Introduction to Programmable Cryptography

Rizwan Kazi ‘20

Atomhacks 2025

Kadena

Introduction

I can prove to you that I have a message M such that $\mathrm{sha}(M) = 0xa91af3ac\ldots$, *without revealing M. But not just for the hash function sha. I can do this for any function you want.*

OxPARC, December 4th, 2024.

Cryptography is the discipline of writing a message in *ciphertext*, usually by translating *plaintext* according to some (frequently changing) *keytext*, with the aim of protecting a secret from adversaries, interceptors, intruders, interlopers, eavesdroppers, opponents, or simply attackers, opponents, and enemies. Professional cryptography protects not only the plaintext, but also the key and, more generally, tries to protect the whole *cryptosystem*.

Cryptocurrency

Crypto*graphy* is what makes **crypto***currency* possible.

Cryptocurrency is digital currency designed to work through a computer network that is not reliant on any central authority, such as a government or bank, to uphold or maintain it. **Bitcoin**, the first cryptocurrency, runs on the Bitcoin **blockchain**.

Blockchains 1

Blockchains are distributed ledgers with growing lists of records (*blocks*) that are securely linked together using **cryptographic hash functions**.

[Chaum1982] was the first to propose a blockchain protocol. [Haber&Stornetta1991] was the first to implement a blockchain protocol, hashing document certificates and publishing them in the New York Times every week since 1995.

The infamous [Nakamoto2008] was the first decentralized blockchain. Now, there are more than a thousand different blockchains. I work at one! And Stuart Haber is actually one of our advisors!

Several functions go into the operation of a blockchain; blockchain *transactions* are sets of of function calls to *contracts*, which are collections of function calls to other contracts or standardized primitives.

Bitcoin Genesis Block

```
00000000   01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
00000010   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
00000020   00 00 00 00 3B A3 ED FD  7A 7B 12 B2 7A C7 2C 3E   ....;...z{..z..,>
00000030   67 76 8F 61 7F C8 1B C3  88 8A 51 32 3A 9F B8 AA   gv.a.... ..Q2:...
00000040   4B 1E 5E 4A 29 AB 5F 49  FF FF 00 1D 1D AC 2B 7C   K.^J)._I......+|
00000050   01 01 00 00 00 01 00 00  00 00 00 00 00 00 00 00   ................
00000060   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
00000070   00 00 00 00 00 00 FF FF  FF FF 4D 04 FF FF 00 1D   ...... ....M.....
00000080   01 04 45 54 68 65 20 54  69 6D 65 73 20 30 33 2F   ..EThe Times 03/
00000090   4A 61 6E 2F 32 30 30 39  20 43 68 61 6E 63 65 6C   Jan/2009 Chancel
000000A0   6C 6F 72 20 6F 6E 20 62  72 69 6E 6B 20 6F 66 20   lor on brink of
000000B0   73 65 63 6F 6E 64 20 62  61 69 6C 6F 75 74 20 66   second bailout f
000000C0   6F 72 20 62 61 6E 6B 73  FF FF FF FF 01 00 F2 05   or banks........
000000D0   2A 01 00 00 00 43 41 04  67 8A FD B0 FE 55 48 27   *....CA.g...UH'
000000E0   19 67 F1 A6 71 30 B7 10  5C D6 A8 28 E0 39 09 A6   .g..q0..\..(.9..
000000F0   79 62 E0 EA 1F 61 DE B6  49 F6 BC 3F 4C EF 38 C4   yb...a..I..?L.8.
00000100   F3 55 04 E5 1E C1 12 DE  5C 38 4D F7 BA 0B 8D 57   .U......\8M....W
00000110   8A 4C 70 2B 6B F1 1D 5F  AC 00 00 00 00            .Lp+k.._.....
```

Cryptographic hash functions

This raw hex file is stored, obfuscated, verified, and used as the immutable starting point of Bitcoin. The header is double-hashed to produce a unique identifier; miners then solve SHA-256 puzzles to validate blocks. SHA-256 is essential to secure the Merkle tree that is Bitcoin.

At Kadena, we use BLAKE2s. Ethereum uses Keccak-256.

SHA-256

Introduction

- All variables are 32-bit unsigned integers and addition is modulo 2^{32}
- Each round uses a constant $k[i]$ and message schedule entry $w[i]$
- The compression function uses 8 working variables, a through h
- Big-endian is used for constants and message block data parsing

Initialization

Algorithm 1 Hash values: 32 bits from square roots of (first 8) primes

$\mathtt{h}_0 \leftarrow \mathtt{0x6a09e667}$

$\mathtt{h}_1 \leftarrow \mathtt{0xbb67ae85}$

$\mathtt{h}_2 \leftarrow \mathtt{0x3c6ef372}$

$\mathtt{h}_3 \leftarrow \mathtt{0xa54ff53a}$

$\mathtt{h}_4 \leftarrow \mathtt{0x510e527f}$

$\mathtt{h}_5 \leftarrow \mathtt{0x9b05688c}$

$\mathtt{h}_6 \leftarrow \mathtt{0x1f83d9ab}$

$\mathtt{h}_7 \leftarrow \mathtt{0x5be0cd19}$

Algorithm 2 Round constants from cube roots of (first 64) primes

$k[0, \ldots, 63] \leftarrow 0x428a2f98, \ldots, 0xc67178f2$

Initialization | Python Implementation

```python
h0 = 0x6a09e667
h1 = 0xbb67ae85
h2 = 0x3c6ef372
h3 = 0xa54ff53a
h4 = 0x510e527f
h5 = 0x9b05688c
h6 = 0x1f83d9ab
h7 = 0x5be0cd19

k = [
    0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
    0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
    0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
    0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
    0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
    0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
    0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
    0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2
]
```

Padding

Algorithm 3 Pre-processing: Padding

Start with message of length L bits
Append a single `'1'` bit
Append K `'0'` bits such that $L + 1 + K + 64 \equiv 0 \pmod{512}$
Append L as a 64-bit big-endian integer

```python
def pad(message: bytes) -> bytes:
    L = len(message) * 8
    K = (-(L + 1 + 64)) % 512
    padding = b'\x80' + (b'\x00' * ((K + 1)
  // 8 - 1))
    padded_message = message + padding + L.
  to_bytes(8, 'big')
    return padded_message
```

Algorithm 4 Message schedule array $w[0, \ldots, 63]$

```
for each 512-bit chunk do
    Initialize w[0,...,63] to all zeros
    Copy chunk into w[0,...,15]
    for i = 16 to 63 do
        s0 ← (w[i − 15] ⋙ 7) ⊕ (w[i − 15] ⋙ 18) ⊕ (w[i − 15] ≫ 3)
        s1 ← (w[i − 2] ⋙ 17) ⊕ (w[i − 2] ⋙ 19) ⊕ (w[i − 2] ≫ 10)
        w[i] ← w[i − 16] + s0 + w[i − 7] + s1
```

Processing 2

Algorithm 5 Compression function: main loop calculations

$\mathtt{a} \leftarrow \mathtt{h}_0$, $\mathtt{b} \leftarrow \mathtt{h}_1$, $\mathtt{c} \leftarrow \mathtt{h}_2$, $\mathtt{d} \leftarrow \mathtt{h}_3$
$\mathtt{e} \leftarrow \mathtt{h}_4$, $\mathtt{f} \leftarrow \mathtt{h}_5$, $\mathtt{g} \leftarrow \mathtt{h}_6$, $\mathtt{h} \leftarrow \mathtt{h}_7$
for $i = 0$ to 63 **do**
    $S_1 \leftarrow (\mathtt{e} \ggg 6) \oplus (\mathtt{e} \ggg 11) \oplus (\mathtt{e} \ggg 25)$
    $ch \leftarrow (\mathtt{e} \wedge \mathtt{f}) \oplus (\neg \mathtt{e} \wedge \mathtt{g})$
    $temp1 \leftarrow \mathtt{h} + S_1 + ch + \mathtt{k}[i] + \mathtt{w}[i]$
    $S_0 \leftarrow (\mathtt{a} \ggg 2) \oplus (\mathtt{a} \ggg 13) \oplus (\mathtt{a} \ggg 22)$
    $maj \leftarrow (\mathtt{a} \wedge \mathtt{b}) \oplus (\mathtt{a} \wedge \mathtt{c}) \oplus (\mathtt{b} \wedge \mathtt{c})$
    $temp2 \leftarrow S_0 + maj$

Processing 3

Algorithm 6 Compression function: state updates

```
for i = 0 to 63 do                ▷ continued
    h ← g
    g ← f
    f ← e
    e ← d + temp1
    d ← c
    c ← b
    b ← a
    a ← temp1 + temp2
```

Processing 4

Algorithm 7 Compression function: hash updates

Update hash values:
$h_0 \leftarrow h_0 + a$
$h_1 \leftarrow h_1 + b$
$h_2 \leftarrow h_2 + c$
$h_3 \leftarrow h_3 + d$
$h_4 \leftarrow h_4 + e$
$h_5 \leftarrow h_5 + f$
$h_6 \leftarrow h_6 + g$
$h_7 \leftarrow h_7 + h = 0$

Algorithm 8 Final SHA-256 hash

1: Concatenate $\mathtt{h}_0||\mathtt{h}_1||\mathtt{h}_2||\mathtt{h}_3||\mathtt{h}_4||\mathtt{h}_5||\mathtt{h}_6||\mathtt{h}_7$
2: Output the 256-bit final hash value (digest)

Processing and Results | Python Implementation

```python
def rightrotate(value: int, bits: int) ->
    int:
    return ((value >> bits) | (value << (32
    - bits))) & 0xFFFFFFFF
```

Processing and Results | Python Implementation

```
def sha256(message: bytes) -> bytes:
    hash_values = [h0, h1, h2, h3, h4, h5, h6, h7]
    padded = pad(message)
    for chunk in [padded[i:i+64] for i in range(0, len(padded), 64)]:
        w = [0] * 64
        for i in range(16):
            w[i] = int.from_bytes(chunk[i*4:(i+1)*4], 'big')
        for i in range(16, 64):
            s0 = rightrotate(w[i-15], 7) ^ rightrotate(w[i-15], 18) ^ (w[i-15] >> 3)
            s1 = rightrotate(w[i-2], 17) ^ rightrotate(w[i-2], 19) ^ (w[i-2] >> 10)
            w[i] = (w[i-16] + s0 + w[i-7] + s1) & 0xFFFFFFFF
        a, b, c, d, e, f, g, h = hash_values
        for i in range(64):
            S1 = rightrotate(e, 6) ^ rightrotate(e, 11) ^ rightrotate(e, 25)
            ch = (e & f) ^ ((~e) & g)
            temp1 = (h + S1 + ch + k[i] + w[i]) & 0xFFFFFFFF
            S0 = rightrotate(a, 2) ^ rightrotate(a, 13) ^ rightrotate(a, 22)
            maj = (a & b) ^ (a & c) ^ (b & c)
            temp2 = (S0 + maj) & 0xFFFFFFFF
            h = g
            g = f
            f = e
            e = (d + temp1) & 0xFFFFFFFF
            d = c
            c = b
            b = a
            a = (temp1 + temp2) & 0xFFFFFFFF
```

Processing and Results | Python Implementation

```python
        hash_values = [
            (hash_values[0] + a) & 0xFFFFFFFF,
            (hash_values[1] + b) & 0xFFFFFFFF,
            (hash_values[2] + c) & 0xFFFFFFFF,
            (hash_values[3] + d) & 0xFFFFFFFF,
            (hash_values[4] + e) & 0xFFFFFFFF,
            (hash_values[5] + f) & 0xFFFFFFFF,
            (hash_values[6] + g) & 0xFFFFFFFF,
            (hash_values[7] + h) & 0xFFFFFFFF
        ]
    return b''.join(h.to_bytes(4, 'big') for h in hash_values)
```

Keccak-256

Introduction

- Uses sponge construction (absorb-squeeze) instead of Merkle-Damgård
- State: 5x5x64-bit 3D array (1600 bits total)
- Parameters: Bitrate r = 1088, Capacity c = 512, Rounds=24
- Big-endian convention for input/output

Initialization

Algorithm 9 Round constants from cube roots of (first 64) primes

1: Initialize 5x5 state matrix to zeros

2: $RC[0,\ldots,23] \leftarrow$ $\mathtt{0x0000000000000001}, \ldots, \mathtt{0x8000000000008008}$

Initialization | Python Implementation

```python
state = [[0x0 for _ in range(5)] for _ in range(5)]

k = [
    0x0000000000000001, 0x0000000000008082, 0x800000000000808A,
    0x8000000080008000, 0x000000000000808B, 0x0000000080000001,
    0x8000000080008081, 0x8000000000008009, 0x000000000000008A,
    0x0000000000000088, 0x0000000080008009, 0x000000008000000A,
    0x000000008000808B, 0x800000000000008B, 0x8000000000008089,
    0x8000000000008003, 0x8000000000008002, 0x8000000000000080,
    0x000000000000800A, 0x800000008000000A, 0x8000000080008081,
    0x8000000000008080, 0x0000000080000001, 0x8000000080008008
]

def rot(value: int, n: int) -> int:
    return ((value << (64 - n)) | (value >> n)) & 0xFFFFFFFFFFFFFFFF

```

Padding

Algorithm 10 Multi-rate padding

1: Append `0x01` byte
2: Pad with zeros until length $\equiv r - 8 \pmod{r}$
3: Append final `0x80` byte

Padding | Python Implementation

```python
def pad(message: bytes) -> bytes:
    padded = message + b'\x01'
    while (len(padded)*8) % 1088 != 0:
        padded += b'\x00'
    return padded[:-1] + b'\x80'

```

Processing 1

Algorithm 11 Absorption phase

```
for each 1088-bit block do
    XOR block into state[0..16] (first 17 lanes)
    Apply Keccak-f permutation
```

Processing 2

Algorithm 12 Keccak-f[1600] permutation steps

```
for 24 rounds do
    θ: Mix columns with parity
    ρ: Rotate lanes
    π: Permute lanes
    χ: Non-linear mixing
    ι: XOR round constant
```

Theta | Python Implementation

```python
def theta(state):
    C = [0]*5
    D = [0]*5
    for x in range(5):
        C[x] = state[x][0] ^ state[x][1] ^ state[x][2]
 ^ state[x][3] ^ state[x][4]
    for x in range(5):
        D[x] = C[(x-1)%5] ^ rot(C[(x+1)%5], 1)
    for x in range(5):
        for y in range(5):
            state[x][y] ^= D[x]
    return state

```

Rho and Pi | Python Implementation

```python
def rho_pi(state):
    new_state = [[0]*5 for _ in range(5)]
    offsets = [
        [ 0, 36,  3, 41, 18],
        [ 1, 44, 10, 45,  2],
        [62,  6, 43, 15, 61],
        [28, 55, 25, 21, 56],
        [27, 20, 39,  8, 14]
    ]
    for x in range(5):
        for y in range(5):
            new_state[y][(2*x + 3*y) % 5] = rot(state[x][y], offsets[x][y])
    return new_state
```

Chi | Python Implementation

```python
def chi(state):
    new_state = [[0]*5 for _ in range(5)]
    for x in range(5):
        for y in range(5):
            new_state[x][y] = state[x][y] ^ (
                (~state[(x+1)%5][y]) & state[(x+2)%5][y])
    return new_state
```

Iota | Python Implementation

```python
def iota(state, round_num):
    state[0][0] ^= RC[round_num]
    return state
```

Processing 3

Algorithm 13 Squeezing phase (fixed to 256 bits for Keccak-256)

1: Extract first 256 bits from state
2: **if** more output needed **then**
3: Apply Keccak-f permutation
4: Extract next 256 bits

Keccak-f Permutation | Python Implementation

```python
def keccak_f(state):
    for round_num in range(24):
        state = theta(state)
        state = rho_pi(state)
        state = chi(state)
        state = iota(state, round_num)
    return state
```

Processing and Results | Python Implementation

```python
def keccak256(message: bytes) -> bytes:
    state = [[0]*5 for _ in range(5)]
    padded = pad(message)
    for i in range(0, len(padded), 136):
        block = padded[i:i+136] + bytes(136 - len(padded[i:i+136]))
        for x in range(5):
            for y in range(5):
                if 5*x + y < 17:
                    state[x][y] ^= int.from_bytes(
                        block[(5*x + y)*8:(5*x + y +1)*8], 'little')
        state = keccak_f(state)
    output = b''
```

Processing and Results | Python Implementation

```
    for x in range(5):
        for y in range(5):
            if 5*x + y < 4:
                output += state[x][y].to_bytes(8, 'little')
    return output[:32]
```

BLAKE2s

Introduction

- Part of BLAKE2 family, optimized for 8-32 bit platforms
- Produces digests from 1 to 32 bytes
- Uses modified ChaCha stream cipher with 10 rounds
- Supports keyed hashing, salt, and personalization
- Big-endian convention for parameters, little-endian for data

Initialization

Algorithm 14 Initialization with parameter block

1: $\texttt{IV}_0, \ldots, \texttt{IV}_7 \leftarrow$ fractional sqrts of primes
$IV = [0x6A09E667, \ldots, 0x5BE0CD19]$

2: $\texttt{h}_0, \ldots, \texttt{h}_7 \leftarrow \texttt{IV} \oplus (\texttt{param\_block})$

3: $\texttt{c} \leftarrow 0$ (byte counter)

Initialization | Python Implementation

```
IV = [
    0x6A09E667, 0xBB67AE85, 0x3C6EF372, 0xA54FF53A,
    0x510E527F, 0x9B05688C, 0x1F83D9AB, 0x5BE0CD19
]
def blake2s_init(digest_size=32, key=b''):
    h = IV.copy()
    param_block = (
        (digest_size | (len(key) << 8) | (0x01 << 16))
        | (0x01 << 24)
    )
    h[0] ^= param_block
    return {
        'h': h,
        't': [0, 0],
        'buffer': bytearray(64),
        'digest_size': digest_size,
        'key': key.ljust(32, b'\x00')[:32]
    }

```

Padding

Algorithm 15 BLAKE2s padding scheme

1: Pad message with zeros to multiple of 64 bytes
2: Last block contains original message length

Padding | Python Implementation

```python
def blake2s_pad(message: bytes):
    padded = bytearray(message)
    padded += b'\x00' * (-len(message) % 64)
    padded[-8:] = (len(message) * 8).to_bytes(8, '
   little')
    return padded

```

Processing 1

Algorithm 16 Message processing loop

1: **for** each 64-byte block **do**
2: Update byte counter $\texttt{t} \leftarrow \texttt{t} + 64$
3: Compress block with current state

Processing 2

Algorithm 17 Compression function

function $\textsc{Compress}(h, block, t, last)$

  $v_0, \ldots, v_{15} \leftarrow h \parallel IV \parallel t \parallel \texttt{constants}$

  for 10 rounds **do**

    Apply G mixers to columns/diagonals

  $h \leftarrow h \oplus v_0, \ldots, v_7 \oplus v_8, \ldots, v_{15}$

G Mixer | Python Implementation

```python
def G(a: int, b: int, c: int, d: int, mx: int, my: int
    ):
    a = (a + b + mx) & 0xFFFFFFFF
    d = ((d ^ a) << 16 | (d ^ a) >> 16) & 0xFFFFFFFF
    c = (c + d) & 0xFFFFFFFF
    b = ((b ^ c) << 12 | (b ^ c) >> 20) & 0xFFFFFFFF
    a = (a + b + my) & 0xFFFFFFFF
    d = ((d ^ a) << 8 | (d ^ a) >> 24) & 0xFFFFFFFF
    c = (c + d) & 0xFFFFFFFF
    b = ((b ^ c) << 7 | (b ^ c) >> 25) & 0xFFFFFFFF
    return a, b, c, d

```

```python
def compress(ctx, block, last):
    m = [int.from_bytes(block[i*4:(i+1)*4], 'little')
    for i in range(16)]
    v = ctx['h'] + IV + [
        ctx['t'][0] ^ 0xFFFFFFFF, ctx['t'][1],
        0xFFFFFFFF if last else 0, 0
    ]

```

Compression | Python Implementation

```python
    for round in range(10):
        v[0], v[4], v[8], v[12] = G(v[0], v[4], v[8], v[12], m[0], m[1])
        v[1], v[5], v[9], v[13] = G(v[1], v[5], v[9], v[13], m[2], m[3])
        v[2], v[6], v[10], v[14] = G(v[2], v[6], v[10], v[14], m[4], m[5])
        v[3], v[7], v[11], v[15] = G(v[3], v[7], v[11], v[15], m[6], m[7])

        v[0], v[5], v[10], v[15] = G(v[0], v[5], v[10], v[15], m[8], m[9])
        v[1], v[6], v[11], v[12] = G(v[1], v[6], v[11], v[12], m[10], m[11])
        v[2], v[7], v[8], v[13] = G(v[2], v[7], v[8], v[13], m[12], m[13])
        v[3], v[4], v[9], v[14] = G(v[3], v[4], v[9], v[14], m[14], m[15])
```

Compression | Python Implementation

```python
    for i in range(8):
        ctx['h'][i] ^= v[i] ^ v[i+8]

```

Results | Python Implementation

```python
def blake2s(message: bytes, digest_size=32, key=b'') -> bytes:
    ctx = blake2s_init(digest_size, key)
    padded = blake2s_pad(message)

    for i in range(0, len(padded), 64):
        ctx['t'][0] += 64
        if ctx['t'][0] < 64:
            ctx['t'][1] += 1
        compress(ctx, padded[i:i+64], i == len(padded)-64)

    return b''.join(h.to_bytes(4, 'little') for h in ctx['h'][:digest_size//4])

```