Mathematical Foundations of SHA-256 and Merkle Trees: Exploring Number Theory, Tree Structures, and Algorithmic Complexity in Blockchain

Aurelia Jennifer Gunawan – 13524089[1,2]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: [1]aurellia.jennifer@gmail.com , [2]13524089@std.stei.itb.ac.id

***Abstract*—This paper explores the role of number theory, tree theory, and algorithmic complexity in building secure and efficient blockchain systems through SHA-256 and Merkle trees. SHA-256 is analyzed using number-theoretic concepts such as modular arithmetic and prime-based constants, while Merkle trees are explored as binary structures enabling efficient proof-of-membership and data integrity. The paper also contrasts the time complexity of Merkle tree transaction verification with linear data structures. The findings highlight the foundational role of discrete mathematics in ensuring blockchain security, scalability, and immutability.**

***Keywords*— blockchain, SHA-256, Merkle tree, number theory, tree structure, time complexity**

I. Introduction

The development of blockchain technology marks a significant advancement in decentralized and tamper-resistant data systems. A blockchain is a distributed ledger composed of sequential data blocks, each cryptographically linked to its predecessor. This structure ensures immutability, as altering a single block would invalidate all subsequent blocks and making tampering computationally impractical without majority control. Consensus mechanisms, such as Proof of Work or Proof of Stake, ensure trust and coordination among participating nodes.

Each block consists of a header, which contains metadata such as a timestamp, cryptographic nonce, Merkle root, and previous block hash, and a body comprising verified transactions. The inclusion of the previous hash forms a secure, hash-linked chain of blocks.

Blockchain's integrity and efficiency rely heavily on two core components, the SHA-256 hash function and the Merkle tree. Although widely used in systems like Bitcoin, their mathematical foundations are often underappreciated. SHA-256 draws from number theory, including modular arithmetic and prime-based constant derivation, while Merkle trees are rooted in tree theory, enabling efficient and verifiable proofs of data inclusion through recursive binary hashing.

II. Theoretical Basis

A. Foundations of Number Theory

Number theory is a branch of pure mathematics devoted to the study of integers and the relationships between them. It deals with fundamental concepts such as divisibility, prime numbers, greatest common divisor, and modular arithmetic. Despite its abstract origin, number theory is highly relevant in modern computer science applications, including cryptography and hashing.

1) Integers and Divisibility

Number theory begins with the study of integers and the rules of divisibility. And integer a is said to divide another integer b (denoted $a \mid b$) if there exists an integer c such that $b = ac$. This fundamental notion leads to concepts like prime numbers, common divisors, and congruences.

2) The Eucledian Algorithm and Greatest Common Divisor (GCD)

The greatest common divisor (GCD) of two non-zero integers a and b denoted gcd(a, b), is the largest integer that divides both. The Euclidean Algorithm provides an efficient method to compute the GCD using repeated division:

$$\begin{aligned}
r_0 &= r_1q_1 + r_2 && 0 \le r_2 < r_1, \\
r_1 &= r_2q_2 + r_3 && 0 \le r_3 < r_2, \\
&\cdots \\
r_{n-2} &= r_{n-1}q_{n-1} + r_n && 0 \le r_n < r_{n-1} \\
r_{n-1} &= r_nq_n + 0
\end{aligned}$$

The last non-zero remainder is the GCD.

3) Linear Combinations

An important property of the GCD is its representations as a linear combination of the two integers:

$$\gcd(a,b) = ma + nb$$

For some integers m and n. This result, known as Bezout's Identity, is foundational in solving Diophantine equations and computing modular inverses.

4) Prime Numbers and The Fundamental Theorem of Arithmetic

A prime number is a positive integer greater than 1 that has no divisors other than 1 and itself. The Fundamental Theorem of Arithmetic states that every integer $n \geq 2$ can be uniquely factored into prime numbers (up to order). This property forms the backbone of number theory and underlies many cryptographic systems.

5) Relatively Prime Numbers

Two integers a and b are said to be relatively prime, if gcd(a, b) = 1. When this condition holds, there always exist integers m and n such that:

$$ma + nb = 1$$

This property is essential in modular arithmetic, particularly in determining the existence of modular inverses.

6) Modular Arithmetic

Modular arithmetic is a central topic in number theory where numbers are considered "wrapped around" after reaching a certain value called the modulus. For a positive integer m, two integers a and b are congruent modulo m if:

$$a \equiv b \ (mod\ m) \Leftrightarrow \ m|(a-b)$$

Modular arithmetic is governed by a set of well-defined rules that dictate how arithmetic operations such as addition, multiplication, and exponentiation behave under a given modulus.

Let m be a positive integer.

1. *If $a \equiv b(mod\ m)$ and c is any integer, then:*
   a. $a + c \equiv b + c(mod\ m)$
   b. $ac \equiv bc(mod\ m)$
   c. $a^p \equiv b^p(mod\ m)$ *for any non-negative integer p*

   This means both sides can be added, multiplied, or raised to a power by the same constant.
2. *If $a \equiv b(mod\ m)$ and $c \equiv d(mod\ m)$ then:*
   a. $a + c \equiv b + d(mod\ m)$
   b. $ac \equiv bd(mod\ m)$

   This means two congruent integers (modulo is the same number) can be added or multiplied.

However, this set of rules does not include division operations, because dividing both sides of a congruence by an integer does not always preserve the congruence. That is, congruence under modular arithmetic is not guaranteed to hold after division.

7) Modular Inverses

Given integers a and m such that gcd(a, m) = 1, there exists a unique modular inverse of a mod m, denoted a^{-1}, satisfying:

$$a \cdot a^{-1} \equiv 1(mod\ m)$$

This inverse can be computed using the extended Euclidean algorithm, and is crucial in solving linear congruences and in cryptographic algorithms such as RSA.

8) Systems of Congruence and The Chinese Remainder Theorem

A system of linear congruences involves finding an integer x that satisfies multiple modular conditions. If the moduli are pairwise coprime, the Chinese Remainder Theorem (CRT) guarantees a unique solution modulo the product in the moduli. Formally, for:

$$x \equiv a_1(mod\ m_1)$$
$$x \equiv a_2(mod\ m_2)$$
$$\dots$$
$$x \equiv a_n(mod\ m_n)$$

with gcd(m_i, m_j) = 1 for all $i \neq j$, there exists a unique solution modulo $M = m_1 m_2 \dots m_n$.

9) Fermat's Little Theorem

Let p be a prime and a and integer such that gcd(a, p) = 1. Then:

$$a^{p-1} \equiv 1(mod\ p)$$

This result, known as Fermat's Little Theorem, is widely used in modular exponentiation and public key cryptography

B. Tree Theory

A tree is rigorously defined as an undirected graph that possesses two fundamental characteristics:

1. it must be connected, and
2. it must not contain any circuits or cycles.

This definition implies that in a tree, there is always a unique simple path between any two vertices, and the addition of any single edge between non-adjacent vertices will invariably create exactly one circuit.

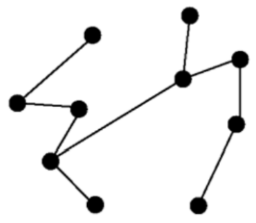

Fig. 2. Tree Illustration
Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/23-Pohon-Bag1-2024.pdf

For a finite, undirected graph G with n vertices and m edges, the following seven properties are demonstrably equivalent, meaning if any one of them is true, all others are also true, thus characterizing G as a tree:

1. G is a tree.
2. Every pair of distinct vertices in G is connected by exactly one simple path.
3. G is connected and contains exactly m = n – 1 edges.
4. G contains no circuits and possesses exactly m = n – 1 edges.

5. G contains no circuits, and the addition of any new edge to G (between any two non-adjacent vertices) results in the formation of exactly one circuit.
6. G is connected and the removal of any single edge disconnects the graph.
7. G is a connected graph in which every edge is a bridge.

a) Forest

A forest is a collection of one or more disjoint trees, Alternatively, it can be conceptualized as an undirected graph that is not necessarily connected but contains no circuits. In a forest, each of its connected components is a tree. This means that if a graph is acyclic but disconnected, it is considered a forest where each connected segment forms an individual tree.

b) Spanning Trees

For any given connected graph G, a spanning tree is defined as a spanning subgraph of G that is also a tree. A spanning subgraph is one that includes all the vertices of the original graph G. A spanning tree is constructed by selectively removing edges from the original graph G such that all vertices remain connected, no cycles are formed, and the total number of edges is minimized to n - 1. Every connected graph is guaranteed to possess at least one spanning tree.

In the context of disconnected graphs, if a graph has k connected components, then it will have k corresponding spanning trees, one for each component, these k spanning trees collectively form a spanning forest.

c) Minimum Spanning Trees (MST)

When dealing with a connected and weighted graph, a minimum spanning tree is a spanning tree whose sum of edge weights is the smallest among all potential spanning trees of that graph. The concept of an MST is paramount in optimization problems where the goal is to connect all nodes in a network with the lowest possible cumulative "cost".

There are two widely used algorithms to determine an MST:

1. Prim's Algorithm
   This algorithm builds an MST by iteratively adding the minimum-weight edge that connects a vertex in the growing tree to a vertex outside it, avoiding cycles, until all vertices are included.
2. Kruskal's Algorithm
   This algorithm forms an MST by sorting all edges by weight in ascending order and iteratively adding edges that do not create a cycle, until n-1 edges have been selected.

It is important to note that while the total weight of an MST of a given weighted graph is unique, the specific set of edges that constitute the MST may not be unique. This non-uniqueness can occur if there are multiple edges with the same weight that could be chosen at a particular step in the algorithm, leading to different valid MST structures with the identical minimum total weight.

d) Rooted Trees

A rooted tree is a specialized form of tree where one vertex is designated as the root. In a rooted tree, a natural hierarchical structure emerges. The edges are conventionally understood to be directed away from the root, establishing parent-child relationships. Key terminology associated with rooted trees includes:

1. Parent
   The vertex directly above another vertex on the path from the root.
2. Child
   A vertex directly connected to and below a parent vertex.
3. Ancestor
   Any vertex on the path from the root to a given vertex, excluding the vertex itself.
4. Descendant
   Any vertex in the subtree rooted at a given vertex, excluding the vertex itself.
5. Sibling
   Vertices that share the same parent.
6. Leaf
   A vertex with no children.
7. Internal Node
   A vertex that has at least one child.
8. Level/Depth
   The length of the path from the root to a given vertex. The root is typically at level 0.
9. Height
   The maximum level of any leaf in a tree, or the length of the longest path from the root to a leaf.

C. Algorithmic Complexity

In an algorithm, efficiency is considered equally important as correctness. An efficient algorithm aims to minimize the computational resources required, specifically time and space, relative to the input size *n*. The time complexity, *T(n)*, is the number of elementary steps or operations as a function of *n*, while the space complexity, *S(n)*, indicates the amount of memory consumed during execution.

Algorithm performance is commonly expressed using asymptotic notations, such as:

1. Big-O
   The Big-O notation, $O\big(f(n)\big)$, defines an upper bound on the growth rate of an algorithm's runtime by focusing on its dominant term as $n \to \infty$. For instance, an algorithm with $T(n) = 2n^2 + 6n + 1$ is classified as $O(n^2)$, as the quadratic term dominates the overall behavior for large inputs.
2. Big-Omega
   The Big-Omega notation, $\Omega(f(n))$, defines a lower bound.

3. Big-Theta
   The Big-Theta notation, $\Theta(f(n))$, provides a tight bound when both upper and lower growth rates are equivalent.

Common time complexity classes include:

1. Constant time complexity ($O(1)$)
2. Logarithmic time complexity ($O(\log n)$)
3. Linear time complexity ($O(n)$)_
4. Polynomial time complexity (i.e., $O(n^2), O(n^5)$)
5. Exponential and factorial time complexity (i.e., $O(2^n), O(n!)$)

These classifications facilitate comparison between algorithms solving the same problem. For instance, selection sort and buble sort operate in $O(n^2)$, while quicksort runs in $O(n \log n)$ on average, offering better performance for large input sizes.

III. The SHA-256 Hash Function

A. Overview of SHA-256

SHA-256 (Secure Hash Algorithm 256) was developed by the National Security Agent (NSA) and published by the National Institute of Standards and Technology (NIST) in 2001. It was introduced to replace SHA-1, which had become increasingly susceptible to brute-force and collision attacks. As a cryptographic hash function, SHA-256 produces a fixed-length 256-bit hash output regardless of the size of the input messages.

According to the NIST specification for SHA-256, SHA-256 processes data in fixed-size blocks. For each block, it creates a message schedule of 64 words, 32 bits each, labeled W_0, W_1, …, W_{63}. The algorithm also uses eight 32-bit working variables, labeled *a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*, along with two temporary values, T_1 and T_2, to compute the compression function. The hash values are updated in each round using logical and arithmetic operations like bitwise shifts and modular addition. These values are stored in eight 32-bit words, $H_0^{(i)}$ to $H_7^{(i)}$, which are updated with each processed block. After all blocks are processed, the final 256-bit hash, $H^{(N)}$, represents the unique fingerprint of the input data.

B. SHA-256 Computation Process

1) SHA-256 Preprocessing

Before computation begins, the input message *M* undergoes a preprocessing phase to ensure it meets the specific formatting requirements.

a. Converting the Message into Binary

To start the hashing process, the message is first transformed into its binary representation by encoding each character using the American Standard Code for Information Interchange (ASCII) standard.

As an example, consider the message "portsmouth". The table below presents the binary representation of each character based on its ASCII encoding.

Character	Binary
p	01110000
o	01101111
r	01110010
t	01110100
s	01110011
m	01101101
o	01101111
u	01110101
t	01110100
h	01101000

Tab. 3.1. Binary representation of each character in "portsmouth"

b. Padding the Message

The message is padded to ensure its length (in bits) is congruent to 448 module 512. Padding begins with the addition of a single '1' bit, followed by a sequence of '0' bits, whose length is determined by how many bits are needed to reach the required alignment. Finally, a 64-bit big-endian integer representing the original length of the message is appended to the end of the padded message. This guarantees that the total length of the message is a multiple of 512 bits, and it allows the algorithm to process the message in fixed-size blocks.

From previous example, each character is represented using 8 bits, and these binary values are combined into one continuous string. After that, a single '1' bit is appended to the end of the string, as follows:

```
01110000011011110111001001110100011100110110110101
1011110111010101110100011010001
```

Next, we determine the length of the original message. In this example, the message is 10 characters long, each occupying 8 bits, giving a total length of 80 bits. This length is then converted into binary.

$$80 = 0101000$$

After that, we append this binary length value to the end of the message. Before doing so, we insert a series of '0's between the message and the length, padding the data until it reaches a total size of 512 bits.

```
01110000011011110111001001110100011100110110110101
10111101110101011101000110100010000000000000000000
00000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
000001010000
```

If the original message exceeds 512 bits, it is divided into multiple 512-bit blocks, and the length value is added only to the final block.

c. Parsing the Message

Once the message has been padded, it is divided into *N* 512-bit blocks, labeled $M^{(1)}$, $M^{(2)}$, …, $M^{(N)}$. These blocks serve as the input units for the main computation process.

0. 01110000011011110111001001110100
1. 01110011011011010110111101110101
2. 01110100011010001000000000000000

.

.

.

15. 00000000000000000000000001010000

d. Setting Initial Hash Values

The algorithm initializes eight working hash values $H_0^{(0)}, H_1^{(0)}, \ldots, H_7^{(0)}$, each a 32-bit word. In SHA-256, the initial hash values are derived from the square roots of the first eight prime numbers. These values are fixed and constant regardless of the input message. To compute them, each prime number is square rooted, and the fractional part (i.e., the value modulo 1) is extracted. This fractional portion is then multiplied by 2^{32}, or equivalently 16^8, and the result is rounded down to the nearest whole number to produce the final initial constant.

$$int((\sqrt{p}\ mod\ 1) * 2^{32})$$

Next, we convert the results into hexadecimal form.

$$a = H_0^{(0)} = 6a09e667$$
$$b = H_1^{(0)} = bb67ae85$$
$$c = H_2^{(0)} = 3c6ef372$$
$$d = H_3^{(0)} = a54ff53a$$
$$e = H_4^{(0)} = 510e527f$$
$$f = H_5^{(0)} = 9b05688c$$
$$g = H_6^{(0)} = 1f83d9ab$$
$$h = H_7^{(0)} = 5be0cd19$$

These initial values form the starting state of the hashing procedure and are subsequently updated during the processing of each message block.

2) SHA-256 Hash Computation Process

For each message block $M^{(i)}$, the algorithm follows four major steps as specified in FIPS PUB 180-4.

a. Prepare the Message Schedule W_t

The message schedule consists of 64 32-bit words

$$\mathrm{W_t} = \begin{cases} M_t^{(i)}, & 0 \le t \le 15 \\ \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16}, & 16 \le t \le 63 \end{cases}$$

Here, σ_0 and σ_1 are bitwise functions defined as:

$$\sigma_0(x) = (x \gg 7) \oplus (x \gg 18) \oplus (x \gg 3)$$
$$\sigma_1(x) = (x \gg 17) \oplus (x \gg 19) \oplus (x \gg 10)$$

For example, this is the calculation of the 17th word, W_{16}, to illustrate the process. To calculate W_{16}, we need to obtain W_0, $\sigma_0(W_1)$, W_9, and $\sigma_1(W_{14})$.

W_0 = 01110000011011110111001001110100
W_9 = 00000000000000000000000000000000

W_1 = 01110011011011010110111101110101

$$\sigma_0(W_{t-15}) = \sigma_0(W_1) = (W_1 \gg 7) \oplus (W_1 \gg 18) \oplus (W_1 \gg 3)$$

$(W_1 \gg 7) =$	11101010111001101101101011011110
$(W_1 \gg 18) =$	01011011110111010101110011011011
$(W_1 \gg 7) \oplus (W_1 \gg 18) =$	10110001001110111000011000000101
$(W_1 \gg 3) =$	00001110011011011010110111101110
$\sigma_0(W_1) =$	10111111010101100010101111101011

W_{14} = 00000000000000000000000000000000

$$\sigma_1(W_{14}) = (W_{14} \gg 17) \oplus (W_{14} \gg 19) \oplus (W_{14} \gg 10)$$

$(W_{14} \gg 17) =$	00000000000000000000000000000000
$(W_{14} \gg 19) =$	00000000000000000000000000000000
$(W_{14} \gg 17) \oplus (W_{14} \gg 19) =$	00000000000000000000000000000000
$(W_{14} \gg 10) =$	00000000000000000000000000000000
$\sigma_1(W_{14}) =$	00000000000000000000000000000000

$$W_{16} = \sigma_1(W_{14}) + W_9 + \sigma_0(W_1) + W_0$$

$\sigma_1(W_{14}) =$	00000000000000000000000000000000
$W_9 =$	00000000000000000000000000000000
$\sigma_1(W_{14}) + W_9 =$	00000000000000000000000000000000
$\sigma_0(W_1) =$	10111111010101100010101111101011
$\sigma_1(W_{14}) + W_9 + \sigma_0(W_1) =$	10111111010101100010101111101011
$W_0 =$	01110000011011110111001001110100
$W_{16} =$	00101111110001011001111001011111
Hexadecimal =	2fc59e5f

b. Initialize Working Variables

Eight working variables *a*, *b*, *c*, *d*, *e*, *f*, *g*, *h* are initialized with the hash value from the previous block.

$$a = H_0^{(i-1)}$$
$$b = H_1^{(i-1)}$$
$$c = H_2^{(i-1)}$$
$$d = H_3^{(i-1)}$$
$$e = H_4^{(i-1)}$$
$$f = H_5^{(i-1)}$$
$$g = H_6^{(i-1)}$$
$$h = H_7^{(i-1)}$$

c. Main Compression loop (for t =0 to 63)

Each round computes two temporary values, T_1 and T_2.

$$T_1 = h + \sum\nolimits_1 (e) + Ch(e,f,g) + K_t + W_t$$
$$T_2 = \sum\nolimits_0 (a) + Maj(a,b,c)$$

Where the bitwise operations are as follows:

$$\sum\nolimits_0 (x) = (x \gg 2) \oplus (x \gg 13) \oplus (x \gg 22)$$
$$\sum\nolimits_1 (x) = (x \gg 6) \oplus (x \gg 11) \oplus (x \gg 25)$$
$$Ch(x,y,z) = (x \wedge y) \oplus (\neg x \wedge z)$$
$$Maj(x,y,z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$$

$K_t = A\ set\ constant\ (different\ for\ each\ iteration)$

Then, the working variables are updated.

$$h = g$$
$$g = f$$
$$f = e$$
$$e = d + T_1$$
$$d = c$$

$$c = b$$
$$b = a$$
$$a = T_1 + T_2$$

All additions are performed modulo 2^{32}.

For example, this is the computation for the first iteration (t =0), to illustrate the process. Firstly, we will find T_1, the calculation is as follows:

e = 510e527f = 01010001000011100101001001111111
f = 9b05688c = 10011011000001010110100010001100
g = 1f83d9ab = 00011111100000111101100110101011
h = 5be0cd19 = 01011011111000001100110100011001

$\sum_1(e) = (e \gg 6) \oplus (e \gg 11) \oplus (e \gg 25)$

$(e \gg 6) =$ 11111101010001000011100101001001
$(e \gg 11) =$ 01001111111010100010000111001010
$(e \gg 6) \oplus (e \gg 11) =$ 10110010101011100001100010000011
$(e \gg 25) =$ 10000111001010010011111110101000
$\Sigma_1(e) =$ 00110101100001110010011100101011

$Ch(e,f,g) = (e \wedge f) \oplus (\neg e \wedge g)$

$(e \wedge f) =$ 00010001000001000100000000001100
$(\neg e \wedge g) =$ 00001110100000011000100110000000
$Ch(e,f,g) =$ 00011111100001011100100110001100

$T_1 = h + \sum_1(e) + Ch(e,f,g) + K_t + W_t$

h = 01011011111000001100110100011001
$\Sigma_1(e) =$ 00110101100001110010011100101011
$Ch(e,f,g) =$ 00011111100001011100100110001100
$K_0 =$ 01000010100010100010111110011000
$W_0 =$ 01110000011011110111001001110100
$T_1 =$ 01100011111001110101111111011100

Secondly, we will find T_2, the calculation is as follows:

a = 6a09e667 = 01101010000010011110011001100111
b = bb67ae85 = 10111011011001111010111010000101
c = 3c6ef372 = 00111100011011101111001101110010

$\sum_0(a) = (a \gg 2) \oplus (a \gg 13) \oplus (a \gg 22)$

$(a \gg 2) =$ 11011010100000100111100110011001
$(a \gg 13) =$ 00110011001110110101000001001111
$(a \gg 2) \oplus (a \gg 13) =$ 11101001101110010010100111010110
$(a \gg 22) =$ 00100111100110011001110110101000
$\Sigma_0(a) =$ 11001110001000000101101000111110

$Maj(a,b,c) = (a \wedge b) \oplus (a \wedge c) \oplus (b \wedge zc)$

$(a \wedge b) =$ 00101010000000001101001100000101
$(a \wedge c) =$ 00101000000010001110001001100010
$(a \wedge b) \oplus (a \wedge c) =$ 00000010000010010100010001100111
$(b \wedge c) =$ 00111000011001101010001000000000
$Maj(a,b,c) =$ 00111010011011111110011001100111

$T_2 = \sum_0(a) + Maj(a,b,c)$

$\Sigma_0(a) =$ 11001110001000000101101000111110
$Maj(a,b,c) =$ 00111010011011111110011001100111
$T_2 =$ 00001000100100000100110101100101

Next, we can calculate a and e with the previously obtained T_1 and T_2.

a = T_1+T_2
= 01101100011101111111101011000001
= 6c77fac1
d = a54ff53a
= 10100101010011111111010100111010
e = d + T_1
= 00001001001101110101010100010110

Finally, we now have the final values for the first iteration (t = 0)

a	6c77fac1
b	6a09e667
c	bb67ae85
d	3c6ef372
e	09375516
f	510e527f
g	9b05688c
h	1f83d9ab

Tab. 3.2. Final hash values after the 1st iteration

This process will be performed for 63 more iterations, using the values from the previous iteration, which are being updated every round. At the 64th iteration, we will have the final values as below.

a	d2c53dd2
b	a56ddf12
c	cec9ec4d
d	013bf08b
e	03b8f3ad
f	fb054bf4
g	73bad7c0
h	b197c87e

Tab. 3.3. Final values after the 64th iteration

d. Compute Intermediate Hash Values

After completing all 64 rounds, update the intermediate hash values.

$$H_j^{(i)} = H_j^{(i-1)} + Current\ value\ of\ corresponding\ variable, 0 \le j \le 7$$

Using the final values from the previous example, we can obtain the intermediate hash values and its hexadecimal form.

For instance, this is the calculation for $H_0^{(i)}$, to illustrate the process.

$$H_0^{(i)} = H_0^{(i-1)} + H_0^{(0)}$$

$H_0^{(i-1)}$ = a

$H_0^{(i)}$ = 11010010110001010011110111010010 +
01101010000010011110011001100111
= 00111100110011110010010000111001

Lastly, this is the complete conversion of the intermediate hash values into hexadecimal form.

$H_j^{(i)}$	Binary	Hexadecimal
$H_0^{(i)}$	00111100110011110010010000111001	3ccf2439
$H_1^{(i)}$	01100000110101011000110110010111	60d58d97
$H_2^{(i)}$	00001011001110001101111110111111	0b38dfbf
$H_3^{(i)}$	10100110100010111110010111000101	a68be5c5

$H_4^{(i)}$	01010100110001110100011000101100	54c7462c
$H_5^{(i)}$	10010110000010101011010010000000	960ab480
$H_6^{(i)}$	10010011001111101011000101101011	933eb16b
$H_7^{(i)}$	00001101011110001001010110010111	0d789597

Tab. 3.4. Conversion of every intermediate hash values into hexadecimal

After all N message blocks are processed, the final 256-bit hash is the concatenation of

$$H(M) = H_0^{(N)}||H_1^{(N)}||H_2^{(N)}||H_3^{(N)}||H_4^{(N)}||H_5^{(N)}||H_6^{(N)}||H_7^{(N)}$$

Finally, the final 256-bit hash for the message "portsmouth" is 3ccf243960d58d970b38dfbfa68be5c554c7462c960ab480933eb16b0d789597.

C. *Characteristics of SHA-256*

1) *Digest Length*

SHA-256 generates a hash output of exactly 256 bits (32 bytes), as indicated by its name. This fixed-length output ensures consistency, regardless of the input size.

2) *Irreversible Quality (Preimage Resistance)*

SHA-256 is a one-way function, this means it is computationally infeasible to reverse the process. The original input data is not retrievable with only the hash digest.

3) *Input Sensitivity*

SHA-256 is sensitive to case and white space, meaning that case changes, white space variations, and formatting differences produces a completely different hash. This ensures security against collision attacks.

String	SHA-256
"test"	9f86d081884c7d659a2feaa0c55ad015a3bf4f1b2b0b822cd15d6c15b0f00a08
"TEST"	94ee059335e587e501cc4bf90613e0814f00a7b08bc7c648fd865a2af6a22cc2
" TEST"	e7885219bfbeeb8b283a4c94fe67610458fb395bd96a56eefdcf325d98702a67

Tab. 3.5. Examples of Input Sensitivity in SHA-256

IV. Merkle Trees in Blockchain

A. *Overview of Blockchain and Block Structure*

Blockchain technology is built on two hash-based data structures, linked lists with hash pointers and Merkle trees, to achieve a secure and tamper-resistant system. At its core, a blockchain functions as a linked list of data blocks, where each block references its predecessor using a hash pointer instead of a regular pointer. Within each individual block, transactions are systematically organized using a Merkle tree. The root of this tree, known as the Merkle root, is included in the block header and serves as a compact and verifiable summary of all transactions in the block.

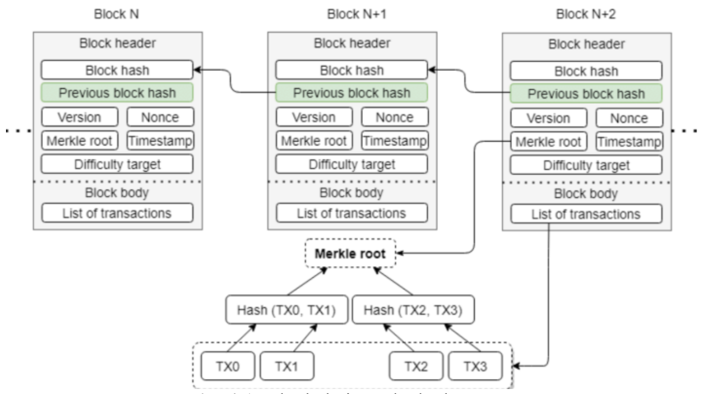

Fig. 4.1. Blockchain and Block Structure
Source: https://www.researchgate.net/figure/Blockchain-and-block-structure_fig1_351730117

B. *Overview of Linked List with Hash Pointer*

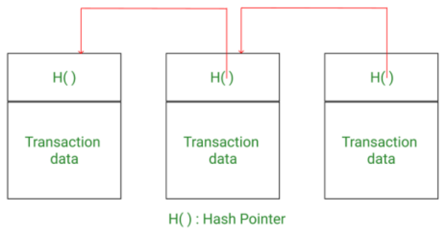

Fig. 4.2. Blockchain as linked list with hash pointer
Source: https://www.geeksforgeeks.org/software-engineering/blockchain-merkle-trees/

Unlike a regular pointer that indicates a memory location, a hash pointer directs to the data's storage and incorporates cryptographic hash of the data. This dual functionality allows hash pointers to both retrieve the data and simultaneously confirm its unaltered state.

In blockchain's architecture, each block holds transaction data and a hash pointer that cryptographically binds it to the preceding block. This hash pointer functions as a unique digital fingerprint of the previous block's header. This design makes the chain highly tamper evident. Any modification to historical data would necessitate recalculating the hashes of all subsequent blocks, which is an extraordinarily difficult computational feat, therefore ensuring the data's immutability.

C. *Overview of Merkle Tree*

A Merkle tree, also known as a hash tree, is a cryptographic data structure that organizes data in the form of a binary tree. This structure, originally proposed by Ralph Merkle in the late 1970s, is widely used in blockchain systems to provide data integrity, enable proof of inclusion, and detect tampering with minimal computational overhead.

In a blockchain block, each leaf node of the Merkle tree contains the cryptographic hash of a single transaction. Each internal node holds the hash of the combined hashes of its immediate child nodes. The process of recursive hashing continues upward until a single hash value, known as the Merkle root, is obtained. This root serves as a compact summary of all transactions in the block and is stored in the block header.

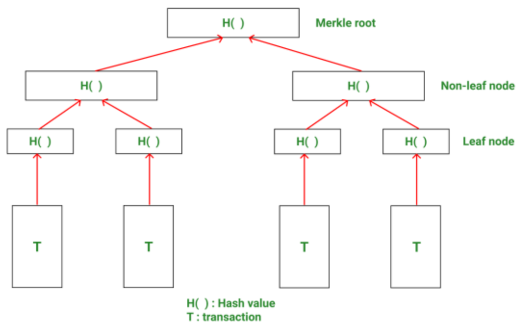

Fig. 4.3. Merkle Tree Structure
Source: https://www.geeksforgeeks.org/software-engineering/blockchain-merkle-trees/

To preserve the binary structure, Merkle trees require an even number of leaf nodes, If the number of transactions is odd, the final hash is duplicated to maintain balance. Merkle trees are constructed from the bottom, this means transaction data is hashed at the leaf level and successive levels are generated by hashing pairs of child nodes, ultimately forming the Merkle root. This results in an inverted binary tree structure, where data flows upward.

D. *Merkle Proof*

Merkle proof, also known as proof of inclusion, is a cryptographic method to verify whether a piece of data belongs to a specific Merkle tree without revealing the full dataset. Instead of traversing the entire tree or storing all transactions, only the hashes along the path from the target leaf to the root are needed for verification. Merkle proof is built on the principle of one-way hashing (i.e., SHA-256), which ensures that two distinct inputs do not produce the same hash, and that each hash uniquely represents the data it was derived from.

For example, let's look at the diagram below to illustrate the process.

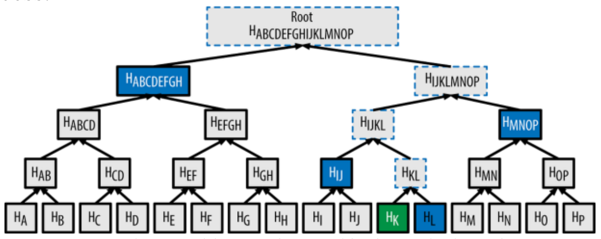

Fig. 4.4. Using Merkle Proof to Verify the Inclusion of Data K
Source: https://medium.com/crypto-0-nite/merkle-proofs-explained-6dd429623dc5

Firstly, a one-way hash function is applied to K, which results in H_K. Without revealing K, we combine H_K with H_L to obtain H_{KL}. Then, we combine H_{KL} with H_{IJ} to form H_{IJKL}. Next, we combine H_{IJKL} with H_{MNOP} and $H_{ABCDEFGH}$ until we reconstruct the Merkle root. If this matches the known root, it confirms K's inclusion.

V. Time Complexity and Computational Efficiency

The advantage of implementing Merkle tree structures within blockchain systems lies in their computational efficiency. This efficiency is reflected when verifying the presence of a specific transaction within a block. Merkle tree's hierarchical structure allows for logarithmic time verification using Merkle proof.

Let n represent the total number of transactions (leaf nodes) in a block. Since the tree is binary and full, with the assumption that n is a power of two, the number of levels or the height of the tree is:

$$h = \log_2 n$$

To verify whether a particular transaction T is present in the Merkle tree, a Merkle proof is performed. This process involves:

1. Computing the hash of T, which is a constant-time operation.
2. Retrieving and combining $\log_2 n$ sibling hashes along the path from the leaf node to the root.
3. Performing $\log_2 n$ hash operations, because a binary tree of n leaves has the height of $\log_2 n$, to iteratively reconstruct the Merkle root.
4. Compare the resulting Merkle root with the one stored in the block header.

Thus, the number of hash computations required is:

$$Number\ of\ hash\ operations = \log_2 n$$

Assume that every hash operation take constant time $O(1)$, given the fixed output size of SHA-256 and the uniformity of input sizes during tree traversal. Therefore, the total time complexity, in Big-O notation, for the Merkle proof is:

$$T(n) = \log_2 n = O(\log n)$$

In contrast, if the transactions are stored in a linear data structure, like a linked list, verifying the inclusion of T will require linear traversal from the head of the list. In the worst case scenario, the desired transaction resides at the end of the list or may not exist, requiring inspection of all n element:

$$T(n) = O(n)$$

In conclusion, the Merkle tree, achieving logarithmic time complexity $O(\log n)$, offers a faster mechanism for transaction verification than the linear data structure (i.e., a linked list), which require $O(n)$ time.

VI. Conclusion

In this paper, we explored the foundational mathematical concepts that underpin the operation of SHA-256 and Merkle trees within blockchain systems. By analyzing SHA-256 through the lens of number, we highlighted how modular arithmetic, and prime-derived constants contribute to its cryptographic strength.

Concurrently, we studied the structure of Merkle trees using concepts from tree theory and algorithmic complexity. We demonstrated how their hierarchical construction allows for efficient proof-of-membership operations in O(log n) time, providing a scalable and secure method of transaction verification. In contrast to linear data structures, Merkle trees offer significant performance advantages in large-scale, decentralized environments.

SHA-256 and Merkle trees illustrate how discrete mathematics serves as a critical foundation for secure and efficient blockchain architecture. This work reaffirms that the interplay between theoretical constructs and practical engineering is essential in the ongoing development of robust cryptographic systems.

Acknowledgment

First and foremost, the author would like to express gratitude to God for the guidance, and perseverance throughout the semester and during the completion of this paper. The author is also deeply thankful to the lecturers of the IF1220 Discrete Mathematics course at ITB, Dr. Ir. Rinaldi, M.T., and Arrival Dwi Sentosa, M.T., for the dedication to share their knowledge and guide their students throughout the course. Their guidance played a significant role in deepening the author's understanding of the subject. Appreciation is also extended to the author's friends, and family for their continuous encouragement and motivation throughout the semester and during the writing of this paper.

References

[1] R. Munir, "Teori Bilangan (Bagian 1)," Informatika STEI ITB, Bandung, Indonesia. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/15-Teori-Bilangan-Bagian1-2024.pdf. [Accessed: Jun. 7, 2025].

[2] R. Munir, "Teori Bilangan (Bagian 2)," Informatika STEI ITB, Bandung, Indonesia. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/16-Teori-Bilangan-Bagian2-2024.pdf. [Accessed: Jun. 7, 2025].

[3] R. Munir, "Teori Bilangan (Bagian 3)," Informatika STEI ITB, Bandung, Indonesia. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/17-Teori-Bilangan-Bagian3-2024.pdf. [Accessed: Jun. 7, 2025].

[4] R. Munir, "Pohon (Bagian 1)," Informatika STEI ITB, Bandung, Indonesia. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/23-Pohon-Bag1-2024.pdf. [Accessed: Jun. 7, 2025].

[5] R. Munir, "Pohon (Bagian 2)," Informatika STEI ITB, Bandung, Indonesia. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/24-Pohon-Bag2-2024.pdf. [Accessed: Jun. 7, 2025].

[6] Federal Information Processing Standards Publication 180-4, "Secure Hash Standard," National Institute of Standards and Technology, Gaithersburg, Maryland. [Online]. Available: http://dx.doi.org/10.6028/NIST.FIPS.180-4. [Accessed: Jun. 7, 2025].

[7] GeeksforGeeks, "SHA-256 and SHA-3," GeeksforGeeks. [Online]. Available: https://www.geeksforgeeks.org/computer-networks/sha-256-and-sha-3/. [Accessed: Jun. 7, 2025].

[8] T. Chitty, "The Mathematics of Bitcoin — SHA-256," The Startup. [Online]. Available: https://medium.com/swlh/the-mathematics-of-bitcoin-74ebf6cefbb0. [Accessed: Jun. 7, 2025]

[9] R. Munir, "Kompleksitas Algoritma (Bagian 1)," Informatika STEI ITB, Bandung, Indonesia. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/25-Kompleksitas-Algoritma-Bagian1-2024.pdf. [Accessed: Jun. 8, 2025].

[10] R. Munir, "Kompleksitas Algoritma (Bagian 2)," Informatika STEI ITB, Bandung, Indonesia. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/26-Kompleksitas-Algoritma-Bagian2-2024.pdf. [Accessed: Jun. 8, 2025].

[11] GeeksforGeeks, "Blockchain Merkle Trees," GeeksforGeeks. [Online]. Available: https://www.geeksforgeeks.org/software-engineering/blockchain-merkle-trees/. [Accessed: Jun. 8, 2025].

[12] B. Prahalad, "Merkle Proofs Explained," Crypto-0-nite. [Online]. Available: https://medium.com/crypto-0-nite/merkle-proofs-explained-6dd429623dc5. [Accessed: Jun. 8, 2025].

Statement

I hereby declare that this paper is an original work, written entirely on my own, and does not involve adaptation, translation, or plagiarism of any other individual's work.

Bandung, 19 Juni 2025

Aurelia

Aurelia Jennifer Gunawan - 13524089