_1 y_1 + x_2 y_0 && 2^{102} \\
z_3 &= x_0 y_3 + x_1 y_2 + x_2 y_1 + x_3 y_0 && 2^{153}
\end{aligned}
$$

Multiplication, part I

How do we multiply? Set $z = xy$. Then we can write down the coefficients of $z = z_0 + z_1 2^{51} + z_2 2^{102} + \ldots$

$$
\begin{aligned}
z_0 &= x_0 y_0 && 1 \\
z_1 &= x_0 y_1 + x_1 y_0 && 2^{51} \\
z_2 &= x_0 y_2 + x_1 y_1 + x_2 y_0 && 2^{102} \\
z_3 &= x_0 y_3 + x_1 y_2 + x_2 y_1 + x_3 y_0 && 2^{153} \\
z_4 &= x_0 y_4 + x_1 y_3 + x_2 y_2 + x_3 y_1 + x_4 y_0 && 2^{204}
\end{aligned}
$$

Multiplication, part I

How do we multiply? Set $z = xy$. Then we can write down the coefficients of $z = z_0 + z_1 2^{51} + z_2 2^{102} + \dots$

$$
\begin{aligned}
z_0 &= x_0 y_0 && 1 \\
z_1 &= x_0 y_1 + x_1 y_0 && 2^{51} \\
z_2 &= x_0 y_2 + x_1 y_1 + x_2 y_0 && 2^{102} \\
z_3 &= x_0 y_3 + x_1 y_2 + x_2 y_1 + x_3 y_0 && 2^{153} \\
z_4 &= x_0 y_4 + x_1 y_3 + x_2 y_2 + x_3 y_1 + x_4 y_0 && 2^{204} \\
z_5 &= x_1 y_4 + x_2 y_3 + x_3 y_2 + x_4 y_1 && 2^{255}
\end{aligned}
$$

Multiplication, part I

How do we multiply? Set $z = xy$. Then we can write down the coefficients of $z = z_0 + z_1 2^{51} + z_2 2^{102} + \dots$

$$
\begin{aligned}
z_0 &= x_0 y_0 && 1 \\
z_1 &= x_0 y_1 + x_1 y_0 && 2^{51} \\
z_2 &= x_0 y_2 + x_1 y_1 + x_2 y_0 && 2^{102} \\
z_3 &= x_0 y_3 + x_1 y_2 + x_2 y_1 + x_3 y_0 && 2^{153} \\
z_4 &= x_0 y_4 + x_1 y_3 + x_2 y_2 + x_3 y_1 + x_4 y_0 && 2^{204} \\
z_5 &= x_1 y_4 + x_2 y_3 + x_3 y_2 + x_4 y_1 && 2^{255} \\
z_6 &= x_2 y_4 + x_3 y_3 + x_4 y_2 && 2^{306}
\end{aligned}
$$

Multiplication, part I

How do we multiply? Set $z = xy$. Then we can write down the coefficients of $z = z_0 + z_1 2^{51} + z_2 2^{102} + \dots$

$$\begin{aligned}
z_0 &= x_0y_0 && 1\\
z_1 &= x_0y_1 + x_1y_0 && 2^{51}\\
z_2 &= x_0y_2 + x_1y_1 + x_2y_0 && 2^{102}\\
z_3 &= x_0y_3 + x_1y_2 + x_2y_1 + x_3y_0 && 2^{153}\\
z_4 &= x_0y_4 + x_1y_3 + x_2y_2 + x_3y_1 + x_4y_0 && 2^{204}\\
z_5 &= x_1y_4 + x_2y_3 + x_3y_2 + x_4y_1 && 2^{255}\\
z_6 &= x_2y_4 + x_3y_3 + x_4y_2 && 2^{306}\\
z_7 &= x_3y_4 + x_4y_3 && 2^{357}
\end{aligned}$$

Multiplication, part I

How do we multiply? Set $z = xy$. Then we can write down the coefficients of $z = z_0 + z_1 2^{51} + z_2 2^{102} + \dots$

$$
\begin{aligned}
z_0 &= x_0y_0 && 1\\
z_1 &= x_0y_1 + x_1y_0 && 2^{51}\\
z_2 &= x_0y_2 + x_1y_1 + x_2y_0 && 2^{102}\\
z_3 &= x_0y_3 + x_1y_2 + x_2y_1 + x_3y_0 && 2^{153}\\
z_4 &= x_0y_4 + x_1y_3 + x_2y_2 + x_3y_1 + x_4y_0 && 2^{204}\\
z_5 &= x_1y_4 + x_2y_3 + x_3y_2 + x_4y_1 && 2^{255}\\
z_6 &= x_2y_4 + x_3y_3 + x_4y_2 && 2^{306}\\
z_7 &= x_3y_4 + x_4y_3 && 2^{357}\\
z_8 &= x_4y_4 && 2^{408}
\end{aligned}
$$

Multiplication, part II

Since $p = 2^{255} - 19$, we have $2^{255} \equiv 19 \pmod{p}$.

Multiplication, part II

Since $p = 2^{255} - 19$, we have $2^{255} \equiv 19 \pmod{p}$.

This means that we can do inline reduction:

$$z_0 + z_1 2^{51} + z_2 2^{102} + z_3 2^{153} + z_4 2^{204} + z_5 2^{255} + z_6 2^{306} + z_7 2^{357} + z_8 2^{408}$$

Multiplication, part II

Since $p = 2^{255} - 19$, we have $2^{255} \equiv 19 \pmod{p}$.

This means that we can do inline reduction:

$$
\begin{aligned}
&z_0 + z_1 2^{51} + z_2 2^{102} + z_3 2^{153} + z_4 2^{204} + z_5 2^{255} + z_6 2^{306} + z_7 2^{357} + z_8 2^{408} \\
&\equiv (z_0 + 19z_5) + (z_1 + 19z_6)2^{51} + (z_2 + 19z_7)2^{102} + (z_3 + 19z_8)2^{153} + z_4 2^{204} \pmod{p}
\end{aligned}
$$

Multiplication, part II

Since $p = 2^{255} - 19$, we have $2^{255} \equiv 19 \pmod{p}$.

This means that we can do inline reduction:

$$
\begin{aligned}
&z_0 + z_1 2^{51} + z_2 2^{102} + z_3 2^{153} + z_4 2^{204} + z_5 2^{255} + z_6 2^{306} + z_7 2^{357} + z_8 2^{408} \\
&\equiv (z_0 + 19z_5) + (z_1 + 19z_6)2^{51} + (z_2 + 19z_7)2^{102} + (z_3 + 19z_8)2^{153} + z_4 2^{204} \pmod{p}
\end{aligned}
$$

We can combine this with the formulas on the previous slide:

Multiplication, part II

Since $p = 2^{255} - 19$, we have $2^{255} \equiv 19 \pmod{p}$.

This means that we can do inline reduction:

$$\begin{aligned} &z_0 + z_1 2^{51} + z_2 2^{102} + z_3 2^{153} + z_4 2^{204} + z_5 2^{255} + z_6 2^{306} + z_7 2^{357} + z_8 2^{408} \\ &\equiv (z_0 + 19 z_5) + (z_1 + 19 z_6) 2^{51} + (z_2 + 19 z_7) 2^{102} + (z_3 + 19 z_8) 2^{153} + z_4 2^{204} \pmod{p} \end{aligned}$$

We can combine this with the formulas on the previous slide:

$$z_0 = x_0 y_0 + 19(x_1 y_4 + x_2 y_3 + x_3 y_2 + x_4 y_1) \qquad 1$$

Multiplication, part II

Since $p = 2^{255} - 19$, we have $2^{255} \equiv 19 \pmod{p}$.

This means that we can do inline reduction:

$$z_0 + z_1 2^{51} + z_2 2^{102} + z_3 2^{153} + z_4 2^{204} + z_5 2^{255} + z_6 2^{306} + z_7 2^{357} + z_8 2^{408}$$
$$\equiv (z_0 + 19z_5) + (z_1 + 19z_6)2^{51} + (z_2 + 19z_7)2^{102} + (z_3 + 19z_8)2^{153} + z_4 2^{204} \pmod{p}$$

We can combine this with the formulas on the previous slide:

$$z_0 = x_0y_0 + 19(x_1y_4 + x_2y_3 + x_3y_2 + x_4y_1) \qquad 1$$
$$z_1 = x_0y_1 + x_1y_0 + 19(x_2y_4 + x_3y_3 + x_4y_2) \qquad 2^{51}$$

Multiplication, part II

Since $p = 2^{255} - 19$, we have $2^{255} \equiv 19 \pmod{p}$.

This means that we can do inline reduction:

$$\begin{aligned} & z_0 + z_1 2^{51} + z_2 2^{102} + z_3 2^{153} + z_4 2^{204} + z_5 2^{255} + z_6 2^{306} + z_7 2^{357} + z_8 2^{408} \\ & \equiv (z_0 + 19 z_5) + (z_1 + 19 z_6) 2^{51} + (z_2 + 19 z_7) 2^{102} + (z_3 + 19 z_8) 2^{153} + z_4 2^{204} \pmod{p} \end{aligned}$$

We can combine this with the formulas on the previous slide:

$$\begin{aligned} z_0 &= x_0 y_0 + 19(x_1 y_4 + x_2 y_3 + x_3 y_2 + x_4 y_1) && 1 \\ z_1 &= x_0 y_1 + x_1 y_0 + 19(x_2 y_4 + x_3 y_3 + x_4 y_2) && 2^{51} \\ z_2 &= x_0 y_2 + x_1 y_1 + x_2 y_0 + 19(x_3 y_4 + x_4 y_3) && 2^{102} \end{aligned}$$

Multiplication, part II

Since $p = 2^{255} - 19$, we have $2^{255} \equiv 19 \pmod{p}$.

This means that we can do inline reduction:

$$\begin{aligned} & z_0 + z_1 2^{51} + z_2 2^{102} + z_3 2^{153} + z_4 2^{204} + z_5 2^{255} + z_6 2^{306} + z_7 2^{357} + z_8 2^{408} \\ & \equiv (z_0 + 19 z_5) + (z_1 + 19 z_6) 2^{51} + (z_2 + 19 z_7) 2^{102} + (z_3 + 19 z_8) 2^{153} + z_4 2^{204} \pmod{p} \end{aligned}$$

We can combine this with the formulas on the previous slide:

$$\begin{aligned} z_0 &= x_0 y_0 + 19(x_1 y_4 + x_2 y_3 + x_3 y_2 + x_4 y_1) && 1 \\ z_1 &= x_0 y_1 + x_1 y_0 + 19(x_2 y_4 + x_3 y_3 + x_4 y_2) && 2^{51} \\ z_2 &= x_0 y_2 + x_1 y_1 + x_2 y_0 + 19(x_3 y_4 + x_4 y_3) && 2^{102} \\ z_3 &= x_0 y_3 + x_1 y_2 + x_2 y_1 + x_3 y_0 + 19(x_4 y_4) && 2^{153} \end{aligned}$$

Multiplication, part II

Since $p = 2^{255} - 19$, we have $2^{255} \equiv 19 \pmod{p}$.

This means that we can do inline reduction:

$$\begin{aligned} & z_0 + z_1 2^{51} + z_2 2^{102} + z_3 2^{153} + z_4 2^{204} + z_5 2^{255} + z_6 2^{306} + z_7 2^{357} + z_8 2^{408} \\ & \equiv (z_0 + 19z_5) + (z_1 + 19z_6)2^{51} + (z_2 + 19z_7)2^{102} + (z_3 + 19z_8)2^{153} + z_4 2^{204} \pmod{p} \end{aligned}$$

We can combine this with the formulas on the previous slide:

$$\begin{aligned} z_0 &= x_0y_0 + 19(x_1y_4 + x_2y_3 + x_3y_2 + x_4y_1) && 1 \\ z_1 &= x_0y_1 + x_1y_0 + 19(x_2y_4 + x_3y_3 + x_4y_2) && 2^{51} \\ z_2 &= x_0y_2 + x_1y_1 + x_2y_0 + 19(x_3y_4 + x_4y_3) && 2^{102} \\ z_3 &= x_0y_3 + x_1y_2 + x_2y_1 + x_3y_0 + 19(x_4y_4) && 2^{153} \\ z_4 &= x_0y_4 + x_1y_3 + x_2y_2 + x_3y_1 + x_4y_0 && 2^{204} \end{aligned}$$

Rust implementation, part I

Let's write this in Rust:

```
impl<'a, 'b> Mul<&'b FieldElement64> for &'a FieldElement64 {
    type Output = FieldElement64;
    fn mul(self, _rhs: &'b FieldElement64) -> FieldElement64 {
        #[inline(always)]
        fn m(x: u64, y: u64) -> u128 { (x as u128) * (y as u128) }

        // Alias self, _rhs for more readable formulas
        let a: &[u64; 5] = &self.0; let b: &[u64; 5] = &_rhs.0;
        // 64-bit precomputations to avoid 128-bit multiplications
        let b1_19 = b[1]*19; let b2_19 = b[2]*19; let b3_19 = b[3]*19; let b4_19 = b[4]*19;

        // Multiply to get 128-bit coefficients of output
        let c0 = m(a[0],b[0]) + m(a[4],b1_19) + m(a[3],b2_19) + m(a[2],b3_19) + m(a[1],b4_19);
        let c1 = m(a[1],b[0]) + m(a[0],b[1])  + m(a[4],b2_19) + m(a[3],b3_19) + m(a[2],b4_19);
        let c2 = m(a[2],b[0]) + m(a[1],b[1])  + m(a[0],b[2])  + m(a[4],b3_19) + m(a[3],b4_19);
        let c3 = m(a[3],b[0]) + m(a[2],b[1])  + m(a[1],b[2])  + m(a[0],b[3])  + m(a[4],b4_19);
        let c4 = m(a[4],b[0]) + m(a[3],b[1])  + m(a[2],b[2])  + m(a[1],b[3])  + m(a[0],b[4]);
```

Rust implementation, part I

Let's write this in Rust:

```
impl<'a, 'b> Mul<&'b FieldElement64> for &'a FieldElement64 {
    type Output = FieldElement64;
    fn mul(self, _rhs: &'b FieldElement64) -> FieldElement64 {
        #[inline(always)]
        fn m(x: u64, y: u64) -> u128 { (x as u128) * (y as u128) }

        // Alias self, _rhs for more readable formulas
        let a: &[u64; 5] = &self.0; let b: &[u64; 5] = &_rhs.0;
        // 64-bit precomputations to avoid 128-bit multiplications
        let b1_19 = b[1]*19; let b2_19 = b[2]*19; let b3_19 = b[3]*19; let b4_19 = b[4]*19;

        // Multiply to get 128-bit coefficients of output
        let c0 = m(a[0],b[0]) + m(a[4],b1_19) + m(a[3],b2_19) + m(a[2],b3_19) + m(a[1],b4_19);
        let c1 = m(a[1],b[0]) + m(a[0],b[1])  + m(a[4],b2_19) + m(a[3],b3_19) + m(a[2],b4_19);
        let c2 = m(a[2],b[0]) + m(a[1],b[1])  + m(a[0],b[2])  + m(a[4],b3_19) + m(a[3],b4_19);
        let c3 = m(a[3],b[0]) + m(a[2],b[1])  + m(a[1],b[2])  + m(a[0],b[3])  + m(a[4],b4_19);
        let c4 = m(a[4],b[0]) + m(a[3],b[1])  + m(a[2],b[2])  + m(a[1],b[3])  + m(a[0],b[4]);
```

However, the c_i are too big: we want `u64`s, not `u128`s.

Rust implementation, part II

To finish, we reduce the size of the coefficients by carrying their values upwards into higher coefficients: $(c_{i+1}, c_i) \leftarrow (c_{i+1} + \lfloor c_i/2^{51} \rfloor, c_i \mod 2^{51})$

Rust implementation, part II

To finish, we reduce the size of the coefficients by carrying their values upwards into higher coefficients: $(c_{i+1}, c_i) \leftarrow (c_{i+1} + \lfloor c_i/2^{51} \rfloor, c_i \mod 2^{51})$

```
        let low_51_bit_mask = (1u64 << 51) - 1;
        c1 +=  c0 >> 51;
        let mut c0: u64 = (c0 as u64) & low_51_bit_mask;
        c2 +=  c1 >> 51;
        let c1: u64 = (c1 as u64) & low_51_bit_mask;
        c3 +=  c2 >> 51;
        let c2: u64 = (c2 as u64) & low_51_bit_mask;
        c4 +=  c3 >> 51;
        let c3: u64 = (c3 as u64) & low_51_bit_mask;
        c0 += ((c4 >> 51) as u64) * 19;
        let c4: u64 = (c4 as u64) & low_51_bit_mask;

        // Now all c_i fit in u64; reduce again to enforce c_i < 2^51
        FieldElement64::reduce([c0,c1,c2,c3,c4])
    }
}
```

Rust implementation, part II

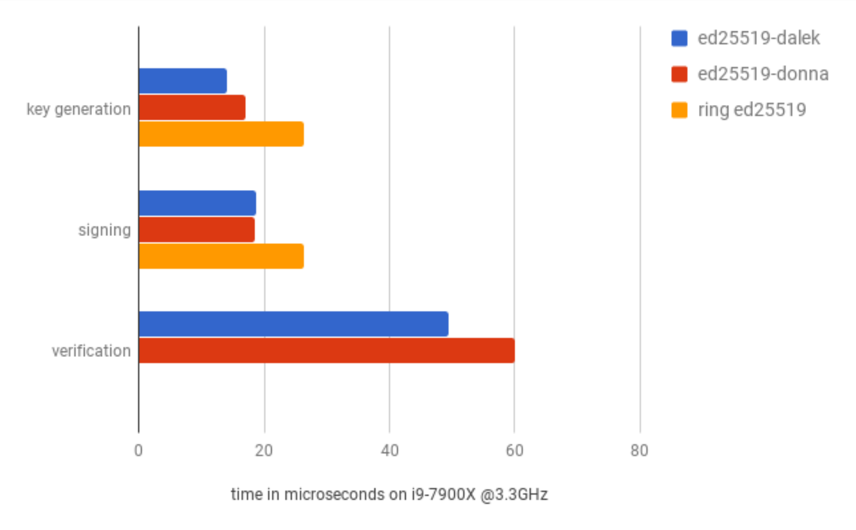
To finish, we reduce the size of the coefficients by carrying their values upwards into higher coefficients: $(c_{i+1}, c_i) \leftarrow (c_{i+1} + \lfloor c_i/2^{51} \rfloor, c_i \mod 2^{51})$

```
        let low_51_bit_mask = (1u64 << 51) - 1;
        c1 +=  c0 >> 51;
        let mut c0: u64 = (c0 as u64) & low_51_bit_mask;
        c2 +=  c1 >> 51;
        let c1: u64 = (c1 as u64) & low_51_bit_mask;
        c3 +=  c2 >> 51;
        let c2: u64 = (c2 as u64) & low_51_bit_mask;
        c4 +=  c3 >> 51;
        let c3: u64 = (c3 as u64) & low_51_bit_mask;
        c0 += ((c4 >> 51) as u64) * 19;
        let c4: u64 = (c4 as u64) & low_51_bit_mask;

        // Now all c_i fit in u64; reduce again to enforce c_i < 2^51
        FieldElement64::reduce([c0,c1,c2,c3,c4])
    }
}
```

And... except for some comments and debug assertions, that's essentially the implementation we use!

How fast is it?



Rust features we love, and features we want to improve

Constant-time code and LLVM

Rust's code generation is done by LLVM.

Constant-time code and LLVM

Rust's code generation is done by LLVM. It's really good at optimizing and generating code!

Constant-time code and LLVM

Rust’s code generation is done by LLVM. It’s really good at optimizing and generating code!

One worry is that the optimizer could, in theory, break constant-time properties of the implementation. What does this mean?

Constant-time code and LLVM

Rust's code generation is done by LLVM. It's really good at optimizing and generating code!

One worry is that the optimizer could, in theory, break constant-time properties of the implementation. What does this mean?

A side channel is a way for an adversary to determine internal program state by watching it execute.

Constant-time code and LLVM

Rust's code generation is done by LLVM. It's really good at optimizing and generating code!

One worry is that the optimizer could, in theory, break constant-time properties of the implementation. What does this mean?

A side channel is a way for an adversary to determine internal program state by watching it execute. For instance, if the program branches on secret data, an observer could learn which branch was taken (and hence information about the secrets).

Constant-time code and LLVM

Rust's code generation is done by LLVM. It's really good at optimizing and generating code!

One worry is that the optimizer could, in theory, break constant-time properties of the implementation. What does this mean?

A side channel is a way for an adversary to determine internal program state by watching it execute. For instance, if the program branches on secret data, an observer could learn which branch was taken (and hence information about the secrets).

To prevent this, the implementation's behaviour should be *uniform with respect to secret data*.

Constant-time code and LLVM

Rust's code generation is done by LLVM. It's really good at optimizing and generating code!

One worry is that the optimizer could, in theory, break constant-time properties of the implementation. What does this mean?

A side channel is a way for an adversary to determine internal program state by watching it execute. For instance, if the program branches on secret data, an observer could learn which branch was taken (and hence information about the secrets).

To prevent this, the implementation's behaviour should be *uniform with respect to secret data*. LLVM's optimizer, on `x86_64`, doesn't currently break our code.

Constant-time code and LLVM

Rust's code generation is done by LLVM. It's really good at optimizing and generating code!

One worry is that the optimizer could, in theory, break constant-time properties of the implementation. What does this mean?

A side channel is a way for an adversary to determine internal program state by watching it execute. For instance, if the program branches on secret data, an observer could learn which branch was taken (and hence information about the secrets).

To prevent this, the implementation's behaviour should be *uniform with respect to secret data*. LLVM's optimizer, on `x86_64`, doesn't currently break our code.

In the future, we'd like to do CI testing of the generated binaries: Rust, but verify.

Rust everywhere with `no_std` and FFI

Rust is capable of targeting many platforms, and targeting extremely constrained environments using `no_std`.

Rust everywhere with `no_std` and FFI

Rust is capable of targeting many platforms, and targeting extremely constrained environments using `no_std`.

`-dalek` works with `no_std`, so Rust code using `-dalek` can provide FFI and be embedded in weird places:

Rust everywhere with `no_std` and FFI

Rust is capable of targeting many platforms, and targeting extremely constrained environments using `no_std`.

`-dalek` works with `no_std`, so Rust code using `-dalek` can provide FFI and be embedded in weird places:

Tony Arcieri (@bascule) got `ed25519-dalek` running on an embedded PowerPC CPU inside of a hardware security module, and is working on running it under SGX;

Rust everywhere with `no_std` and FFI

Rust is capable of targeting many platforms, and targeting extremely constrained environments using `no_std`.

`-dalek` works with `no_std`, so Rust code using `-dalek` can provide FFI and be embedded in weird places:

Tony Arcieri (@bascule) got `ed25519-dalek` running on an embedded PowerPC CPU inside of a hardware security module, and is working on running it under SGX;

Filippo Valsorda (@FiloSottile)'s `rustgo` allows coordinating Rust function calls with the Go runtime with minimal overhead, and used calling `curve25519-dalek` as an example.

Rust everywhere with `no_std` and FFI

Rust is capable of targeting many platforms, and targeting extremely constrained environments using `no_std`.

`-dalek` works with `no_std`, so Rust code using `-dalek` can provide FFI and be embedded in weird places:

Tony Arcieri (@bascule) got `ed25519-dalek` running on an embedded PowerPC CPU inside of a hardware security module, and is working on running it under SGX;

Filippo Valsorda (@FiloSottile)'s `rustgo` allows coordinating Rust function calls with the Go runtime with minimal overhead, and used calling `curve25519-dalek` as an example. (It's $3\times$ faster than the implementation in the Go standard library).

Rust features which could be better

- The Eye of Sauron

Rust features which could be better

- The Eye of Sauron `&(&(&(&()))))`

Rust features which could be better

- The Eye of Sauron `&(&(&(&()))))`
  Rust's operator traits take arguments of type `T`, not of type `&T`.

Rust features which could be better

- The Eye of Sauron `&(&(&(&()))))`
  Rust's operator traits take arguments of type `T`, not of type `&T`.
  To avoid a copy/move on every operation, you need to implement `Mul` for `&T` instead of `T`:

Rust features which could be better

- The Eye of Sauron `&(&(&(&())))`

  Rust's operator traits take arguments of type `T`, not of type `&T`.

  To avoid a copy/move on every operation, you need to implement `Mul` for `&T` instead of `T`:

  ```
  let u = &Z.square() - &(&constants::d4 * &ss);
  ```

Rust features which could be better

- The Eye of Sauron `&(&(&(&())))`
  Rust's operator traits take arguments of type `T`, not of type `&T`.
  To avoid a copy/move on every operation, you need to implement `Mul` for `&T` instead of `T`:
  `let u = &Z.square() - &(&constants::d4 * &ss);`
  This gets messy quickly. Possible solution: auto-borrow `Copy` types?

Rust features which could be better

- The Eye of Sauron `&(&(&(&())))`
  Rust's operator traits take arguments of type `T`, not of type `&T`.
  To avoid a copy/move on every operation, you need to implement `Mul` for `&T` instead of `T`:
  ```
  let u = &Z.square() - &(&constants::d4 * &ss);
  ```
  This gets messy quickly. Possible solution: auto-borrow `Copy` types?
- const generics!

Rust features which could be better

- The Eye of Sauron `&(&(&(&())))`
  Rust's operator traits take arguments of type `T`, not of type `&T`.
  To avoid a copy/move on every operation, you need to implement `Mul` for `&T` instead of `T`:
  ```
  let u = &Z.square() - &(&constants::d4 * &ss);
  ```
  This gets messy quickly. Possible solution: auto-borrow `Copy` types?
- const generics!
  We've already thought of cool ways to abuse const generics to optimize field arithmetic.

Rust features which could be better

- The Eye of Sauron `&(&(&(&()))))`

  Rust's operator traits take arguments of type `T`, not of type `&T`.

  To avoid a copy/move on every operation, you need to implement `Mul` for `&T` instead of `T`:

  ```
  let u = &Z.square() - &(&constants::d4 * &ss);
  ```

  This gets messy quickly. Possible solution: auto-borrow `Copy` types?

- const generics!

  We've already thought of cool ways to abuse const generics to optimize field arithmetic.

  Basic idea: statically track the sizes of intermediate values, and use specialization to insert reductions only when necessary.

Implementing crypto with `-dalek`

`macros_rule!` and zero-knowledge proofs

Zero-knowledge proofs allow users to prove statements about secret values without revealing any extra information.

`macros_rule!` and zero-knowledge proofs

Zero-knowledge proofs allow users to prove statements about secret values without revealing any extra information.

Example: given points A, B, G, H, and a secret value x, I want to prove that $A = Gx$ and $B = Hx$ without revealing anything about my secret x value.

`macros_rule!` and zero-knowledge proofs

Zero-knowledge proofs allow users to prove statements about secret values without revealing any extra information.

Example: given points A, B, G, H, and a secret value x, I want to prove that $A = Gx$ and $B = Hx$ without revealing anything about my secret x value.

Implementing these proofs involves a lot of boilerplate, especially for proving more complicated expressions in zero knowledge.

`macros_rule!` and zero-knowledge proofs

Zero-knowledge proofs allow users to prove statements about secret values without revealing any extra information.

Example: given points A, B, G, H, and a secret value x, I want to prove that $A = Gx$ and $B = Hx$ without revealing anything about my secret x value.

Implementing these proofs involves a lot of boilerplate, especially for proving more complicated expressions in zero knowledge.

Solution: our `zkp` crate has an experimental zero-knowledge proof compiler in Rust macros.

`macros_rule!` and zero-knowledge proofs

Zero-knowledge proofs allow users to prove statements about secret values without revealing any extra information.

Example: given points A, B, G, H, and a secret value x, I want to prove that $A = Gx$ and $B = Hx$ without revealing anything about my secret x value.

Implementing these proofs involves a lot of boilerplate, especially for proving more complicated expressions in zero knowledge.

Solution: our `zkp` crate has an experimental zero-knowledge proof compiler in Rust macros.

```
create_nipk!{dleq, (x), (A, B, G, H) : A = (G * x), B = (H * x) }
```

`macros_rule!` and zero-knowledge proofs

Zero-knowledge proofs allow users to prove statements about secret values without revealing any extra information.

Example: given points A, B, G, H, and a secret value x, I want to prove that $A = Gx$ and $B = Hx$ without revealing anything about my secret x value.

Implementing these proofs involves a lot of boilerplate, especially for proving more complicated expressions in zero knowledge.

Solution: our `zkp` crate has an experimental zero-knowledge proof compiler in Rust macros.

```
create_nipk!{dleq, (x), (A, B, G, H) : A = (G * x), B = (H * x) }
```

This creates a `dleq` module with all the code for creating and verifying these proof statements, using Serde to convert to/from wire format.

Implementing rangeproofs with `-dalek`

Another type of zero-knowledge proof is a rangeproof: proving that a secret number lies in a particular range, without revealing any other information.

Implementing rangeproofs with `-dalek`

Another type of zero-knowledge proof is a rangeproof: proving that a secret number lies in a particular range, without revealing any other information.

These are used in confidential transaction systems, and in a future anti-censorship system we designed for Tor.

Implementing rangeproofs with `-dalek`

Another type of zero-knowledge proof is a rangeproof: proving that a secret number lies in a particular range, without revealing any other information.

These are used in confidential transaction systems, and in a future anti-censorship system we designed for Tor.

Basic idea: to prove $x \in [0, b^n]$, write x in base b as $x = \sum_{i=0}^{n-1} x_i b^i$, and prove that each digit is in range: $x_i \in [0, b]$.

Implementing rangeproofs with `-dalek`

Another type of zero-knowledge proof is a rangeproof: proving that a secret number lies in a particular range, without revealing any other information.

These are used in confidential transaction systems, and in a future anti-censorship system we designed for Tor.

Basic idea: to prove $x \in [0, b^n]$, write x in base b as $x = \sum_{i=0}^{n-1} x_i b^i$, and prove that each digit is in range: $x_i \in [0, b]$.

Verification essentially amounts to checking each digit's proof: if each digit is in range, the whole number is in range.

Implementing rangeproofs with `-dalek`

Another type of zero-knowledge proof is a rangeproof: proving that a secret number lies in a particular range, without revealing any other information.

These are used in confidential transaction systems, and in a future anti-censorship system we designed for Tor.

Basic idea: to prove $x \in [0, b^n]$, write x in base b as $x = \sum_{i=0}^{n-1} x_i b^i$, and prove that each digit is in range: $x_i \in [0, b]$.

Verification essentially amounts to checking each digit's proof: if each digit is in range, the whole number is in range.

We implemented the Back-Maxwell rangeproof, which uses $b = 3$ and shares data between digits to save space.

Implementing rangeproofs with -dalek: (partial) code

```
// mi_H[i] = m^i * H = 3^i * H in the loop below, construct these serially here:
let mut mi_H = vec![*H; n];
let mut mi2_H = vec![*H; n];
for i in 1..n {
    mi2_H[i-1] = &mi_H[i-1] + &mi_H[i-1];
    mi_H[i] = &mi_H[i-1] + &mi2_H[i-1];
}
mi2_H[n-1] = &mi_H[n-1] + &mi_H[n-1];

// Need to collect into a Vec to get par_iter()
let indices: Vec<_> = (0..n).collect();
let compressed_Ris: Vec<_> = indices.par_iter().map(|j| {
    let i = *j;

    let Ci_minus_miH = &self.C[i] - &mi_H[i];
    let P = vartime::multiscalar_mult(&[self.s_1[i], -&self.e_0], &[G, Ci_minus_miH]);
    let ei_1 = Scalar::hash_from_bytes::<Sha512>(P.compress().as_bytes());

    let Ci_minus_2miH = &self.C[i] - &mi2_H[i];
    let P = vartime::multiscalar_mult(&[self.s_2[i], -&ei_1], &[G, Ci_minus_2miH]);
    let ei_2 = Scalar::hash_from_bytes::<Sha512>(P.compress().as_bytes());

    let Ri = &self.C[i] * &ei_2;

    Ri.compress()
}).collect();
```

Thank you!

Isis Agora Lovecruft	Henry de Valence
@isislovecruft	@hdevalence
isis@patternsinthevoid.net	hdevalence@hdevalence.ca
https://patternsinthevoid.net	https://hdevalence.ca