Dive Into Monero

A sassy romp through applied cryptography

Sarang Noether, Ph.D.

Monero Research Lab

8 November 2018

Part I

Prototyping with Python

Data types

Private keys (Scalar): integers in the range $[0, \ell)$
$\ell = 2^{252} + 27742317777372353535851937790883648493$

Public keys (Point): curve points on a curve equivalent to `Curve25519`
$(a, A) \equiv (a, aG)$

The curve point G is a fixed public generator.

Notation: lowercase are scalars, uppercase are curve points

Try it out

```
from dumb25519 import *
a = random_scalar()
A = G*a

b = random_scalar()
B = G*b

assert A+B == G*(a+b)
assert A+A == A*Scalar(2)
assert B-B == Z
```

Motto: Algebra mostly works the way you want it to.

dumb25519 library functionality

Scalars: add, subtract, multiply, invert $(1/a)$

Points: add, subtract, scalar multiply

Hashing: `hash_to_scalar`, `hash_to_point`

Randomness: `random_scalar`, `random_point`

Task: Schnorr signature

Key generation: $(x, X) = (x, xG)$

Signing: Choose message M and random scalar r
Let $c = \mathrm{H_s}(rG, M)$
Let $s = r - xc$
Signature is (s, c)

Verifying: Ensure $\mathrm{H_s}(sG + cX, M) = c$

Write a tool that performs Schnorr signature generation and verification. Write unit tests.

Task: Pedersen commitment

Let G and H be curve points. Generate H using a hash: $\mathsf{H_p}(\ldots)$

Commitment: $\mathrm{Com}(x, r) = xG + rH$

Suppose we can commit to a transaction amount x using a Pedersen commitment with randomness r.

Write a tool that, given a collection of commitments and randomness (but not the amounts), determines if the sum of the committed amounts is zero. Write unit tests.

Part II

Blockchain Data with Python

Quick setup

```
sudo apt install python-requests
```

This library lets you easily obtain Python data structures from API requests that return JSON data:

```
data = requests.get('https://example.com/api/gimme_data').json()
```

Task: Block information from explorer

Base URL: `https://xmrchain.net/api`

Recent blocks: `/transactions?limit=10`

Transaction by hash: `/transaction/A0B1C2D3E4F5...`

Write a tool that uses these endpoints to obtain distributions for the previous 10 blocks:

- Transactions per block
- Inputs per transaction
- Outputs per transaction

Note that coinbase transactions do not have inputs.

Task: Block information from daemon

Documentation:
`https://www.getmonero.org/resources/developer-guides/daemon-rpc.html`

Host: local daemon or `http://node.moneroworld.com:18089`

The RPC interfaces vary depending on method.

Write a tool that obtains information about recent blocks.

- `get_height`: current block height
- `get_block`: data on a specific block
- `get_transactions`: data on specific transactions

Part III

Unit Tests with C++

“Quick” setup

```
git clone https://github.com/monero-project/monero
```

(follow instructions on GitHub to build)

```
cd build/Linux/release-v0.XX/release/tests/performance_tests
./performance_tests --filter=\*test_bullet\*
```

Task: multiexponentiation timing

Monero uses dedicated algorithms to compute **multiexponentiations** (multiexp) of this form:

$$a_1 P_1 + a_2 P_2 + ... + a_n P_n$$

There are three algorithms available in the codebase: Bos-Coster, Straus, and Pippenger.

Write or run a set of performance tests to determine the optimal algorithm for a multiexp of size $n = 4$, size $n = 128$, and size $n = 1024$. (We used timing tests like these to optimize the verification speed of Bulletproofs.)