Optimizations for Bitcoin key cracking

Ryan Castellucci - White Ops

Slides: https://rya.nc/d9 Video (when released): https://rya.nc/r2

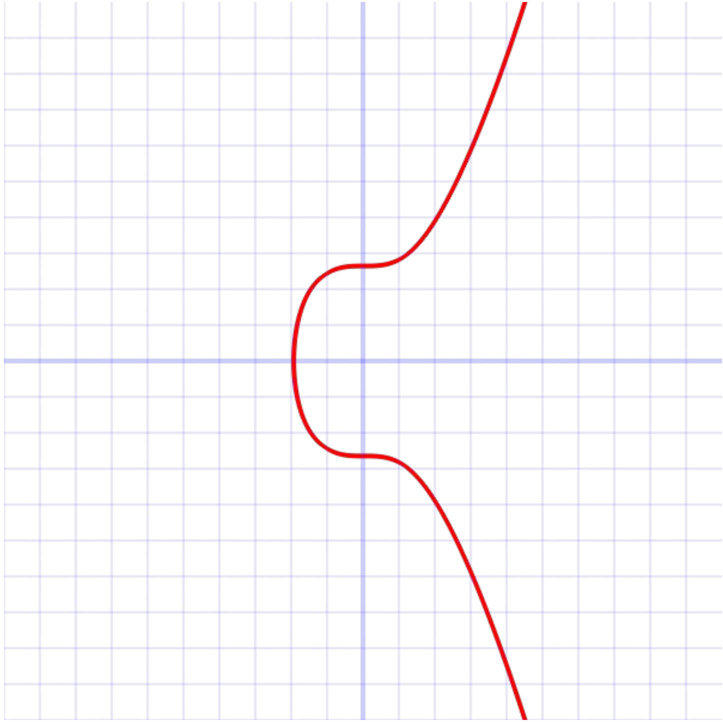
Brainflayer - An increasingly general Cryptocurrency cracker

- Originally released at DEF CON 23 for cracking “Brain Wallets”
- Now over four times the original performance (for Brain Wallets)
- Various modes with special case optimizations
- Hardened algorithms (WarpWallet, Brainwallet.io)
- Ethereum support (Keccak, live.ether.camp, Parity/Direct ICAP, Quorum)
- Single public key P2SH addresses
- Public keys *x* coordinates (works on signature nonces)
- Raw private key input (hex, for external tools)
- Raw P2SH script input (hex, for external tools)
- Confirms bloom filter hits to eliminate false positives

Bitcoin keys

- Elliptic Curve Digital Signature Algorithm with secp256k1 parameters
- Points on curve (used as public keys) satisfy $y^2 = x^3 + 7$ in a finite field (p)
- Prime field, $2^{256} - 2^{32} - 977$
- Private keys are integers in a different finite field (n)
  - Also prime: `0xfffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141`
- Public keys are points formed by multiplying a private key by the “base point”
  - x = `0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798`
  - y = `0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8`
- Going from public to private key requires finding the “discrete logarithm”
- This is Hard without a quantum computer. Bitcoin has some extra protection.
- See https://rya.nc/d8 if you want more details

Bitcoin keys



Bitcoin key generation

- Just generate an arbitrary 256 bit (32 byte) value, it’ll *probably* work
    - Unless it’s 0 or *n*.
- Weakly random and “clever” key generation is easy.

Bitcoin address generation

- Addresses are really a convenient shorthands for payment scripts
  - https://en.bitcoin.it/wiki/Script
- If reuse is avoided, provides some protection against quantum attacks
- Addresses starting with 1: RIPEMD160(SHA256(Public Key)) - P2PKH
- Addresses starting with 3: RIPEMD160(SHA256(Script)) - P2SH
- Other raw scripts are possible, but don’t work with SW that needs an address
- Ethereum uses the last 160 bits of Keccak256(Public Key)
- Bitcoin allows uncompressed and compressed representations of public keys
  - Uncompressed: 0x04 <32 byte x coordinate> <32 byte y coordinate>
  - Compressed: (0x02|0x03) <32 byte x coordinate>

General optimizations

- Minimize memory copies within reason
- Inline functions where possible
- Allocate memory at startup
- Static buffer initialization
- Threads are hard
- Map data files into shared memory

Faster hashing - Intel SHA256 assembly

- Intel’s Whitepaper: https://rya.nc/r3
- Runtime selection between SHA-NI, AVX2, AVX, SSSE3 or C implementation
- Only the transform function is accelerated
- Incremental hash computation only partially supported
- An optimized RIPEMD160 implementation would be nice
- Also SHA3/Keccak?

Faster hashing - Precomputed padding

- “Hello world!” and “Hello world!\x00” should have different values
- Hash([DATA][\x80\x00\x00…\x00\x00][SIZE]) - pad to fill up “block”
- RIPEMD160 only needed with a 256 bit SHA256 hash as input
- Statically allocate a single pre-padded RIPEMD160 block
- Write SHA256 hash directly to block

Faster hashing - Precomputed padding

```
/* static padding for 256 bit input */
static uint8_t rmd160_256[64] = {
  0x00, 0x00, 0x00, 0x00,  0x00, 0x00, 0x00, 0x00, // input
  0x00, 0x00, 0x00, 0x00,  0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00,  0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00,  0x00, 0x00, 0x00, 0x00,
  0x80, 0x00, 0x00, 0x00,  0x00, 0x00, 0x00, 0x00, // fixed padding
  0x00, 0x00, 0x00, 0x00,  0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00,  0x00, 0x00, 0x00, 0x00,
  /* length 256 bits, little endian uint64_t */
  0x00, 0x01, 0x00, 0x00,  0x00, 0x00, 0x00, 0x00  // fixed length
};
```

Faster hashing - Precomputed padding

```
void Hash160(uint8_t hash[], const uint8_t data[], size_t len) {
  SHA2_256(rmd160_256, data, len);
  ripemd160_rawcompress(rmd160_256, hash);
}
```

Faster hashing - Precomputed padding

```
// caller is responsible for padding
inline void Hash160_Raw(uint8_t hash[], const uint8_t data[], uint64_t nblk) {
  uint32_t *rmd160_in = (uint32_t *)rmd160_256;
  uint32_t state[8] = { 0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a,
                        0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19 };
  sha256_transform_func(state, data, nblk);
  rmd160_in[0] = be32(state[0]); rmd160_in[1] = be32(state[1]);
  rmd160_in[2] = be32(state[2]); rmd160_in[3] = be32(state[3]);
  rmd160_in[4] = be32(state[4]); rmd160_in[5] = be32(state[5]);
  rmd160_in[6] = be32(state[6]); rmd160_in[7] = be32(state[7]);
  ripemd160_rawcompress(rmd160_256, hash);
}
```

Faster hashing - Precomputed padding

```
/* static padding for length 264 bits, big endian uint64_t */
static uint8_t input33[64] = {
  0x00, 0x00, 0x00, 0x00,  0x00, 0x00, 0x00, 0x00, // input
  0x00, 0x00, 0x00, 0x00,  0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00,  0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00,  0x00, 0x00, 0x00, 0x00,
  0x00, 0x80, 0x00, 0x00,  0x00, 0x00, 0x00, 0x00, // fixed padding
  0x00, 0x00, 0x00, 0x00,  0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00,  0x00, 0x00, 0x00, 0x00,
  /* length 264 bits, big endian uint64_t */
  0x00, 0x00, 0x00, 0x00,  0x00, 0x00, 0x01, 0x08 // fixed length
};
```

Faster hashing - Precomputed padding

```
void Hash160_33(uint8_t hash[], const uint8_t data[]) {
  memcpy(input33, data, 33);
  Hash160_Raw(hash, input33, 1);
}
```

Attacking multiple cryptocurrencies at once

- Many altcoins (e.g. Dogecoin, Litecoin, Defcoin, etc) are very close
- Mix-and-match pubkey hashes, figure out which altcoin was hit in post
- Ethereum uses the same elliptic curve, but a different address format
- Compute public key, apply multiple transforms

Public key transforms

- Selected via Brainflayer's -c flag
- Active transforms in a null terminated array of structs
  - Function pointer to transform function
  - One character identifier for transform type

```
typedef struct pubhashfn_s {
  void (*fn)(hash160_t *, const unsigned char *);
  char id;
} pubhashfn_t;
```

Public key transforms - Uncompressed

```c
static void uhash160(hash160_t *h, const unsigned char *upub) {
  Hash160_65(h->uc, upub);
}
```

Public key transforms - Compressed

```
static void chash160(hash160_t *h, const unsigned char *upub) {
  unsigned char *cpub = (unsigned char *)upub;
  unsigned char hdr;

  hdr = upub[0]; // save public key header byte
  // quick and dirty public key compression
  cpub[0] = 0x02 | (upub[64] & 0x01); // 0x02 if even, 0x03 if odd

  Hash160_33(h->uc, cpub);
  cpub[0] = hdr; // restore public key header byte
}
```

Public key transforms - Ethereum

```c
static void ehash160(hash160_t *h, const unsigned char *upub) {
  SHA3_256_CTX ctx;
  unsigned char hash[SHA256_DIGEST_LENGTH];
  /* compute hash160 for uncompressed public key */
  /* keccak_256_last160(pub) */
  KECCAK_256_Init(&ctx);
  KECCAK_256_Update(&ctx, upub+1, 64);
  KECCAK_256_Final(hash, &ctx);
  memcpy(h->uc, hash+12, 20);
}
```

Public key transforms - Truncated x coordinate

```
static void xhash160(hash160_t *h, const unsigned char *upub) {
  memcpy(h->uc, upub+1, 20);
}
```

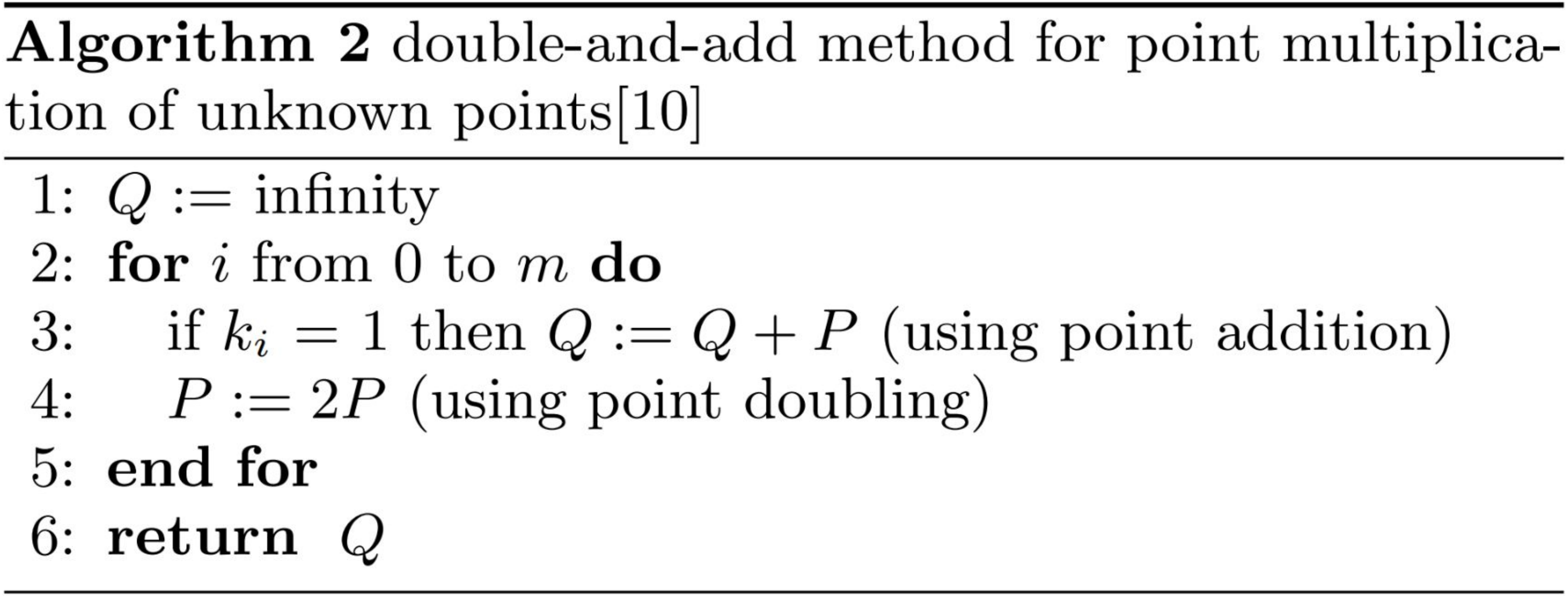
Public key transforms - P2SH 1-of-1 multisig

```c
static void mhash160(hash160_t *h, const unsigned char *upub) {
  static unsigned char spk[] = {
    OP_1, 33, // 33 byte public key
    0,
    0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0,
    OP_1, OP_CHECKMULTISIG
  };
  memcpy(spk+2, upub, 33);
  spk[2] = 0x02 | (upub[64] & 0x01);
  Hash160_37(h->uc, spk);
}
```

Relative speed of operations (fast to slow)

- Adding two field elements
- Multiplying two field elements
- Doubling a point
- Adding two points together
- Inverting a field element
- Multiplying an integer by a point

Naive point multiplication



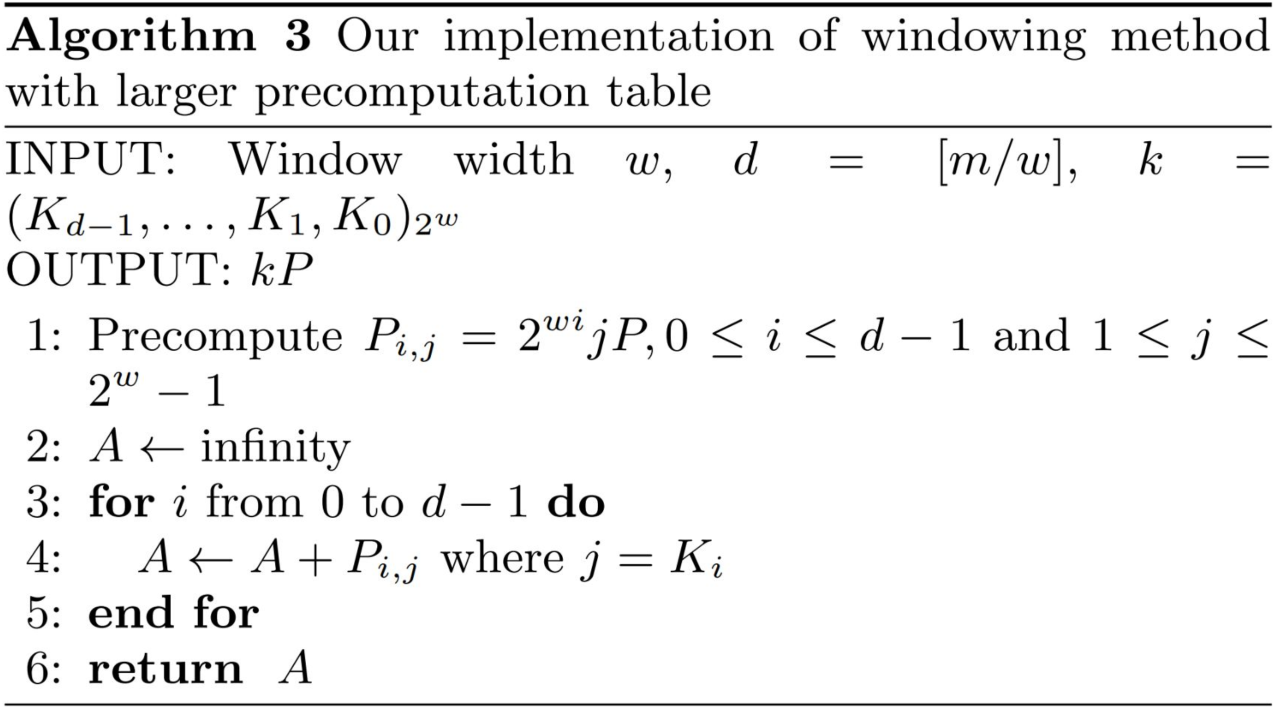
Algorithm 2 double-and-add method for point multiplication of unknown points[10]

```
$Q :=$ infinity
for $i$ from 0 to $m$ do
    if $k_i = 1$ then $Q := Q + P$ (using point addition)
    $P := 2P$ (using point doubling)
end for
return $Q$
```

Naive point multiplication

- Start with an accumulator holding the additive identity point
    - Infinity, rather than zero
- Take the binary representation of an integer (private key)
- Starting with LSB…
- Add base point to accumulator if 1
- Double the base point
- If the next bit is 1, add the doubled point

Partial precomputation for point multiplication



Algorithm 3 Our implementation of windowing method with larger precomputation table

INPUT: Window width w, $d = [m/w]$, $k = (K_{d-1}, \ldots, K_1, K_0)_{2^w}$

OUTPUT: kP

```
Precompute $P_{i,j} = 2^{wi} jP, 0 \leq i \leq d-1$ and $1 \leq j \leq 2^w - 1$
$A \leftarrow$ infinity
for $i$ from 0 to $d-1$ do
    $A \leftarrow A + P_{i,j}$ where $j = K_i$
end for
return $A$
```

Partial precomputation for point multiplication

- Initialize a 2d array of points
- Fill the 0th column with points 0 through 2^n-1
- Fill rest of columns with values from previous column times 2^n
- Point multiply becomes ceil(256/n) lookup-and-add operations
- Memory use approx 88 × ceil(256/n) × 2^n bytes
- Speed scales decently (RAM becomes a limit)

Partial precomputation for point multiplication

- libsecp256k1 does this already with a 4 bit window
- Brainflayer defaults to a 16 bit window (calculated on startup, ~88MiB RAM)
- 24 bit window is good (~16GiB RAM) ~2.5x speedup
- 25 bit window mostly wastes memory
- 26 bit window is slightly faster (~55GiB RAM)
- 29 bit window for further improvement, but takes ~396GiB RAM
- 32 bit window takes ~2.8TiB RAM
- Generation requires ~triple the memory
- Use standalone generation tool to create a file, then load from disk
- Code from Guangyan Song @ UCL - Paper: https://rya.nc/r4

Affine vs Jacobian coordinates

- Affine (x, y) representing a point $y^2 = x^3 + 7$
- Jacobian (x, y, z) represent the affine point $(x/z^2, y/z^3)$
- Affine > Jacobian: (x, y, 1)
- Computations done in Jacobian because it’s faster
- The z value does not stay 1

Arithmetic with prime finite fields - Addition

- How about a field order of 5?

Arithmetic with prime finite fields - Addition

- How about a field order of 5?
- 1 + 1 = 2

Arithmetic with prime finite fields - Addition

- How about a field order of 5?
- 1 + 1 = 2
- 2 + 2 = 4

Arithmetic with prime finite fields - Addition

- How about a field order of 5?
- 1 + 1 = 2
- 2 + 2 = 4
- 3 + 3 = 1
  - Really, 3 + 3 = 6 % (modulo) 5 = 1

Arithmetic with prime finite fields - Addition

- How about a field order of 5?
- 1 + 1 = 2
- 2 + 2 = 4
- 3 + 3 = 1
  - Really, 3 + 3 = 6 % (modulo) 5 = 1
- 4 + 1 = 0
- 0 + 3 = 3
- 4 + 4 = 3
- 3 + 2 = 0
- 3 + 4 = 2

Arithmetic with prime finite fields - Subtraction

- Still using field order of 5
- 1 - 1 = 0
- 3 - 3 = 0
- 4 - 2 = 2
- 0 - 1 = 4
- 2 - 4 = 3
- 1 - 4 = 2
- 3 - 2 = 1
- 0 - 4 = 1

Arithmetic with prime finite fields - Multiplication

- Still using field order of 5
- 1 × 1 = 1
- 2 × 2 = 4
- 3 × 3 = 4
- 4 × 4 = 1
- 2 × 3 = 1
- 4 × 2 = 3
- 3 × 4 = 2
- 4 × 1 = 4

Arithmetic with prime finite fields - Division

- Still using field order of 5
- …

Arithmetic with prime finite fields - Division

- Still using field order of 5
- …
- Can’t do this directly
- However, $y \div x \equiv y \times 1/x \equiv y \times x^{-1}$

Arithmetic with prime finite fields - Inversion

- Still using field order of 5
- AKA “multiplicative inverse”
- Computed with “extended Euclidean algorithm” (see https://rya.nc/d7)
- x × y = 1, given x find y (the inverse)
- 1 × 1 = 1
- 2 × 3 = 1
- 3 × 2 = 1
- 4 × 4 = 1
- 0 … don’t divide by zero

Arithmetic with prime finite fields - Division (sort of)

- Still using field order of 5
- $4 \div 2 \equiv 4 \times 2^{-1} \equiv 4 \times 1/2 \equiv 4 \times 3 = 2$
- $2 \div 2 \equiv 2 \times 2^{-1} \equiv 2 \times 1/2 \equiv 2 \times 3 = 1$
- $3 \div 3 \equiv 3 \times 3^{-1} \equiv 3 \times 1/3 \equiv 3 \times 2 = 1$
- $4 \div 4 \equiv 4 \times 4^{-1} \equiv 4 \times 1/4 \equiv 4 \times 4 = 1$
- $4 \div 3 \equiv 4 \times 3^{-1} \equiv 4 \times 1/3 \equiv 4 \times 2 = 3$
- $3 \div 2 \equiv 3 \times 2^{-1} \equiv 3 \times 1/2 \equiv 3 \times 3 = 4$
- $2 \div 3 \equiv 2 \times 3^{-1} \equiv 3 \times 1/3 \equiv 2 \times 2 = 4$
- $1 \div 4 \equiv 1 \times 4^{-1} \equiv 1 \times 1/4 \equiv 1 \times 4 = 4$
- $1 \div 3 \equiv 1 \times 3^{-1} \equiv 1 \times 1/3 \equiv 1 \times 2 = 2$

Converting Jacobian coordinates back to affine

- Need to do “division”
- To “divide” we need to find the inverse of z
- Finding an inverse is slow compared to other operations

Algebra to the rescue!

- We can do one inversion for a group of z values
- ...plus three multiplications per z
- Way faster
- ~1.7x speedup in Brain Wallet mode
- ~16x speedup in incremental mode

Batch inversion

z_0

z_1

z_2

z_3

Batch inversion

z_0
z_1
z_2
z_3

z_0

z_0

Batch inversion

z_0
z_1
z_2
z_3

z_0
$z_1 \times z_0$

z_0
$z_0 z_1$

Batch inversion

z_0
z_1
z_2
z_3

z_0
$z_1 \times z_0$
$z_2 \times z_0 z_1$

z_0
$z_0 z_1$
$z_0 z_1 z_2$

Batch inversion

z_0

z_1

z_2

z_3

z_0

$z_1 \times z_0$

$z_2 \times z_0z_1$

$z_3 \times z_0z_1z_2$

z_0

z_0z_1

$z_0z_1z_2$

$z_0z_1z_2z_3$

Batch inversion

z_0
z_1
z_2
z_3

z_0
$z_1 \times z_0$
$z_2 \times z_0z_1$
$z_3 \times z_0z_1z_2$

z_0
z_0z_1
$z_0z_1z_2$
$z_0z_1z_2z_3$

$1/z_0z_1z_2z_3$

Batch inversion

z_0
z_1
z_2
z_3

z_0
$z_1 \times z_0$
$z_2 \times z_0z_1$
$z_3 \times z_0z_1z_2$

$z_0z_1z_2/z_0z_1z_2z$
3

$z_0z_1z_2 \times 1/z_0z_1z_2z_3$

z_0
z_0z_1
$z_0z_1z_2$
$z_0z_1z_2z_3$

$1/z_0z_1z_2z_3$

Batch inversion

z_0
z_1
z_2
z_3

z_0
$z_1 \times z_0$
$z_2 \times z_0z_1$
$z_3 \times z_0z_1z_2$

$\cancel{z_0}\cancel{z_1}\cancel{z_2}/\cancel{z_0}\cancel{z_1}\cancel{z_2}z$
3

$\cancel{z_0}\cancel{z_1}\cancel{z_2} \times 1/\cancel{z_0}\cancel{z_1}\cancel{z_2}z_3$

z_0
z_0z_1
$z_0z_1z_2$
$z_0z_1z_2z_3$

$1/z_0z_1z_2z_3$

$1/z_3$

Batch inversion

z_0
z_1
z_2
z_3

z_0
z_1 × z_0
z_2 × z_0z_1
z_3 × $z_0z_1z_2$

$z_3/z_0z_1z_2z_3$

z_3 ×
$1/z_0z_1z_2z_3$

z_0
z_0z_1
$z_0z_1z_2$
$z_0z_1z_2z_3$

$1/z_0z_1z_2z_3$

$1/z_3$

Batch inversion

z_0
z_1
z_2
z_3

z_0
$z_1 \times z_0$
$z_2 \times z_0z_1$
$z_3 \times z_0z_1z_2$

$\cancel{z_3}/z_0z_1z_2\cancel{z_3}$

$\cancel{z_3} \times$
$1/z_0z_1z_2\cancel{z_3}$

z_0
z_0z_1
$z_0z_1z_2$
$z_0z_1z_2z_3$

$1/z_0z_1z_2$

$1/z_3$

Batch inversion

z_0
z_1
z_2
z_3

z_0
$z_1 \times z_0$
$z_2 \times z_0z_1$
$z_3 \times z_0z_1z_2$

$z_0z_1/z_0z_1z_2$

$z_0z_1 \times$
$1/z_0z_1z_2$

z_0
z_0z_1
$z_0z_1z_2$
$z_0z_1z_2z_3$

$1/z_0z_1z_2$

$1/z_3$

Batch inversion

z_0
z_1
z_2
z_3

z_0
$z_1 \times z_0$
$z_2 \times z_0z_1$
$z_3 \times z_0z_1z_2$

$\cancel{z_0}\cancel{z_1}/\cancel{z_0}\cancel{z_1}z_2$

$\cancel{z_0}\cancel{z_1} \times$
$1/\cancel{z_0}\cancel{z_1}z_2$

z_0
z_0z_1
$z_0z_1z_2$
$z_0z_1z_2z_3$

$1/z_0z_1z_2$

$1/z_2$
$1/z_3$

Batch inversion

z_0
z_1
z_2
z_3

z_0
$z_1 \times z_0$
$z_2 \times z_0z_1$
$z_3 \times z_0z_1z_2$

$z_2/z_0z_1z_2$

$z_2 \times 1/z_0z_1z_2$

z_0
z_0z_1
$z_0z_1z_2$
$z_0z_1z_2z_3$

$1/z_0z_1z_2$

$1/z_2$
$1/z_3$

Batch inversion

z_0
z_1
z_2
z_3

z_0
$z_1 \times z_0$
$z_2 \times z_0 z_1$
$z_3 \times z_0 z_1 z_2$

$\cancel{z_2} / z_0 z_1 \cancel{z_2}$

$\cancel{z_2} \times 1/z_0 z_1 \cancel{z_2}$

z_0
$z_0 z_1$
$z_0 z_1 z_2$
$z_0 z_1 z_2 z_3$

$1/z_0 z_1$

$1/z_2$
$1/z_3$

Batch inversion

z_0
z_1
z_2
z_3

z_0
$z_1 \times z_0$
$z_2 \times z_0 z_1$
$z_3 \times z_0 z_1 z_2$

$z_0 / z_0 z_1$

$z_0 \times 1/z_0 z_1$

z_0
$z_0 z_1$
$z_0 z_1 z_2$
$z_0 z_1 z_2 z_3$

$1/z_0 z_1$

$1/z_2$
$1/z_3$

Batch inversion

z_0
z_1
z_2
z_3

z_0
$z_1 \times z_0$
$z_2 \times z_0 z_1$
$z_3 \times z_0 z_1 z_2$

$\cancel{z_0}/\cancel{z_0}z_1$

$\cancel{z_0} \times 1/\cancel{z_0}z_1$

z_0
$z_0 z_1$
$z_0 z_1 z_2$
$z_0 z_1 z_2 z_3$

$1/z_0 z_1$

$1/z_1$
$1/z_2$
$1/z_3$

Batch inversion

z_0
z_1
z_2
z_3

z_0
$z_1 \times z_0$
$z_2 \times z_0z_1$
$z_3 \times z_0z_1z_2$

z_1/z_0z_1

$z_1 \times 1/z_0z_1$

z_0
z_0z_1
$z_0z_1z_2$
$z_0z_1z_2z_3$

$1/z_0z_1$

$1/z_1$
$1/z_2$
$1/z_3$

Batch inversion

z_0	z_0	z_0	$1/z_0$
z_1	$z_1 \times z_0$	z_0z_1	$1/z_1$
z_2	$z_2 \times z_0z_1$	$z_0z_1z_2$	$1/z_2$
z_3	$z_3 \times z_0z_1z_2$	$z_0z_1z_2z_3$	$1/z_3$
	$\cancel{z_1}/z_0\cancel{z_1}$	$1/z_0$	
	$\cancel{z_1} \times 1/z_0\cancel{z_1}$		

Batch inversion

z_0	z_0	z_0	$1/z_0$
z_1	$z_1 \times z_0$	z_0z_1	$1/z_1$
z_2	$z_2 \times z_0z_1$	$z_0z_1z_2$	$1/z_2$
z_3	$z_3 \times z_0z_1z_2$	$z_0z_1z_2z_3$	$1/z_3$

Don’t care about side channels

- Older version of libsecp256k1 used
- Faster because many routines weren’t constant time
- TODO: Actively implement more faster algorithms that are not constant time

Special cases - Incremental search

- Secp256k1 keypairs are homomorphic for addition

Special cases - Incremental search

- Secp256k1 keypairs are homomorphic for addition
- Wat?

Special cases - Incremental search

- Secp256k1 keypairs are homomorphic for addition
- Wat?
- Pubkey($priv_a$ + $priv_b$) ≡ Pubkey($priv_a$) + Pubkey($priv_b$)
- Adding public points together much faster that Pubkey(priv)

Special cases - Keys from byte stream

- Say you want to search for keys within a bunch of bytes...
- Pubkey × 2 even faster than Pubkey + Pubkey
- Build “addition” table, Pubkey(0..255)
- Build “subtraction” table, Pubkey(0×2^{248}..255×2^{248})
- Pubkey - sub[high byte]
- double(double(double(double(double(double(double(double(Pubkey))))))))
- Pubkey + add[new byte]

Special cases - Keys from byte stream

```
bc4a0be509791adc771e286a
```

Special cases - Keys from byte stream

```
bc4a0be509791adc771e286a - bc0000000000000000000000
004a0be509791adc771e286a
```

Special cases - Keys from byte stream

```
bc4a0be509791adc771e286a - bc0000000000000000000000
004a0be509791adc771e286a × 256 (eight doublings)
4a0be509791adc771e286a00
```

Special cases - Keys from byte stream

```
bc4a0be509791adc771e286a - bc0000000000000000000000
004a0be509791adc771e286a × 256 (eight doublings)
4a0be509791adc771e286a00 + 00000000000000000000001a
4a0be509791adc771e286a1a
```

Special cases - Keys from byte stream

```
bc4a0be509791adc771e286a - bc0000000000000000000000
004a0be509791adc771e286a × 256 (eight doublings)
4a0be509791adc771e286a00 + 00000000000000000000001a
4a0be509791adc771e286a1a - 4a0000000000000000000000
000be509791adc771e286a1a
```

Special cases - Keys from byte stream

```
bc4a0be509791adc771e286a - bc0000000000000000000000
004a0be509791adc771e286a × 256 (eight doublings)
4a0be509791adc771e286a00 + 00000000000000000000001a
4a0be509791adc771e286a1a - 4a0000000000000000000000
000be509791adc771e286a1a × 256 (eight doublings)
0be509791adc771e286a1a00
```

Special cases - Keys from byte stream

```
bc4a0be509791adc771e286a - bc0000000000000000000000
004a0be509791adc771e286a × 256 (eight doublings)
4a0be509791adc771e286a00 + 00000000000000000000001a
4a0be509791adc771e286a1a - 4a0000000000000000000000
000be509791adc771e286a1a × 256 (eight doublings)
0be509791adc771e286a1a00 + 0000000000000000000000fe
0be509791adc771e286a1afe
```

Original bloom filter

- Bloom filters normally read/set bits by Hash(0, input), Hash(1, input) etc
- In the general case, this is needed
- We’re dealing with a special case
- Can just select different combinations of bytes

Original bloom filter

```
/* 2^32 bits */
#define BLOOM_SIZE (512*1024*1024)

#define BLOOM_SET_BIT(N) (bloom[(N)>>3] = bloom[(N)>>3] | (1<<((N)&7)))
#define BLOOM_GET_BIT(N) ( ( bloom[(N)>>3]>>((N)&7) )&1)

#define BH01(N) (N[1])
#define BH07(N) (N[2]<<16|N[3]>>16)
#define BH10(N) (N[0]<< 8|N[1]>>24)
#define BH19(N) (N[4]<<24|N[0]>> 8)
```

New bloom filter

- Too many addresses now for a 512MiB bloom filter to work well
- Fixed 160 bit (20 byte) input - choose three 16 bit chunks as the “hash”
- 48 bit value for each “hash”
- Bitwise AND mask to range
- Work in progress...

New bloom filter

```
uint64_t h160h_chk(uint8_t *mem, void *h160, uint8_t n, uint64_t mask) {
  const uint16_t *S = h160;
  uint64_t h;

  switch(n) {
    case 45:
      h = ((((uint64_t)S[4])<<32)|(((uint64_t)S[9])<<16)|(((uint64_t)S[5])))&mask;
      if (((mem[(h)>>3]>>((h)&7))&1) == 0) { return 0; }
    case 44:
      h = ((((uint64_t)S[8])<<32)|(((uint64_t)S[6])<<16)|(((uint64_t)S[1])))&mask;
      if (((mem[(h)>>3]>>((h)&7))&1) == 0) { return 0; }
    case 43:
      h = ((((uint64_t)S[2])<<32)|(((uint64_t)S[0])<<16)|(((uint64_t)S[4])))&mask;
```

Bloom filter false positives

```
$ ./hex2blf btc_468762.hex btc_468762.blf
[*] Initializing bloom filter...
[*] Loading hash160s from 'btc_468762.hex' 100.0%
[*] Loaded 264265017 hashes
[*] False positive rate: ~9.981e-04 (1 in ~1.002e+03)
[*] Writing bloom filter to 'btc_468762.blf'...
[+] Success!
```

Bloom filter false positives - Search an array

- Everybody knows a binary search is fastest on an ordered array, right?
- Best case: O(1) Average case: O(log n) Worst case: O(log n)

Bloom filter false positives - Search a list

- Everybody knows a binary search is fastest on an ordered array, right?
- Best case: O(1) Average case: O(log n) Worst case: O(log n)

```
                         \/

|02|16|2a|31|4f|51|5a|60|68|9a|a2|b7|c9|dd|e5|fd|
```

Bloom filter false positives - Search a list

- Everybody knows a binary search is fastest on an ordered array, right?
- Best case: O(1) Average case: O(log n) Worst case: O(log n)

```
             \/

|02|16|2a|31|4f|51|5a|60|68|9a|a2|b7|c9|dd|e5|fd|
```

Bloom filter false positives - Search a list

- Everybody knows a binary search is fastest on an ordered array, right?
- Best case: O(1) Average case: O(log n) Worst case: O(log n)

```
                   \/

|02|16|2a|31|4f|51|5a|60|68|9a|a2|b7|c9|dd|e5|fd|
```

Bloom filter false positives - Search a list

- Everybody knows a binary search is fastest on an ordered array, right?
- Best case: O(1) Average case: O(log n) Worst case: O(log n)

```
                \/

|02|16|2a|31|4f|51|5a|60|68|9a|a2|b7|c9|dd|e5|fd|
```

Bloom filter false positives - Interpolation search

- Assume uniform distribution within search range
- Search at estimate based on low and high values and number of elements
- Best case: O(1) Average case: O(log log n) Worst case: O(n)

Bloom filter false positives - Interpolation search

- Assume uniform distribution within search range
- Search at estimate based on low and high values and number of elements
- Best case: O(1) Average case: O(log log n) Worst case: O(n)

```
                   \/

|02|16|2a|31|4f|51|5a|60|68|9a|a2|b7|c9|dd|e5|fd|
```

Bloom filter false positives - Interpolation search

- Assume uniform distribution within search range
- Search at estimate based on low and high values and number of elements
- Best case: O(1) Average case: O(log log n) Worst case: O(n)

```
                \/

|02|16|2a|31|4f|51|5a|60|68|9a|a2|b7|c9|dd|e5|fd|
```

Questions?

https://rya.nc/brainflayer

Slides: https://rya.nc/d9 Video (when released): https://rya.nc/r2

https://rya.nc/ @ryancdotorg