Fast, Safe, Pure-Rust Elliptic Curve Cryptography

Isis Lovecruft

Noisebridge 10th Anniversary
September 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`?

Rust is excellent

Implementing cryptography with `-dalek`

What is `curve25519-dalek`?

Anatomy of an elliptic curve cryptography implementation

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

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

<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...

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

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

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).

The modus operandi for implementing cryptographic software is writing microarchitecture-optimised, *artisinal assembly*, with the purported goals of lower clock cycles and constantimedness. (However the neither goal is guaranteed, as we'll see later.)

Historical Implementations: Part II

There's still worse.

Historical Implementations: Part II

There's still worse. One cryptographer proclaimed the following, unspecified, undocumented, not-fully-implemented macro "language" for generating artisanal assembly to be...

Historical Implementations: Part II

There's still worse. One cryptographer proclaimed the following, unspecified, undocumented, not-fully-implemented macro "language" for generating artisanal assembly to be...

```
:name:fe:t0:t1:t2:t3:t4:t5:t6:t7:t8:t9:z:out:
fe r:var/r=fe:
enter f:enter/f:>z1=fe#11:
return:nofallthrough:<z_252_3=fe#12:leave:
h=f*g:<f=fe:<g=fe:>h=fe:asm/fe_mul(>h,<f,<g);:
h=f^2^k:<f=fe:>h=fe:#k:asm/fe_sq(>h,<f); for
#(i = 1;i !lt; k;++i) fe_sq(>h,>h);:
:
[...]
fe z_100_50
fe z_100_0
fe z_200_100
fe z_200_0
fe z_250_50
fe z_250_0
fe z_252_2
fe z_252_3

enter pow22523

[...]
z_10_5 = z_5_0^2^5
z_10_0 = z_10_5*z_5_0
z_20_10 = z_10_0^2^10
z_20_0 = z_20_10*z_10_0
z_40_20 = z_20_0^2^20
z_40_0 = z_40_20*z_20_0
z_50_10 = z_40_0^2^10
z_50_0 = z_50_10*z_10_0
z_100_50 = z_50_0^2^50
z_100_0 = z_100_50*z_50_0
z_200_100 = z_100_0^2^100
z_200_0 = z_200_100*z_100_0
z_250_50 = z_200_0^2^50
z_250_0 = z_250_50*z_50_0
z_252_2 = z_250_0^2^2
z_252_3 = z_252_2*z1
return
```

→

```
/* qhasm: z_20_10 = z_10_0^2^10 */
/* asm 1: fe_sq(>z_20_10=fe#2,<z_10_0=fe#1); for (i = 1;i < 10;++i) fe_sq(>z_20_10=fe#2,>z_20_10=fe#2); */
/* asm 2: fe_sq(>z_20_10=t1,<z_10_0=t0); for (i = 1;i < 10;++i) fe_sq(>z_20_10=t1,>z_20_10=t1); */
fe_sq(t1,t0); for (i = 1;i < 10;++i) fe_sq(t1,t1);

/* qhasm: z_20_0 = z_20_10*z_10_0 */
/* asm 1: fe_mul(>z_20_0=fe#2,<z_20_10=fe#2,<z_10_0=fe#1); */
/* asm 2: fe_mul(>z_20_0=t1,<z_20_10=t1,<z_10_0=t0); */
fe_mul(t1,t1,t0);

/* qhasm: z_40_20 = z_20_0^2^20 */
/* asm 1: fe_sq(>z_40_20=fe#3,<z_20_0=fe#2); for (i = 1;i < 20;++i) fe_sq(>z_40_20=fe#3,>z_40_20=fe#3); */
/* asm 2: fe_sq(>z_40_20=t2,<z_20_0=t1); for (i = 1;i < 20;++i) fe_sq(>z_40_20=t2,>z_40_20=t2); */
fe_sq(t2,t1); for (i = 1;i < 20;++i) fe_sq(t2,t2);

/* qhasm: z_40_0 = z_40_20*z_20_0 */
/* asm 1: fe_mul(>z_40_0=fe#2,<z_40_20=fe#3,<z_20_0=fe#2); */
/* asm 2: fe_mul(>z_40_0=t1,<z_40_20=t2,<z_20_0=t1); */
fe_mul(t1,t2,t1);

/* qhasm: z_50_10 = z_40_0^2^10 */
/* asm 1: fe_sq(>z_50_10=fe#2,<z_40_0=fe#2); for (i = 1;i < 10;++i) fe_sq(>z_50_10=fe#2,>z_50_10=fe#2); */
/* asm 2: fe_sq(>z_50_10=t1,<z_40_0=t1); for (i = 1;i < 10;++i) fe_sq(>z_50_10=t1,>z_50_10=t1); */
fe_sq(t1,t1); for (i = 1;i < 10;++i) fe_sq(t1,t1);

/* qhasm: z_50_0 = z_50_10*z_10_0 */
/* asm 1: fe_mul(>z_50_0=fe#1,<z_50_10=fe#2,<z_10_0=fe#1); */
/* asm 2: fe_mul(>z_50_0=t0,<z_50_10=t1,<z_10_0=t0); */
fe_mul(t0,t1,t0);

/* qhasm: z_100_50 = z_50_0^2^50 */
/* asm 1: fe_sq(>z_100_50=fe#2,<z_50_0=fe#1); for (i = 1;i < 50;++i) fe_sq(>z_100_50=fe#2,>z_100_50=fe#2); */
/* asm 2: fe_sq(>z_100_50=t1,<z_50_0=t0); for (i = 1;i < 50;++i) fe_sq(>z_100_50=t1,>z_100_50=t1); */
fe_sq(t1,t0); for (i = 1;i < 50;++i) fe_sq(t1,t1);

/* qhasm: z_100_0 = z_100_50*z_50_0 */
/* asm 1: fe_mul(>z_100_0=fe#2,<z_100_50=fe#2,<z_50_0=fe#1); */
/* asm 2: fe_mul(>z_100_0=t1,<z_100_50=t1,<z_50_0=t0); */
fe_mul(t1,t1,t0);

/* qhasm: z_200_100 = z_100_0^2^100 */
/* asm 1: fe_sq(>z_200_100=fe#3,<z_100_0=fe#2); for (i = 1;i < 100;++i) fe_sq(>z_200_100=fe#3,>z_200_100=fe#3); \
*/
/* asm 2: fe_sq(>z_200_100=t2,<z_100_0=t1); for (i = 1;i < 100;++i) fe_sq(>z_200_100=t2,>z_200_100=t2); */
fe_sq(t2,t1); for (i = 1;i < 100;++i) fe_sq(t2,t2);
```

"THE DEATH OF OPTIMISING COMPILERS"

```
/* qhasm: z_20_10 = z_10_0^2^10 */
/* asm 1: fe_sq(>z_20_10=fe#2,<z_10_0=fe#1); for (i = 1;i < 10;++i) fe_sq(>z_20_10=fe#2,>z_20_10=fe#2); */
/* asm 2: fe_sq(>z_20_10=t1,<z_10_0=t0); for (i = 1;i < 10;++i) fe_sq(>z_20_10=t1,>z_20_10=t1); */
fe_sq(t1,t0); for (i = 1;i < 10;++i) fe_sq(t1,t1);

/* qhasm: z_20_0 = z_20_10*z_10_0 */
/* asm 1: fe_mul(>z_20_0=fe#2,<z_20_10=fe#2,<z_10_0=fe#1); */
/* asm 2: fe_mul(>z_20_0=t1,<z_20_10=t1,<z_10_0=t0); */
fe_mul(t1,t1,t0);

/* qhasm: z_40_20 = z_20_0^2^20 */
/* asm 1: fe_sq(>z_40_20=fe#3,<z_20_0=fe#2); for (i = 1;i < 20;++i) fe_sq(>z_40_20=fe#3,>z_40_20=fe#3); */
/* asm 2: fe_sq(>z_40_20=t2,<z_20_0=t1); for (i = 1;i < 20;++i) fe_sq(>z_40_20=t2,>z_40_20=t2); */
fe_sq(t2,t1); for (i = 1;i < 20;++i) fe_sq(t2,t2);

/* qhasm: z_40_0 = z_40_20*z_20_0 */
/* asm 1: fe_mul(>z_40_0=fe#2,<z_40_20=fe#3,<z_20_0=fe#2); */
/* asm 2: fe_mul(>z_40_0=t1,<z_40_20=t2,<z_20_0=t1); */
fe_mul(t1,t2,t1);

/* qhasm: z_50_10 = z_40_0^2^10 */
/* asm 1: fe_sq(>z_50_10=fe#2,<z_40_0=fe#2); for (i = 1;i < 10;++i) fe_sq(>z_50_10=fe#2,>z_50_10=fe#2); */
/* asm 2: fe_sq(>z_50_10=t1,<z_40_0=t1); for (i = 1;i < 10;++i) fe_sq(>z_50_10=t1,>z_50_10=t1); */
fe_sq(t1,t1); for (i = 1;i < 10;++i) fe_sq(t1,t1);

/* qhasm: z_50_0 = z_50_10*z_10_0 */
/* asm 1: fe_mul(>z_50_0=fe#1,<z_50_10=fe#2,<z_10_0=fe#1); */
/* asm 2: fe_mul(>z_50_0=t0,<z_50_10=t1,<z_10_0=t0); */
```

Historical Implementations: Part II (cont.)

And there's still worse.

Historical Implementations: Part II (cont.)

And there's still 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 (cont.)

And there's still 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 (cont.)

And there's still 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 (cont.)

And there's still 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 (cont.)

And there's still 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.

Rust is excellent

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

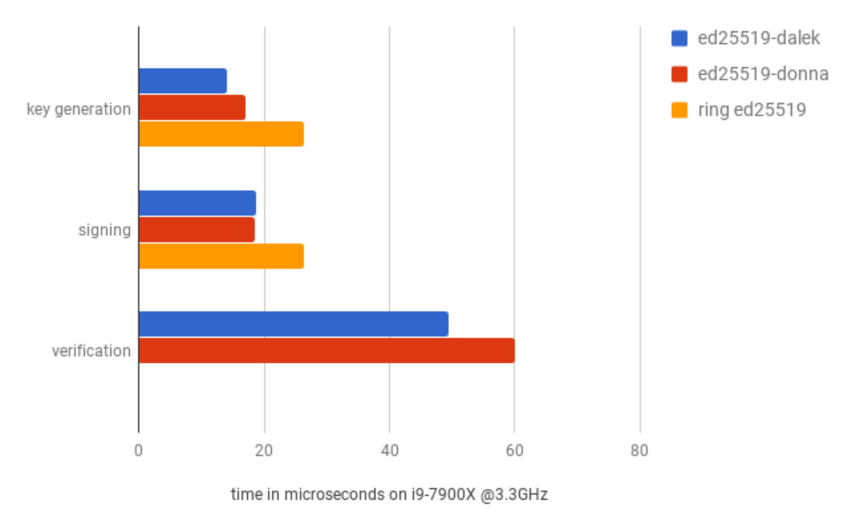
Rust is capable of targeting many platforms, and targeting extremely constrained environments using `no_std`.

`-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).

How fast is it?



Implementing cryptography with `-dalek`

Implementing EdDSA signatures in `ed25519-dalek`

To create an EdDSA signature on a message, m, we first generate our keypair (sk, pk) by choosing the secret scalar, $sk \xleftarrow{\$} \mathbb{Z}/2^{32}\mathbb{Z}$, that is, a random 32-byte string.

Implementing EdDSA signatures in `ed25519-dalek`

To create an EdDSA signature on a message, m, we first generate our keypair (sk, pk) by choosing the secret scalar, $sk \xleftarrow{\$} \mathbb{Z}/2^{32}\mathbb{Z}$, that is, a random 32-byte string.

```
impl SecretKey {
    pub fn generate(csprng: &mut Rng) -> SecretKey {
        let mut sk: SecretKey = SecretKey([0u8; 32]);

        csprng.fill_bytes(&mut sk.0);

        sk
    }
}
```

Implementing EdDSA signatures in `ed25519-dalek`

We then hash sk (RFC8032 specifies SHA-512), take the lower 256 bits of the digest, reduce it as a scalar $x \in \mathbb{Z}/\ell\mathbb{Z}$, and then compute the public key as a point on the curve, $pk \leftarrow xB$ where B is the distinguished basepoint.

Implementing EdDSA signatures in `ed25519-dalek`

We then hash *sk* (RFC8032 specifies SHA-512), take the lower 256 bits of the digest, reduce it as a scalar $x \in \mathbb{Z}/\ell\mathbb{Z}$, and then compute the public key as a point on the curve, $pk \leftarrow xB$ where B is the distinguished basepoint.

```rust
impl PublicKey {
    pub fn from_secret<D>(secret_key: &SecretKey) -> PublicKey
            where D: Digest<OutputSize = U64> + Default {
        let mut h:         D = D::default();
        let mut hash: [u8; 64] = [0u8; 64];
        let     pk:   [u8; 32];
        let mut digest: &mut [u8; 32];

        h.input(secret_key.as_bytes());
        hash.copy_from_slice(h.fixed_result().as_slice());

        digest = array_mut_ref!(&mut hash, 0, 32);
        digest[0]  &= 248;
        digest[31] &= 127;
        digest[31] |= 64;

        pk = (&Scalar(*digest) * &constants::ED25519_BASEPOINT_TABLE).compress_edwards().to_bytes();
        PublicKey(CompressedEdwardsY(pk))
    }
```

Implementing EdDSA signatures in `ed25519-dalek`

To sign the message m, we “expand” sk by hashing it and reduce the low 256-bits to a scalar `sk_expanded` $\in \mathbb{Z}/\ell\mathbb{Z}$. We then hash the concatenation of the high 256-bits of the digest and the message, and reduce the resulting digest into a scalar $r_0 \in \mathbb{Z}/\ell\mathbb{Z}$ which we multiply by the basepoint to produce the point $r \leftarrow r_0 \times B$.

Next, because relying on the sanctity of hash functions is an enormously appropropriate model for real world scenarios, we now compress r into its Edwards y-coordinate (as a 32-byte array). We concentenate the compressed form with the public key (also compressed), as well as — again — the message. The output digest is again reduced as a scalar `hram_digest` $\in \mathbb{Z}/\ell\mathbb{Z}$.

Finally, we compute $s \leftarrow$ `hram_digest` $\times$ `sk_expanded` $+ r_0$ and t as the compressed Edwards y-coordinate of r.

The signature now consists of 32 bytes of t concatenated with 32 bytes of s.

Implementing EdDSA signatures in `ed25519-dalek`

```rust
impl Keypair{
    pub fn sign<D>(&self, message: &[u8]) -> Signature
            where D: Digest<OutputSize = U64> + Default {
        let mut h: D = D::default();
        let mut hash: [u8; 64] = [0u8; 64];
        let mut signature_bytes: [u8; 64] = [0u8; SIGNATURE_LENGTH];

        h.input(self.secret.as_bytes());
        hash.copy_from_slice(h.fixed_result().as_slice());

        let expanded_key_secret: Scalar = expand_key(&hash);

        h = D::default(); h.input(&hash[32..]); h.input(&message);
        hash.copy_from_slice(h.fixed_result().as_slice());

        let mesg_digest: Scalar = Scalar::reduce(&hash);

        let r: ExtendedPoint = &mesg_digest * &constants::ED25519_BASEPOINT_TABLE;

        h = D::default(); h.input(&r.compress_edwards().to_bytes()[..]); h.input(self.public.as_bytes()); h.input(&message);
        hash.copy_from_slice(h.fixed_result().as_slice());

        let hram_digest: Scalar = Scalar::reduce(&hash);

        let s: Scalar = Scalar::multiply_add(&hram_digest, &expanded_key_secret, &mesg_digest);
        let t: CompressedEdwardsY = r.compress_edwards();

        signature_bytes[..32].copy_from_slice(&t.0); signature_bytes[32..64].copy_from_slice(&s.0);
        Signature(*array_ref!(&signature_bytes, 0, 64))
    }

    /// Verify a signature on a message with this keypair's public key.
    pub fn verify<D>(&self, message: &[u8], signature: &Signature) -> bool
            where D: FixedOutput<OutputSize = U64> + BlockInput + Default + Input {
        self.public.verify::<D>(message, signature)
    }
}
```

For signature verification,

Implementing EdDSA signatures in ed25519-dalek

```
impl PublicKey {
    pub fn verify<D>(&self, message: &[u8], signature: &Signature) -> bool
            where D: Digest<OutputSize = U64> + Default {
        use curve25519_dalek::edwards::vartime;

        let mut h: D = D::default();
        let mut a: ExtendedPoint;
        let ao:  Option<ExtendedPoint>;
        let r: ExtendedPoint;
        let digest: [u8; 64];
        let digest_reduced: Scalar;

        if signature.0[63] & 224 != 0 {
            return false;
        }
        ao = self.decompress();

        if ao.is_some() {
            a = ao.unwrap();
        } else {
            return false;
        }
        a = -(&a);

        let top_half:    &[u8; 32] = array_ref!(&signature.0, 32, 32);
        let bottom_half: &[u8; 32] = array_ref!(&signature.0,  0, 32);

        h.input(&bottom_half[..]);
        h.input(&self.to_bytes());
        h.input(&message);

        let digest_bytes = h.fixed_result();
        digest = *array_ref!(digest_bytes, 0, 64);
        digest_reduced = Scalar::reduce(&digest);
        r = vartime::double_scalar_mult_basepoint(&digest_reduced, &a, &Scalar(*top_half));

        if slices_equal(bottom_half, &r.compress_edwards().to_bytes()) == 1 {
            return true
        } else {
            return false
        }
    }
}
```

```rust
impl PublicKey {
    pub fn from_secret<D>(secret_key: &SecretKey) -> PublicKey
            where D: Digest<OutputSize = U64> + Default {
        let mut h:           D = D::default();
        let mut hash: [u8; 64] = [0u8; 64];
        let     pk:   [u8; 32];
        let mut digest: &mut [u8; 32];

        h.input(secret_key.as_bytes());
        hash.copy_from_slice(h.fixed_result().as_slice());

        digest = array_mut_ref!(&mut hash, 0, 32);
        digest[0]  &= 248;
        digest[31] &= 127;
        digest[31] |= 64;

        pk = (&Scalar(*digest) * &constants::ED25519_BASEPOINT_TABLE).compress_edwards().to_bytes();
        PublicKey(CompressedEdwardsY(pk))
    }
```

To sign the message m, we "expand" sk by hashing it and reduce the low 256-bits to a scalar `sk_expanded` $\in \mathbb{Z}/\ell/ZZ$. We then hash the concatenation of the high 256-bits of the digest, the public key (compressed to only the Edwards y-coordinate, as an array of 32 bytes), and the message, and reduce the resulting digest into a scalar $r_0 \in \mathbb{Z}/\ell\mathbb{Z}$

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, called Hyphae.

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, called Hyphae.

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, called Hyphae.

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, called Hyphae.

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();
```

Cryptographic implementations we’ve made using `curve25519-dalek`

Cryptographic implementations we've made using `curve25519-dalek`

- `ed25519-dalek` https://github.com/isislovecruft/ed25519-dalek
  EdDSA signatures in pure Rust.

Cryptographic implementations we've made using `curve25519-dalek`

- `ed25519-dalek` https://github.com/isislovecruft/ed25519-dalek
  EdDSA signatures in pure Rust.
- `x25519-dalek` https://github.com/isislovecruft/x25519-dalek
  X25519 elliptic curve Diffie-Hellman(-Merkle) key exchange

Cryptographic implementations we've made using `curve25519-dalek`

- `ed25519-dalek` https://github.com/isislovecruft/ed25519-dalek
  EdDSA signatures in pure Rust.
- `x25519-dalek` https://github.com/isislovecruft/x25519-dalek
  X25519 elliptic curve Diffie-Hellman(-Merkle) key exchange
- `dalek-rangeproofs` https://github.com/isislovecruft/dalek-rangeproofs
  Back-Maxwell rangeproofs using Borromean Ring signatures

Cryptographic implementations we've made using `curve25519-dalek`

- `ed25519-dalek` https://github.com/isislovecruft/ed25519-dalek
  EdDSA signatures in pure Rust.
- `x25519-dalek` https://github.com/isislovecruft/x25519-dalek
  X25519 elliptic curve Diffie-Hellman(-Merkle) key exchange
- `dalek-rangeproofs` https://github.com/isislovecruft/dalek-rangeproofs
  Back-Maxwell rangeproofs using Borromean Ring signatures
- `dalek-credentials` https://github.com/hdevalence/dalek-credentials
  Algebraic Message Authentication Code (aMAC) and an algebraic-MAC-based, efficient, centralised issuer/verifier, anonymous credential scheme.

Cryptographic implementations we've made using `curve25519-dalek`

- `ed25519-dalek` https://github.com/isislovecruft/ed25519-dalek
  EdDSA signatures in pure Rust.
- `x25519-dalek` https://github.com/isislovecruft/x25519-dalek
  X25519 elliptic curve Diffie-Hellman(-Merkle) key exchange
- `dalek-rangeproofs` https://github.com/isislovecruft/dalek-rangeproofs
  Back-Maxwell rangeproofs using Borromean Ring signatures
- `dalek-credentials` https://github.com/hdevalence/dalek-credentials
  Algebraic Message Authentication Code (aMAC) and an algebraic-MAC-based, efficient, centralised issuer/verifier, anonymous credential scheme.
- `zkp` https://github.com/hdevalence/zkp
  A DSL resembing Camenisch-Stadler notation for proving statements about discrete logarithms in the Decaf group on curve25519

Thank you!

Isis Agora Lovecruft (speaker)	Henry de Valence (dalek coauthor)
@isislovecruft	@hdevalence
isis@patternsinthevoid.net	hdevalence@hdevalence.ca
https://patternsinthevoid.net	https://hdevalence.ca