Building, and building on, Bulletproofs

Henry de Valence
Cathie Yun

JANUARY 30, 2019
Part 1
Building...

Ristretto and the ristretto255 group
Parallel elliptic curve arithmetic
Merlin transcripts for ZKPs
How all pieces fit together

Part 2
...and building on

Constraint system proofs
Cloak: confidential assets protocol
ZkVM: zero-knowledge contracts
Aggregation MPC with session types
Part 1

Building Bulletproofs
Ristretto
Many protocols require a prime-order group

2.4 Notation

Let $G$ denote a cyclic group of prime order $p$,

Sounds good, but how do you actually implement this?
## What kind of elliptic curve should we use?

<table>
<thead>
<tr>
<th>Feature</th>
<th>Weierstrass</th>
<th>Edwards</th>
</tr>
</thead>
<tbody>
<tr>
<td>prime-order group</td>
<td>✔️</td>
<td>✘️</td>
</tr>
<tr>
<td>fastest formulas</td>
<td>✘️</td>
<td>✔️</td>
</tr>
<tr>
<td>complete formulas</td>
<td>✘️</td>
<td>✔️</td>
</tr>
<tr>
<td>easy in constant time</td>
<td>✘️</td>
<td>✔️</td>
</tr>
</tbody>
</table>
What’s wrong with having a small cofactor?

Security proofs of the abstract protocol don’t apply to its implementation;

Full validation is expensive and negates the speedups;

Ad-hoc tweaks like multiplication by the cofactor might work...

...but how do you know where to multiply?

...and it might have subtle effects on the protocol.
Examples of cofactor problems

Ed25519 signature verification **differs between single and batch** verification.

As specified in the RFC, the set of valid signatures is not defined!

Onion Service addresses in Tor had to add **extra validation**.

Cofactor problem: 8 addresses for the same server.

Monero had a **critical vulnerability** due to cofactors.

Cofactor problem: allowed **spending the same amount** 8 times.
Edwards curves are simpler...

...but they push complexity upwards into the protocol.
Fixing the mismatch with Decaf and Ristretto

Mike Hamburg’s Decaf gives an efficient construction of prime-order group from a non-prime-order curve.

Ristretto is a variant of Decaf that works with cofactor-8 curves like Curve25519.

https://ristretto.group
Concrete parameters: the *ristretto255* group

A prime-order group.

Canonical encoding, decoding, hash-to-group.

Can use Curve25519 internally, so relatively **easy to extend** existing implementations.

Implementations can **swap out Curve25519** for a faster curve and remain wire compatible.

There is also an **Internet-Draft**.
Hopefully this sounds good already, but...
Parallel formulas for elliptic curve operations

The fastest formulas for elliptic curves are from Hisil, Wong, Carter, Dawson 2008

The paper also has overlooked formulas for 4 CPUs executing in parallel (impractical).

But: the expensive steps are uniform, so tweaking these formulas allows using SIMD.
Is this strategy fast? Yes!

Cost to compute $aA + bB$ for fixed $B$, variable $A$ (e.g., signature verify)

- FourQ (+endo)
- dalek (IFMA)
- dalek (AVX2)
- FourQ (-endo)
- ed25519-donna
- dalek (u64)
- libsecp* (+endo)
- libsecp* (-endo)

**IFMA**

**AVX2**

**ristretto255**: a prime-order group up to 4x faster than secp256k1.
Merlin transcripts
The Fiat-Shamir Heuristic

Converts an interactive argument into a non-interactive one.

Idea: replace a verifier’s random challenges with a hash of the prover’s messages.

Sounds good, but how do you actually implement this?
Hashing data is kind of complicated!

What if you forget to feed data into the hash?

What if your data is ambiguously encoded in the hash?

How do you handle multi-round protocols?

Where do you put domain separators?

... and many more edge cases.
What if there was a first-class transcript object?

Paper

\[ \mathcal{P}_{\text{IP}} \rightarrow \mathcal{V}_{\text{IP}} : L, R \]

\[ \mathcal{V}_{\text{IP}} : x \xleftarrow[\$]{\mathbb{Z}_p^*} \]

\[ \mathcal{V}_{\text{IP}} \rightarrow \mathcal{P}_{\text{IP}} : x \]

Implementation

```python
transcript.commit_point(b"L", L);
transcript.commit_point(b"R", R);
```

```plaintext
let x = transcript.challenge_scalar(b"x");
```
Merlin: STROBE-based transcripts for ZKPs

Implement protocols as if they were interactive, passing a transcript parameter. Transformation is done in software, not by hand.

Byte-oriented API, automatic message framing.

Easy domain separation.

Automatic sequential composition of proofs.
Hopefully this sounds good already, but...
Case study: bad entropy for Schnorr signatures

The signer (prover) needs to generate a blinding factor (nonce) for signing.

- **Leaking** the blinding factor is fatal.
- **Reusing** the blinding factor is fatal.
- **Leaking just a few bits** of blinding over many signatures is fatal.

This class of attacks presumably also applies to more complex ZKPs.
Transcript-based randomness for provers

Merlin provides defense-in-depth with a Transcript RNG, constructed by:

1. **Cloning** the transcript state (binds to public data).
2. **Rekeying** with witness data (binds to prover’s secrets).
3. **Rekeying** with external entropy (non-deterministic).

Synthetic nonce generation, generalized to arbitrary ZKPs.
Putting all the pieces together
Performance of 64-bit rangeproof verification with SIMD backends in curve25519-dalek

**IFMA**

3x faster than libsecp256k1, 7x faster than Monero.

**AVX2**

2x faster than libsecp256k1, 4.6x faster than Monero.
Part 2

Building on Bulletproofs
Constraint system proofs

What they are, and an example of how to use them
Constraints

**Multiplicative** constraint *(secret-secret multiplication)*:

\[ x \cdot y = z \]

**Linear** constraint *(secret variables with cleartext weights)*:

\[ a \cdot x + b \cdot y + c \cdot z + \ldots = 0 \]
Why constraint systems?

A constraint system can represent any efficiently verifiable program.

A CS proof is proof that all the constraints are satisfied by certain secret inputs.

FURTHER READING
https://medium.com/interstellar/programmable-constraint-systems-for-bulletproofs-365b9feb92f7
Extension: using challenges

Bulletproofs allows for CS construction with **no setup**.

This allows us to select a circuit from a family parameterized by **challenges**.

- Get & use **random challenge scalars** from commitments to variables.
- Make **smaller & more efficient** constraint systems (e.g. shuffle)

Currently **under research**.
Shuffle gadget

Permutation is secret and values are preserved.

A = C
B = D

OR

A = D
B = C
Uses equality of polynomials when roots are permuted.

If the equation holds for random \( x \) then \( \{A, B\} \) must equal \( \{C, D\} \) in any order.
```rust
pub fn two_shuffle<CS: ConstraintSystem>(
    cs: &mut CS,
    A: Variable,
    B: Variable,
    C: Variable,
    D: Variable,
) -> Result<(), R1CSError> {
    cs.specify_randomized_constraints(move |cs| {
        // Get challenge scalar x
        let x = cs.challenge_scalar(b"shuffle challenge");
        // (A - x)*(B - x) = input_mul
        let (_, _, input_mul) = cs.multiply(A - x, B - x);
        // (C - x)*(D - x) = output_mul
        let (_, _, output_mul) = cs.multiply(C - x, D - x);
        // input_mul - output_mul = 0
        cs.constrain(input_mul - output_mul);
        Ok(())
    })
}
```
// Make a prover instance
let mut prover = Prover::new(&bpgens, &pcgens, &mut transcript);

// Create commitments and allocate high-level variables for A, B, C, D
let mut rng = rand::thread_rng();
let (A_com, A_var) = prover.commit(A, Scalar::random(&mut rng));
let (B_com, B_var) = prover.commit(B, Scalar::random(&mut rng));
let (C_com, C_var) = prover.commit(C, Scalar::random(&mut rng));
let (D_com, D_var) = prover.commit(D, Scalar::random(&mut rng));

// Add 2-shuffle gadget constraints to the prover's constraint system
two_shuffle(&mut prover, A_var, B_var, C_var, D_var)?;

// Create a proof
let proof = prover.prove()?;
// Make a verifier instance
let mut verifier = Verifier::new(&bpgens, &pcgens, &mut transcript);

// Allocate high-level variables for A, B, C, D from commitments
let A_var = verifier.commit(A_com);
let B_var = verifier.commit(B_com);
let C_var = verifier.commit(C_com);
let D_var = verifier.commit(D_com);

// Add 2-shuffle gadget constraints to the verifier's constraint system
two_shuffle(&mut verifier, A_var, B_var, C_var, D_var)?;

// Verify the proof
verifier.verify(&proof)
Recap: constraint system proofs

API for **multiplicative** and **linear** constraints

Protocol extension for making **challenges**

**Shuffle gadget** using constraint API and challenges
Cloak

Confidential assets with Bulletproofs
Composition of gadgets in Cloak

Cloak transaction is a combination of smaller gadgets with different roles.

Secretly **reorder** \( N \) values.

Secretly **merge** or **move** two values.

Secretly **split** or **move** two values.

Check that value is **not negative**.
Cloak transaction

Observers cannot tell where values are actually split, merged or moved without modification.

Only the prover knows where values are modified or moved.
Randomly ordered input values are grouped by asset type.
Cloak walkthrough

Values of the same asset type are fully merged together.
Non-zero values are reordered to the top, still grouped by asset type.
Cloak walkthrough

Values are split into target payment amounts.
Cloak walkthrough

Values that were grouped by asset type are shuffled into a random order.
Cloak walkthrough

All values are checked to be non-negative.
Complete 3:3 Cloak transaction

Transactions of the same size are indistinguishable.

SPEC & CODE
https://github.com/interstellar/spacesuit
Cloak performance

Most of the cost is **concentrated in range proofs**, the rest is relatively cheap.

⊗ — one multiplication gate

**SPEC & CODE**
https://github.com/interstellar/spacesuit
ZkVM

Zero-knowledge smart contracts
## Introducing ZkVM

<table>
<thead>
<tr>
<th></th>
<th>BTC</th>
<th>EVM</th>
<th>TxVM</th>
<th>ZkVM</th>
</tr>
</thead>
<tbody>
<tr>
<td>deterministic results</td>
<td>✔</td>
<td>✘</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>expressive language</td>
<td>✘</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>safe environment</td>
<td>✔</td>
<td>✘</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>confidentiality</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✔</td>
</tr>
</tbody>
</table>
ZkVM = TxVM + Bulletproofs

**TxVM**
Linear types **Value** and **Contract** with the guaranteed “law of conservation”.

Contracts implement “**object capabilities**” pattern.

State updates via **deterministic tx log**.

**Bulletproofs**
Encrypted **values** and contract **parameters**.

Contracts built with arbitrary **custom constraints**.

Asset flow protected with **Cloak**.

**UNDER DEVELOPMENT**
https://github.com/interstellar/zkvm
Aggregated proofs

with session types
Proof aggregation

Create **one** aggregated proof for **M** values.

**Smaller and faster** to verify than **M** individual proofs.

Requires an **MPC protocol** with multiple parties and a dealer.
Session types

Encode protocol \textbf{states} into the \textbf{Rust type system}.

Each transition method \textbf{consumes the previous state}.
Can only call method on its corresponding state.

This \textbf{statically ensures correct state transitions}.

No possibility of multiple-evaluation attacks that leak secrets.

\textbf{FURTHER READING}

Part 1
Building...

Ristretto and the ristretto255 group
Parallel elliptic curve arithmetic
Merlin transcripts for ZKPs
How all pieces fit together

Part 2
...and building on

Constraint system proofs
Cloak: confidential assets protocol
ZkVM: zero-knowledge contracts
Aggregation MPC with session types
Further reading

**Bulletproofs paper**

**Interstellar research projects**
https://interstellar.com/protocol

**Ristretto group**
https://ristretto.group

**Dalek API & protocol documentation**
https://doc.dalek.rs
https://doc-internal.dalek.rs