ASSIGMENT 3

- KECCAK SHA-3-256

KECCAK SHA-3 VERSION 256
OVERVIEW & IMPLEMENTATION GUIDE

STUDENT: LAURA FORTUNE

GITHUB REPOSITORY: https://github.com/laura-D21126682/sha-3-256

TABLE OF CONTENTS

INTRODUCTION

Cryptography is the practice of securing electronic data in transit or at rest, achieved through methods such as encryption or hashing. In simple terms, encryption involves taking an input(plaintext), obscuring it, and returning an output(ciphertext). The intended(authorised) parties can later decrypt the encrypted output using a 'key'.

HASHING ALGORITHMS

Hashing algorithms differ in that they are one way functions; once an input has been processed and returned that output can never be reversed. Cryptographic hash algorithms are functions that take a piece of data as an input(message), condense and transform it, producing a fixed length output known as the hash digest. The message can be a string value of any length, while the hash digest is typically a byte value much smaller than the input message. The hash digest should be unique to the input message; adjusting just one bit of the message should result in a completely different hash value, this is known as the 'Avalanche effect'.

The National Institute of Standards and Technology (NIST) specifies the following criteria for a function to qualify as a cryptographically secure hash function:

1. Collision resistance
   A collision event occurs when two different messages produce the exact same hash output. Collision resistance refers to the computational difficulty in finding one. If a collision is found or theoretically proven to be feasible, the security of the hash algorithm is significantly undermined, and no longer suitable for cryptographic use.
   Collision resistance strength is measured in bits and calculated as half the length of the hashed output ($L/2\ bits$).
2. Preimage resistance
   Hashing is a one way function, once a message(preimage) has been processed and hashed, it should be computationally prohibitive to reverse the process and retrieve the original message. Preimage resistance directly correlates to the length of the hash digest. For example: a hashed output of 256-bits has a preimage resistance of 256-bits and a successful brute force attack would take up to 2^{256} - an incredibly large number.
3. Second preimage resistance
   The computational infeasibility of finding a different input that produces the same hash value as a 'known' message's corresponding hash digest. For example: an attacker starts with a known message and its hash and their goal is to find another input that will produce the exact same hash value. This resistance value is calculated in the same way as the preimage resistance: 2^n, where 'n' is the bit length of the hash output.

APPLICATION

Secure hashing algorithms can be used to reliably verify the authenticity of data, where any changes malicious or otherwise are instantly detected.
Common applications include digital signatures, Message Authentication Codes (MAC), cryptographic random number generation or simply file verification and indexing (e.g. hash tables).

Most people unknowingly use hash functions every day; on creation passwords are stored in their hashed format rather than plaintext. This means at login, the entered password is hashed before being compared to the stored value. If this database were ever to be compromised and the passwords had been hashed using a valid hashing solution, it would be extremely difficult for the attackers to retrieve the original passwords.

A lesser known, but interesting use of hashing algorithms is in blockchain technology. When a new block is created a cryptographic hash is generated from its entire contents, including the hash of the previous block. Before it gets added to the chain this hash must be validated by other nodes on the network. If a block were to be altered at any stage, it would result in a completely new hash, alerting the other nodes and invalidating all subsequent blocks due to the break in the chain of hashes.

THE SEARCH FOR SHA-3

In 2007, the NIST announced an open competition to find the next SHA standard: SHA-3. While SHA-2 was (and still is) considered cryptographically secure, the search for SHA-3 was a pre-emptive step in response to the successful attacks and collisions found in earlier hash functions: MD4, MD-5, SHA-1. The competition received approximately 64 submissions, and on Oct 2, 2012 Keccak was selected as the successor to SHA-2.

KECCAK

The Keccak algorithm was designed by Guido Bertoni, Joan Daemen(co-author of AES), Michaël Peeters and Gilles Van Assche, and is based on a broader framework known as ‘sponge construction’, formally defined in their 2007 paper ‘Sponge Functions’. Sponge construction was a completely new approach; previous hashing standards were built on Merkle-Damberg compression functions. In Merkle-Damberg construction, input data is processed in fixed size blocks, updating the state with each new block. The final hash is extracted directly from the state once the last block has been applied, making it particularly vulnerable to length extension attacks. In contrast sponge construction performs multiple iterations of permutations on the internal state before returning the hash digest. Keccak has cryptographic applications beyond

hashing, such as MAC functions and stream ciphers. This document, however, will focus on Keccak as defined in the 2011 'Keccak SHA-3 submission', and later standardised by the NIST.

KECCAK SHA-3

One of the reasons Keccak was chosen for SHA-3 was because of its significant difference from its predecessors, due to its innovative design and use of sponge functions.

HIGH-LEVEL

- Sponge construction:
  Defines the overarching architecture of Keccak SHA-3 and operates in two main phases. Absorbing: the input is absorbed into the internal state and processed, and Squeezing, the output is squeezed out to produce the final hash result.

- SHA-3 Parameters:
  - A State Array Matrix ($5 * 5 * w$), where 'w' is equal to 64-bits. The state array consists of 25 lanes (or words), each 64-bits in size.
  - b Width of $Keccak - p$ permutation: 1600-bits total size of the internal state array.
  - c Capacity = ($d * 2$), where 'd' is equal to output length. The capacity determines the security level of the sponge function.
  - r Rate = ($b - c$), determines how much data is processed in each block – the block size.
  - d Output length = A fixed length determined by the chosen SHA-3 variant.
  - w Lane size = 64-bits long, where $w = 2^L$ and $l = 6$
  - M Message = The input string, can be of any length.
  - Nr Number of rounds $(12 + 2^6)$ = 24 permutation rounds are performed.

The specific requirements for the SHA-3 standard are outlined in table 1.0 (*SHA-3-256 highlighted in red)

Table 1.0: Keccak SHA-3 variants

Hash Function	Output d	State b	Rate r (block size)	Capacity c	Collision resistance $d/2$	Preimage Resistance	Second Preimage Resistance
SHA-3-224	224-bit	1600-bit	1152-bit	448-bit	112-bit	224-bit	224-bit
SHA-3-256	256-bit	1600-bit	1088-bit	512-bit	128-bit	256-bit	256-bit
SHA-3-384	384-bit	1600-bit	832-bit	768-bit	192-bit	384-bit	384-bit
SHA-3-512	512-bit	1600-bit	576-bit	1024-bit	256-bit	512-bit	512-bit

SHA-3-256

SHA-3-256 produces a 256-bit digest and operates on a 1600-bit internal state array.

The internal state is made of two parts:

- The 512-bit Capacity which is twice the length of the output (256 * 2), and provides robust protection against attacks.
    - Collision resistance: 112-bits
    - Preimage and second preimage resistance: 256 bits, requiring a brute force attack of up to 2^{256} attempts.
- The 1088-bit Rate is calculated by subtracting the capacity from the total state size (1600-bits), the remaining value gives us the block size. The block size (or bitrate) is the amount of data that can be processed during each permutation iteration.

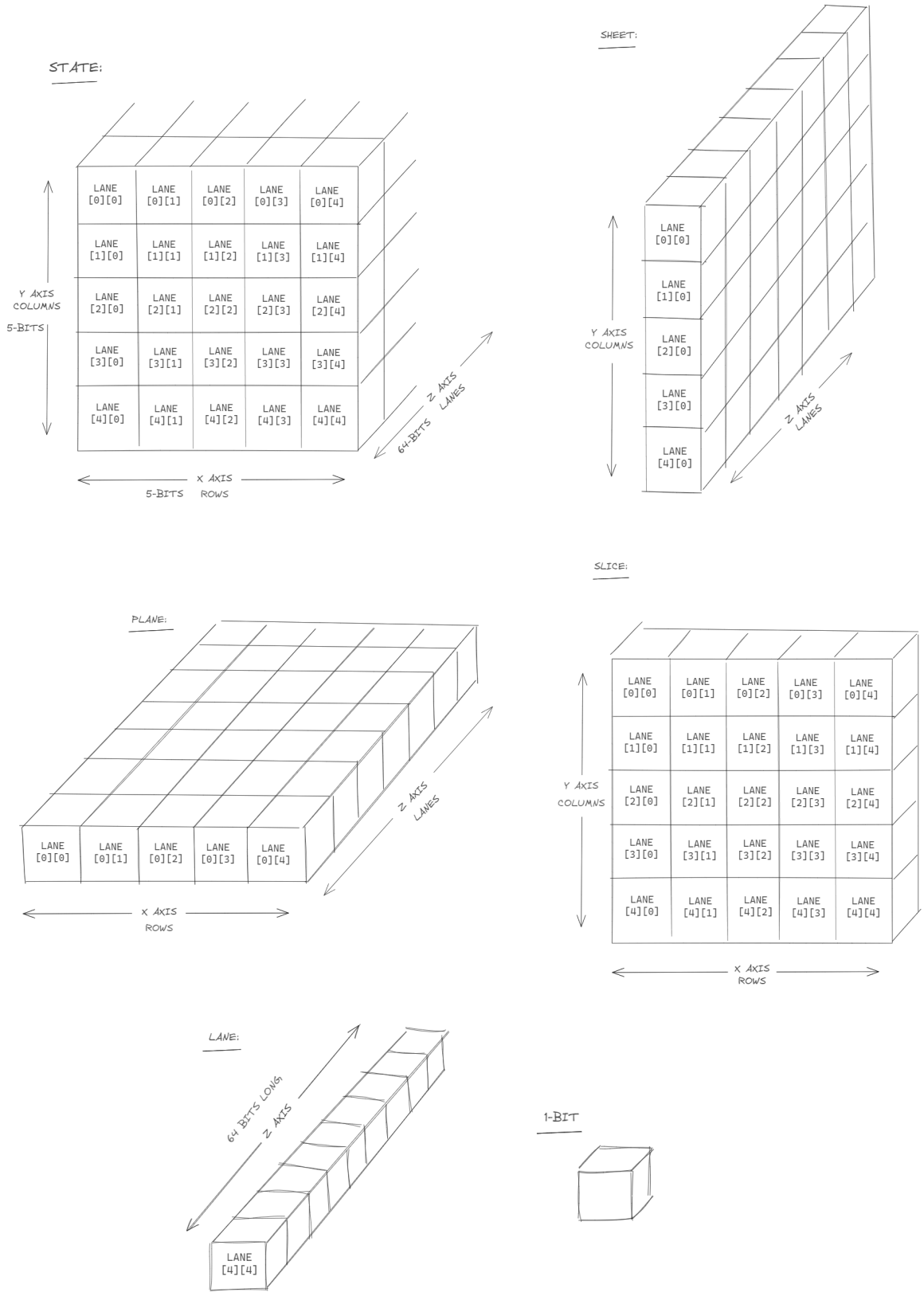
The capacity and rate parameters work in tandem. A higher capacity value correlates with greater resistance to security threats, but it also results in a lower bitrate. This means smaller block sizes are processed on each round, slowing the hashing process. SHA-3-256 is balanced in its approach – the capacity provides strong protection without compromising the efficiency of the hashing performance.

STATE MATRIX

The state matrix is central to Keccak's sponge construction, storing and tracking the algorithm's internal state, as it undergoes multiple transformations. Keccak SHA-3 operates on a state matrix of 1600-bits.

- 25 Lanes of 64-bits (5 rows * 5 rows * 64 Lanes)

Fig 01: State Matrix



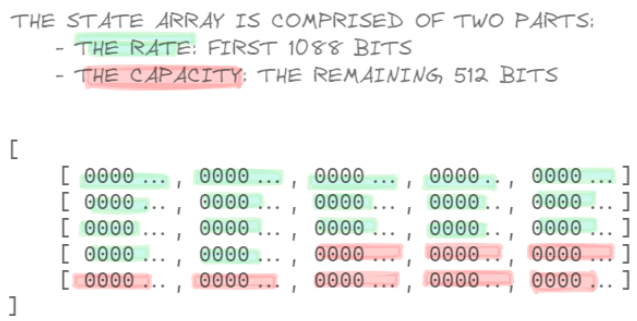
SPONGE CONSTRUCTION IMPLEMENTATION

The following section provides a more detailed description of Keccak SHA-3 Sponge construction, using a message size of 1500-bits as an example input.

INITIALISE STATE

The 1600-bit matrix can be represented as a 2d array and is initialised to all zeros.

Fig 02: State initialisation

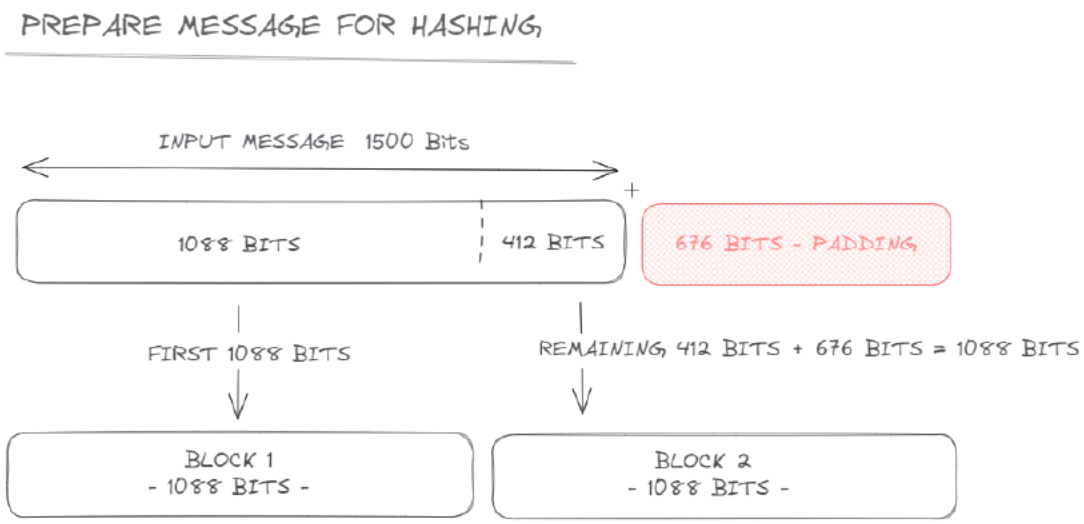



```
// Initialisation
void initialise_state(uint64_t state[5][5]) {
  for(int x = 0; x < 5; x++) {
    for(int y = 0; y < 5; y++) {
      state[x][y] = 0;
    }
  }
}
```

PADDING

The message to be hashed is split into 1088-bit blocks. If the remaining message block is smaller than the required 1088-bits, zeros are used to pad the block to the required bitrate.

Fig 03: Padding



Keccak-256 Padding Rule

The block size for Keccak-256 is 136 bytes $(r\ 1088 / 8 - bits = 136\ bytes)$

```
BLOCKSIZE = 136; // rate: 1088 / 8-bits = 136-bytes
```

Padding length is found by calculating the total number of bytes required to make message length a multiple of block size. This can be achieved using the modulus operator:

```
padding_length = BLOCKSIZE - (message_length % BLOCKSIZE);
```

- If only one byte of padding is needed, byte $0x81$ (10000001) is used.
- Else the padding begins with $0x01$ (00000001), is filled with zeros $0x00$, and ends with a $0x80$ (10000000) - signifying the end of the sequence.

C implementation of Keccak-256 padding rule:

```
// Padding (keccak-256)
void padding(unsigned char *buffer, size_t length, size_t padding_length) {
  if (padding_length == 1) {
    buffer[length] = 0x81;  // 0x81 if only one byte of padding needed
  } else {
    buffer[length] = 0x01; // Padding starts at 0x01
    memset(buffer + length + 1, 0x00, padding_length - 2); // zero filled
    buffer[length + padding_length - 1] = 0x80; // End padding with 0x80
  }
}
```

The NIST SHA-3 implementation stays true to the original Keccak submission, with exception to a minor adjustment in how the padding rule is applied.

- If only one byte of padding is required, byte value $0x86$ (10000110) is used.
- Else padding begins with $0x06$ (00000110), is filled with zeros $0x00$, and ends with a $0x80$ (10000000) - signifying the end of the sequence

ABSORBING PHASE

During the absorption phase the rate is used to incorporate(absorb) the input message into the internal state. Each bit of the message is XORed with a corresponding rate bit.
The capacity is used to ensure hash security and is excluded from the XOR process. The entire state array (rate and capacity) is transformed by the permutation functions.

Fig 03: Absorbing

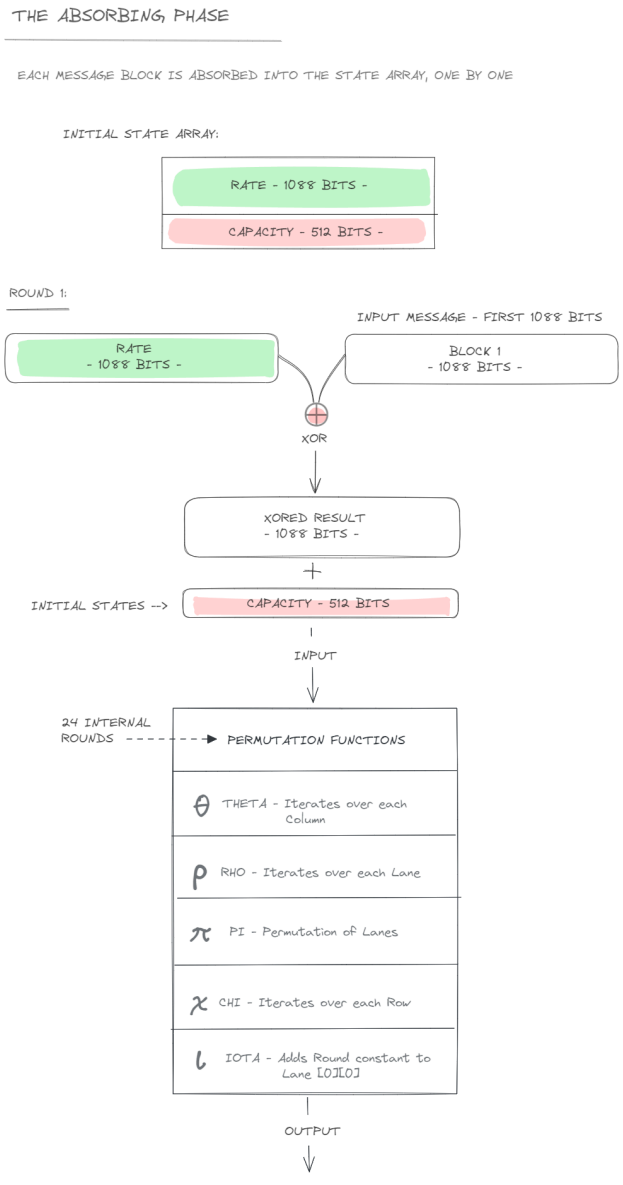
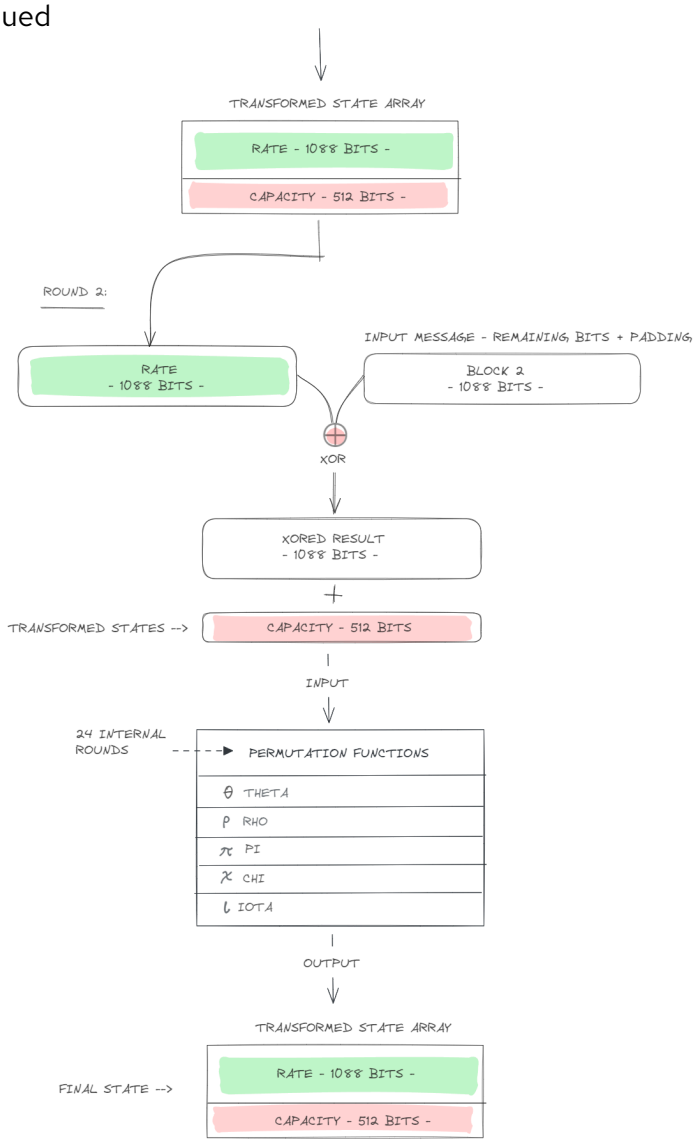


Fig 04: Absorbing Continued



SQUEEZING PHASE

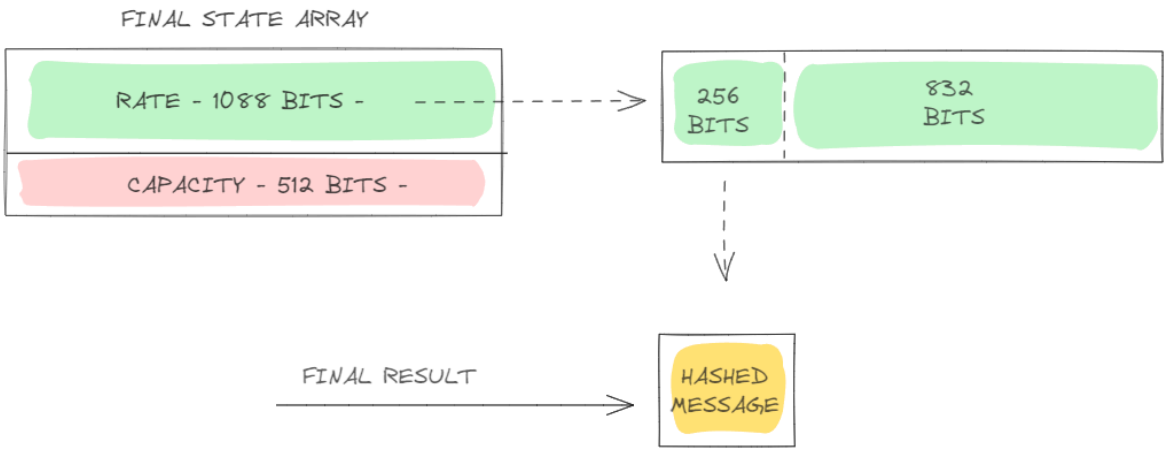
During the squeezing phase the final hash output is extracted (squeezed) from the transformed state array – which undergoes 24 rounds of the permutation functions for each absorbed message block.

If the required hash digest is greater than the rate size in bits:

- The full rate portion of the state is extracted,
- The permutation functions are applied to the state array, and bits are extracted again.
- This process continues until the chosen hash length has been achieved.

However, for SHA-3-256, only 256-bits are extracted. This means the permutation functions do not need to be called again during the squeezing phase. The hash result is equal to the first 256-bits of the final state's rate.

Fig 05: Squeezing



KECCAK-F[1600] PERMUTATIONS

THETA STEP

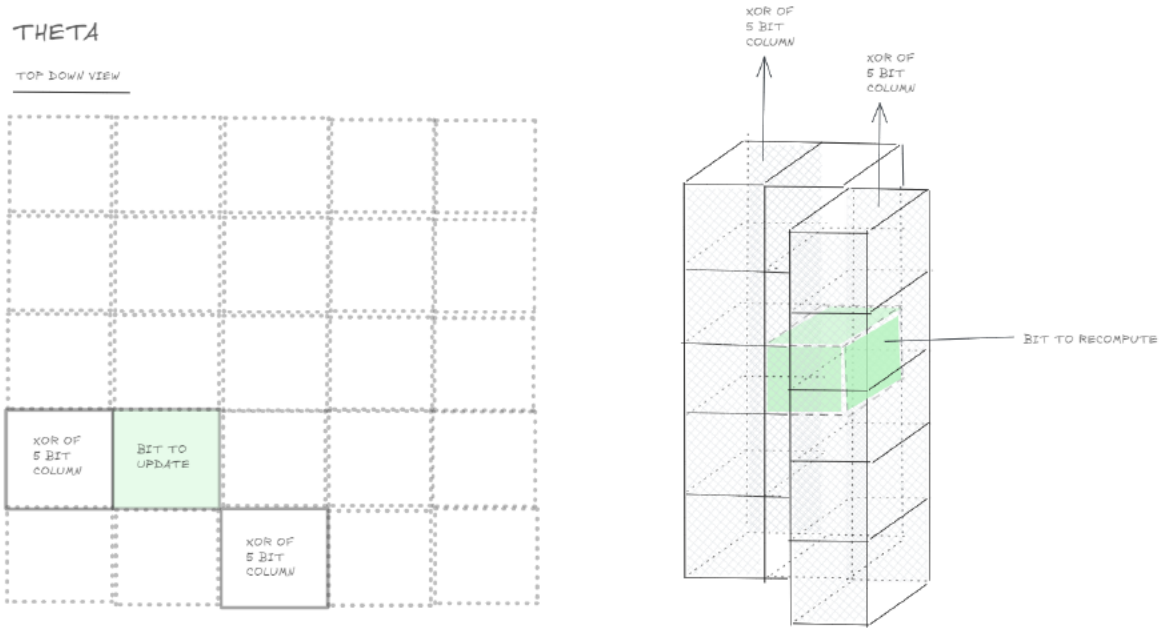
In Theta, the internal state is recomputed/updated. Each of the 1600 state bits are replaced by the XOR sum of the following 11 bits:

1. The original bit
2. The XOR of the 5-bit column "to the left" of the original bit
3. The XOR of the 5-bit column "to the right", and "1 position forward" on the z-axis, from the original bit.

Theta ensures the state is consistently diffused and mixed up across the entire state -with every bit being updated and influenced by its neighbouring column's bits.

Modulus 5 is used for wrapping on the X axis , while modulus 64 is used for wrapping on the Z axis.

Fig 06: Theta



C implementation of Theta:

```
const theta = (state) => {
  let C = new Array(5).fill(0n); // store parity of each column
  let D = new Array(5).fill(0n); // store diffused column parity
```

C[x] - Calculates the column parity (XOR sum) for each column x

```
  // Calculate C
  for (let x = 0; x < 5; x++) {
    for (let y = 0; y < 5; y++) { // C[x] = A[x][0] // Ensure C[x] is reset
      C[x] ^= A[x][y];            // XOR of each sheet
    }
  }
```

D[x]- Calculates the diffused column parity for each column x

- Column parity of the column to the left C[(x + 4) % 5], is XORed with column parity of the column to the right and rotated forward 1 position ROTL64(C[(x + 1) % 5], 1)

```
  // Calculate D
  for (let x = 0; x < 5; x++) {
    D[x] = C[(x + 4) % 5] ^ ROTL64(C[(x + 1) % 5], 1);
  }
```

State A is updated with its new values, each bit of column x is XORed with its corresponding D value

```
  // Put result into A
  for (let x = 0; x < 5; x++) {
    for (let y = 0; y < 5; y++) {
      A[x][y] ^= D[x];
    }
  }
};
```

RHO AND PI STEP

Operates on the 64-bit lanes of the state matrix. State is composed of 25 lanes of 64-bits, 1600-bits total. For simplicity Rho and Pi are combined within the same 24 round loop.

1. Rho:
   First each lane (word) is rotated by a fixed number of bit positions.
2. Pi:
   Then rotated lanes are then permuted, ensuring further shuffling and diffusion of bits.

C implementation of Rho and Pi:

- Initialisation x,y coordinates, and set initial lane position

```
void rho_and_pi(uint64_t state[5][5]) {
  // Initialisation
  int x = 1, y = 0;                          // Initial coordinates
  uint_fast64_t lane = state[x][y];         // value stored at lane(x, y)
```

- Rho and Pi operate inside this 24 round loop

```
for (int t = 0; t < 24; t++) {
```

- Rho: Calculate lane rotation offset
- modulus 64 means bits are wrapped on the z-axis.

```
    int rotationOffset = ((t + 1) * (t + 2) / 2) % 64;
```

- Pi: Update x, y coordinates (permuting the lanes), temporarily store the current states value for the next iteration

```
    // Update coordinates
    int newX = y;                          // x is now equal to y
    int newY = (2 * x + 3 * y) % 5;        // y is now equal to new y position
    uint64_t tempLane = state[newX][newY]; // Temp store value at new coords
```

- Rho: 'ROTL64' bitwise shift, rotates the 64-bit lane 'n' positions left, where n is equal to the rotation offset

```
// Rotate the current lane and update state at new coordinates
uint64_t rotatedBits = ROTL64(lane, rotationOffset);
```

- Pi: Update state at the new x, y position with the rotated lane

```
    state[newX][newY] = rotatedBits;  // Update state at position newX, newY
```

- Prepare for next loop iteration

```
    // Update lane and x,y coordinates for next iteration
    lane = tempLane;
    x = newX;
    y = newY;

  }
}
```

CHI STEP

The Chi step introduces non-linearity to the state matrix. Linearity is when a change to the input results in an expected output result. In non-linearity the output is made intentionally complex and unpredictable, subsequently the algorithm is made considerably more secure and resilient to attacks.

```
void chi(uint64_t state[5][5]) {

  uint64_t C[5]; // Initialise state array to store planes (rows)
  // Processes each of the 5 planes/rows in state array
  for (int y = 0; y < 5; y++) {
    // first inner loop
    for (int x = 0; x < 5; x++) {
```

- C = temporary store for current row values

```
      C[x] = state[x][y]; // store value of current plane/row from state array
    }
    for (int x = 0; x < 5; x++) {
```

1. Bitwise NOT, inverts bits: $\sim c[(x + 1) \% 5]$ one position "to the right" of the current lane
2. Bitwise AND, combines bits: & $c[(x + 2) \% 5]$ is the lane two positions "to the right" of the current lane
3. XOR the current lane with the result from the previous steps and update the state

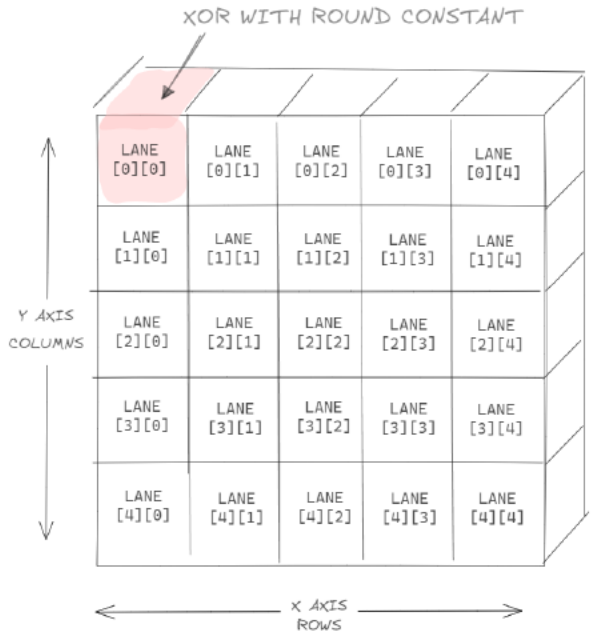
```
      state[x][y] = (C[x] ^ ((~C[(x+1) % 5]) & C[(x+2) % 5]));
    }
  }
}
```

IOTA

The Iota step is straightforward

- Adds const 'RC' from the round constants table
- Lane[0,0] is XOred with the round constant value

```
void iota(uint64_t state[5][5], int round) {
  state[0][0] ^= RC[round];
}
```



Round Constants Table:

```
// round constants
const uint64_t RC[24] = {
  0x0000000000000001, 0x0000000000008082, 0x800000000000808a,
  0x8000000080008000, 0x000000000000808b, 0x0000000080000001,
  0x8000000080008081, 0x8000000000008009, 0x000000000000008a,
  0x0000000000000088, 0x0000000080008009, 0x000000008000000a,
  0x000000008000808b, 0x800000000000008b, 0x8000000000008089,
  0x8000000000008003, 0x8000000000008002, 0x8000000000000080,
  0x000000000000800a, 0x800000008000000a, 0x8000000080008081,
  0x8000000000008080, 0x0000000080000001, 0x8000000080008008
};
```

REFERENCES

[1] G.Bertoni, J. Daemen, M. Peeters, and G. Assche, "The K SHA-3 submission," 2011. [Online]. Available: https://keccak.team/files/Keccak-submission-3.pdf

[2] G.Bertoni, J. Daemen, M. Peeters, and G. Assche, "The K reference," 2011. [Online]. Available: https://keccak.team/files/Keccak-reference-3.0.pdf

[3] M. J. Dworkin, "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions," Jul. 2015, doi: https://doi.org/10.6028/nist.fips.202.

[4] G. Bertoni, J. Daemen, M. Peeters, and G. Assche, "Keccak sponge function family main document," 2010. [Online]. Available: https://keccak.team/obsolete/Keccak-main-2.1.pdf

[5] Wikipedia Contributors, "NIST hash function competition," Wikipedia, Feb. 28, 2024. https://en.wikipedia.org/wiki/NIST_hash_function_competition

[6] Q. Dang, R. Blank, and P. Gallagher, “NIST Special Publication 800-107 Revision 1 Recommendation for Applications Using Approved Hash Algorithms,” 2012. Available: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-107r1.pdf

[7] NIST, “SHA-3 Selection Announcement.” Available: https://csrc.nist.gov/csrc/media/projects/hash-functions/documents/sha-3_selection_announcement.pdf

[8] G. Bertoni, J. Daemen, M. Peeters, and G. Assche, “Sponge Functions.” Available: https://keccak.team/files/SpongeFunctions.pdf

[9] S. Debnath, A. Chattopadhyay, and S. Dutta, “Brief review on journey of secured hash algorithms,” 2017 4th International Conference on Opto-Electronics and Applied Optics (Optronix), Nov. 2017, doi: https://doi.org/10.1109/optronix.2017.8349971.

[10]“Keccak is SHA-3 - Schneier on Security,” Schneier.com, 2021. https://www.schneier.com/blog/archives/2012/10/keccak_is_sha-3.html

[11]chemejon, “SHA-3 Explained in Plain English,” Jon’s Blog, Dec. 06, 2021. https://chemejon.wordpress.com/2021/12/06/sha-3-explained-in-plain-english