A deterministic cryptographic hash function: nar512, 2025

Designing a cryptographic Hash Function: nar512

K V Naresh Karthigeyan, 1CD23IC026

Cambridge Institute of Technology
nareshkarthigeyan.2005@gmail.com

Abstract

In this paper, I propose NAR-512, a hash function that implements a hybrid approach to cryptographic hashing. For the mathematical model to generate deterministic and chaotic pseudo-random numbers, I will use modular arithmetic, bitwise operations like XOR and shifting, pre-defined constants (prime numbers), logistic map for chaos theory. In addition, the logistic map introduces chaotic behavior by iterating a simple non-linear recurrence relation. This is a cryptographic mixing function that resists standard pattern analysis by introducing dynamic iteration counts instead of fixed-round transformations.

1 Introduction

A hash function in the field of cryptography is a mathematical algorithm that takes in a variable input of a string and outputs a fixed-length string of characters, typically a hash value or 'digest'. The significance of cryptographic hash functions like sha-256 is undeniable in the pre-quantum computing era. A hash function H(x) must to have these following properties:

1. Preimage resistance: Given a hash output h, it must be infeasible to find the input that got the output. i.e., the input m cannot be determined. i.e
$$H(m) \neq h.$$
2. Collision Resistance: It must be infeasible to find two distinct inputs a and b such that
$$H(a) \neq H(b).$$
3. Avalanche effect: Small changes in the input should drastically change the output.
$$H(a) \not\sim H(a + \delta)$$

Therefore, let's say, theoretically, if we have the entire works of J K Rowling fed as the input of the hash function H, it must return a fixed-length (typically 256bit) hexadecimal number (or string).

This system is widely used in the industry for storing passwords, and other integral data. Hash functions hold a massive stake in the working of blockchain and cryptography.

2 Examples of existing solutions

Existing cryptographic hash functions include MD5, SHA-1, SHA-2, SHA-256, SHA-3, and others. Among these, SHA-256 is one of the most widely used algorithms. One of its most prominent applications is in Bitcoin, where the SHA-256 hashing algorithm is used for Proof of Work (PoW) to validate blocks in the blockchain.

SHA-256 is also commonly used for securely storing passwords in databases. Due to its preimage resistance, the original input (such as a password) cannot be derived from the hash. The only feasible attack method is a brute-force attack, which has an extremely low probability of success:

$$\frac{1}{2^{256}}$$

This makes SHA-256 highly secure against pre-image attacks under current computational capabilities.

2.1 SHA-256 - Merkle–Damgård construction

Sha-256 stands for Secure Hashing Algorithm 256. As the name suggests, it gives out a 256 bit digest hash on input. It is a widely used cryptographic hash function developed by the NSA as part of the SHA-2 family. It arbitrary length input and produces a 256-bit (32-byte) fixed-length hash. The algorithm follows a Merkle–Damgård construction with a compression function based on bitwise operations, modular additions, and logical functions. It operates on 512-bit blocks and uses 64 rounds of processing, each incorporating a set of constants derived from the fractional parts of the square roots of prime numbers. Given an input message, SHA-256 first pads it to ensure the length is a multiple of 512 bits, then processes it through an iterative hashing structure using an internal state of eight 32-bit words. The final hash is derived from these words, ensuring a high degree of diffusion and security.

Algorithm 1 SHA-256 Hashing Process

Input: Message M of arbitrary length
Output: 256-bit hash H
Step 1: Padding
Append a single '1' bit to M.
Append k '0' bits so that M becomes 448 mod 512.
Append a 64-bit integer representing the original length of M.

Step 2: Initialize Hash Values
Set 8 initial hash values (H_0 to H_7) from the fractional parts of the square roots of the first 8 primes.

Step 3: Process Message in 512-bit Blocks
for *each 512-bit block* **do**
  Break block into 16 32-bit words, extend to 64 words.
  Use 64 round constants derived from prime numbers.
  for *each round i from 0 to 63* **do**
    Compute message schedule and temporary values:
    $\sigma_1 = (e6) \oplus (e11) \oplus (e25)$
    $ch = (e \wedge f) \oplus (\neg e \wedge g)$
    $T1 = h + \sigma_1 + ch + K[i] + W[i]$
    $\sigma_0 = (a2) \oplus (a13) \oplus (a22)$
    $maj = (a \wedge b) \oplus (a \wedge c) \oplus (b \wedge c)$
    $T2 = \sigma_0 + maj$
    Update working variables:
    $h \leftarrow g, g \leftarrow f, f \leftarrow e, e \leftarrow d + T1$
    $d \leftarrow c, c \leftarrow b, b \leftarrow a, a \leftarrow T1 + T2$
  end
  Update hash values: $H_i = H_i + a, \ldots, H_7 = H_7 + h$
end
Step 4: Output the final hash
Concatenate $H_0, H_1, \ldots, H_7$ to form the final 256-bit hash.

2.1.1 Working

The working of Sha-256 is divided into multiple steps **Padding** $\longrightarrow$ **Parsing** $\longrightarrow$ **Compression** $\longrightarrow$ **Final Hash Output** with a few predefined constants.

1. **Padding the Input** – The message is padded so that its length becomes a multiple of 512 bits by appending a '1' bit, followed by zeros, and the original length in 64 bits at the end.

2. **Initializing Hash Values** – The algorithm starts with eight fixed 32-bit constants, derived from the square roots of the first 8 prime numbers.

3. **Processing in 64 Rounds** – Each 512-bit block undergoes 64 transformation rounds, using bitwise operations (XOR, AND, OR, NOT), right shifts, and modular addition.

4. **Updating Eight Working Variables** – In each round, SHA-256 updates eight temporary registers (a, b, c, d, e, f, g, h) to ensure strong diffusion.

5. **Final Hash Output** – After processing all blocks, the modified working variables are combined to produce a 256-bit final hash, ensuring resistance to preimage and collision attacks.

2.2 SHA-3 - Keccak Sponge Construction

SHA-3 (Secure Hash Algorithm 3) is the latest member of the Secure Hash Algorithm family, designed as an alternative to SHA-2. Unlike SHA-256, which follows the Merkle–Damgård construction, SHA-3 is based on the Keccak sponge construction, offering resistance against length extension attacks and enhanced security properties. It takes an arbitrary-length input and produces a fixed-length digest (e.g., 256 bits). The algorithm processes input using a permutation-based sponge function, repeatedly applying a nonlinear transformation to a state matrix. It operates on 1600-bit blocks and performs 24 rounds of computation, utilizing XORs, bitwise rotations, and modular additions to ensure cryptographic strength.

Algorithm 2 SHA-3 Hashing Process

Input: Message M of arbitrary length
Output: 256-bit hash H
Step 1: Padding
Append a domain-specific padding scheme to M to ensure it fits the required block size.

Step 2: Initialize State
Initialize a 1600-bit state matrix (5×5 of 64-bit words) with all bits set to zero.

Step 3: Absorbing Phase
for *each 1088-bit (for SHA3-256) block of* M **do**
  XOR block into the first r bits of the state.
  Apply the Keccak-f[1600] permutation.
end
Step 4: Squeezing Phase
while *more output bits are needed* **do**
  Extract the first r bits of the state as output.
  Apply the Keccak-f[1600] permutation if more bits are required.
end
Step 5: Output the Final Hash
Truncate to 256 bits and return as the final digest.

2.2.1 Working

The working of SHA-3 is fundamentally different from SHA-256, as it follows the Keccak sponge construction instead of the Merkle–Damgård structure. The steps involved are: **Padding** $\longrightarrow$ **Absorbing Phase** $\longrightarrow$ **Permutation** $\longrightarrow$ **Squeezing Phase**

1. **Padding the Input** – The message is padded using a multi-rate padding scheme to ensure that it fits into blocks of the required length.
2. **Absorbing Phase** – The padded input is XORed into the state and then processed through a permutation function iteratively.
3. **Permutation Function** – SHA-3 uses the Keccak-f permutation, which consists of multiple rounds of bitwise operations, rotations, and nonlinear mappings.
4. **Squeezing Phase** – After the input is fully absorbed, the state is repeatedly permuted, and the output bits are extracted to produce the final hash.
5. **Final Hash Output** – The resulting output length depends on the SHA-3 variant (SHA3-224, SHA3-256, SHA3-384, SHA3-512).

3 Proposed hash function: NAR-512

I propose a hash function: NAR-512 that implements a hybrid approach to cryptographic hashing. Taking elements of padding and parsing from sha-256, I attempt to integrate a much more chaotic and determininstic functions to randomize the binary value - to recieve a more resiliant hash.

I also aim for the standard (and only) fixed length output of the hash to be limited to 512 bytes. This is two times as strong as sha-256. On bruteforce attack, the hash function will be twice as strong to break. The number of iterations to break a single hash value would be:

$$\frac{1}{2^{512}} \tag{1}$$

3.1 Mathematical Foundations

The core of the NAR-512 hash function is ensuring strong diffusion, unpredictability, and irreversibility while maintaining efficiency. The function achieves this through a combination of modular arithmetic, bitwise operations, chaotic maps, and number-theoretic techniques.

3.1.1 Modular Arithmetic

Modular arithmetic is fundamental to NAR-512 as it ensures bounded numerical operations while enhancing diffusion. Specifically, modular addition and multiplication introduce non-linearity:

$$(x + y) \mod 2^{32}, \quad (x \times y) \mod 2^{31} - 1$$

The modulus $2^{31} - 1$ is chosen due to its prime nature, contributing to cryptographic strength.

3.1.2 Bitwise Operations and XOR-Based Mixing

Efficient transformations such as XOR, shifts, and rotations are crucial for diffusion. NAR-512 extensively employs these operations.

XOR Operation The XOR operation enhances security by making reversibility difficult and ensuring rapid diffusion:

$$A \oplus B = C$$

- **Non-reversibility:** Given C, recovering A or B is infeasible without additional information.
- **High diffusion:** A single-bit change in the input propagates unpredictably.

Shifts and Rotations Bitwise shifts and rotations further enhance mixing:

$$\text{ROTL}_n(A) = (A \ll n) \vee (A \gg (w - n))$$

$$\text{ROTR}_n(A) = (A \gg n) \vee (A \ll (w - n))$$

where w is the word size (32 bits in this case).

3.1.3 Chaos Theory and Logistic Maps

To introduce chaos into the hash function, NAR-512 utilizes the logistic map:

$$x_{n+1} = r x_n (1 - x_n)$$

where r is carefully selected to be close to 4 (e.g., $r = 3.99$) to maximize chaotic behavior. The logistic map is applied iteratively to intermediate hash values, ensuring extreme sensitivity to input variations.

3.1.4 Irrational Numbers and Pi-Based Constants

To increase unpredictability, NAR-512 extracts constants from the decimal expansion of π. These constants are used in transformations such as:

$$C_i = \lfloor \pi \times 10^{10} \rfloor \mod 2^{32}$$

Additionally, portions of π indexed dynamically based on input data influence state transformations, enhancing entropy.

3.1.5 Fibonacci Sequence Modulo Prime

The Fibonacci sequence introduces additional complexity:

$$F_n = F_{n-1} + F_{n-2}, \quad F_0 = 0, \quad F_1 = 1$$

When reduced modulo a prime number:

$$F_n \mod 2^{31} - 1$$

it exhibits pseudo-random behavior. This sequence is dynamically computed based on input data, injecting further non-linearity into transformations.

3.2 Algorithmic Process

The algorithm of NAR-512 takes a variable input of a string and outputs a 512 bit - or 128 character length hexadecimal "hash" digest.

```
Input: Variable Input String
Output: 512 bit hex digest
```

For this, there are multiple steps involved in the process.

3.2.1 Preprocessing

The preprocessing step, known as **padding**, ensures that the input message aligns with a multiple of 128 bytes, allowing for consistent block-wise processing. The variable-length input message is extended to a fixed size using a structured padding scheme.

For **NAR-512**, the input message M undergoes a padding process similar to SHA-256 but with a key distinction: instead of using conventional zero-padding, additional bits are derived from the digits of the mathematical constant π. This approach enhances unpredictability while maintaining a deterministic structure. The steps are as follows:

1. **Appending a '1' Byte**: A single **1** byte (`0x80`) is appended to signify the end of M.

2. **π-Based Padding**: Instead of filling the remaining space with zeros, each byte of padding is computed using digits of π. Specifically, starting from the **length** l-th digit of π, the next four consecutive digits are summed, then reduced modulo 255 to fit within an 8-bit byte.

   Mathematically, for each padding byte P_i:

$$P_i = \left(\sum_{j=0}^{3} d_{(l+i+j) \mod N} \right) \mod 255 \tag{2}$$

   where d_k represents the k-th digit of π, and N is the total number of available digits in the precomputed π sequence. This process continues until the total message length satisfies:

$$\text{total length} \equiv 448 \mod 512 \tag{3}$$

   ensuring that space remains for the final length encoding.

3. **Appending the Message Length**: The final 64 bits store the binary representation of the original message length (before padding). This ensures that the length can be correctly recovered during processing.

This padding method provides enhanced diffusion and randomness while maintaining deterministic reconstruction, ensuring cryptographic security.

```
Input: Variable length Input String

Output: 2^n length fixed output binary string
        where 2^n > length of input.

BEGIN PROCEDURE
    salt <- 0
    FOR i <- 0 to length - 1:
        output[i] <- 08b(input[i])
        salt <- salt << 5 + salt + input[i]
    END FOR

    output[length] = 0b10000000
    n = 8

    WHILE (length + 9) > 2^n:
        n = n + 1
    END WHILE
```

```
    totalLength = 2^n

    int i = 0
    FOR j <- length + 1 to totalLength - 9:
        int num = 0
        FOR k <- 0 to 3:
            numb += 10 * ((PI[(length + i++ + salt) mod (PI.length - 1)]) - '0')
        END FOR
        output[i] = (numb) mod 255
    END FOR

    len = 064b(length)
    FOR i <- 0 to 7:
        output[totalLength - 8 + i] = (len >> (i * 8)) & 0xFF
    END FOR
END PROCEDURE
```

Example

```
        Input:  abc
        Output: 01100001    01100010    01100011
                10000000    11011100    11010010
                ...
                00000000    00000000    00000011 (512 blocks)
```

3.2.2 Compartmentalization

Following the preprocessing step, the next stage is **compartmentalization**. In this step, the padded message, which consists of 8-bit blocks, is transformed into 32-bit compartments for more efficient processing.

Unlike a direct 1-byte-to-4-byte conversion, each 32-bit compartment is constructed by concatenating four consecutive 8-bit values from the padded message. This transformation reduces the total number of blocks significantly: a 512-byte padded message is condensed into 128 compartments of 4-byte values.

The process follows these steps:

1. Read four consecutive 8-bit values from the padded message.
2. Shift the bits left by 8 positions before appending the next byte.
3. Store the resulting 32-bit block into a new structure for further processing.

```
    PROCEDURE compartmentalize()
        size ← length of padded
        i ← 0
```

```
    WHILE i < size
        block ← 32-bit initialized to 0

        FOR j ← 0 to 3
            block ← block << 8
            block ← block OR padded[i + j]
        END FOR

        ADD block to compartment
        i ← i + 4
    END WHILE
END PROCEDURE
```

This method ensures that the message is efficiently structured for subsequent cryptographic operations, maintaining both integrity and performance.

3.2.3 Logistic Maps, Sigma - M Functions and Pi-Mapping

In this section, we introduce the key transformations used in NAR-512, namely the π-based index transformation, the Logistic Map, and the Sigma-M operations. These functions introduce chaotic behavior, diffusion, and non-linearity into the hashing process.

Bitwise Rotation Functions The macros define left and right rotations within a 32-bit space.

```
Perform Left and Right Bitwise Rotation
INPUT: x (32-bit integer), n (rotation amount), w (bit width)
OUTPUT: Rotated value

BEGIN PROCEDURE ROTL
     ((x << n) OR (x >> (w - n))) AND UINT32_MAX
END PROCEDURE

BEGIN PROCEDURE ROTR
     ((x >> n) OR (x << (w - n))) AND UINT32_MAX
END PROCEDURE
```

Extracting π-Derived Values (π-Mapping) A pseudo-random integer is derived from the digits of π to introduce a deterministic yet complex mapping.

```
INPUT: 32-bit integer index
OUTPUT: Returns an integer in the range [0, 511]
```

```
BEGIN PROCEDURE nPi
    piVal <- 0
    FOR i <- 0 to  30
        piVal <- piVal * 10
        piVal <- piVal + PI[index % PI.size - 2)] - '0')
        index <- index + 1
    END FOR
     piVal mod 511
END PROCEDURE
```

Logistic Map Transformation The Logistic Map introduces chaotic behavior by iterating a simple non-linear recurrence relation.

```
INPUT: 32-bit bitset block, floating-point parameter r
OUTPUT: Transformed bitset block

BEGIN PROCEDURE logisticMap
    x <- block.to_ullong() / UINT64_MAX
    x <- r * x * (1 - x)
    block <- bitset<32>(x * UINT64_MAX)
END PROCEDURE
```

Fibonacci-based Transformation This function computes a modified Fibonacci sequence, accumulating the sum in the reference variable `answer`. It recursively generates Fibonacci numbers starting from the given `prev1` and `prev2` values until the base case is reached. This accumulated sum is later used in the salting process to introduce non-linearity.

```
INPUT:
    - n (integer): The number of Fibonacci iterations.
    - prev1 (integer): The first previous Fibonacci value.
    - prev2 (integer): The second previous Fibonacci value.
    - answer (reference to uint32_t): Stores the accumulated Fibonacci sum.

OUTPUT:
    - Returns an integer (final computed sum of Fibonacci numbers).

BEGIN PROCEDURE fibonacci
    IF n < 3 THEN
         0
    END IF
    current <- prev1 + prev2
    answer <- answer + current
```

```
     fibonacci(n - 1, prev2, current, answer)
END PROCEDURE
```

Sigma Function The Σ function introduces complex diffusion and confusion using bitwise rotations and modular arithmetic.

```
INPUT: 32-bit bitset block
OUTPUT: Modified block with non-linearity

BEGIN PROCEDURE sigma
    piVal <- nPi(block.to_ulong())
    block <- block XOR ROTL(block.to_ulong(), 17, 32)
    block <- block XOR ROTR(block.to_ulong(), 11, 32)
    block <- block XOR ((block.to_ulong() * piVal) + 3149757) mod INT32_MAX
    block <- block OR ((block.to_ulong() XOR (piVal >> 2) * 37) mod INT32_MAX)
    block <- block XOR (piVal << 3)
END PROCEDURE
```

M Function The M function combines Logistic Map transformation with bitwise operations and modular arithmetic.

```
INPUT: 32-bit bitset block
OUTPUT: Modified block with enhanced randomness

BEGIN PROCEDURE M
    piVal <- nPi(block.to_ulong())
    num <- block.to_ulong()
    logisticMap(block, abs(piVal / 3.14))
    block <- block XOR (block << 17)
    block <- block XOR (block >> 19)
    block <- block OR ROTL(num * piVal, abs(piVal - 15), 32)
END PROCEDURE
```

These functions play a crucial role in ensuring diffusion and non-linearity in the hashing process.

3.2.4 Salting

Salting in NAR-512 is the cornerstone of its resilience against precomputed attacks. Unlike conventional hashing mechanisms, which often rely on fixed or predictable salts, NAR-512 employs an evolving salt mechanism derived from chaotic and recursive transformations. This ensures that even minimal changes in input result in vastly different internal states, enhancing unpredictability.

Salting Mechanism The salting process operates on the 32-bit compartments established in the previous step. It leverages a combination of Logistic Map, Fibonacci Perturbation, Sigma-M Diffusion to ensure irreversibile computations and increase avalanche effect.

The procedure is as follows:

```
INPUT: compartment[] (array of 32-bit bitsets)
OUTPUT: Salted compartments with high diffusion

BEGIN PROCEDURE salt()
    FOR i ← 0 to size - 1:
        logisticMap(compartment[i], 3.99)  // Chaotic perturbation
        fb ← 0
        fibonacci(compartment[i].to_ulong() mod 511, 0, 1, fb)
        compartment[(i + 1) mod size] ←
            compartment[i] XOR sigma(compartment[(i + 2) mod size])
            OR fb XOR M(compartment[(i + 7) mod size])
    END FOR
END PROCEDURE
```

Logistic Chaos Integration Each compartment is first passed through a Logistic Map with a near-critical value of $r = 3.99$, ensuring sensitive dependence on initial conditions:

$$x_{n+1} = r \cdot x_n \cdot (1 - x_n) \tag{4}$$

where x_n represents the normalized 32-bit value of the compartment. This introduces chaotic oscillations, ensuring unique state propagation even for slightly different inputs.

Fibonacci Perturbation A recursively computed Fibonacci number based on the compartment's index mod 511 is incorporated into the bitwise state. The Fibonacci sum disrupts linear correlations, reinforcing diffusion.

$$F(n) = F(n-1) + F(n-2), \quad F(0) = 0, \quad F(1) = 1 \tag{5}$$

This value is then XORed into the compartmentalized structure.

Sigma-M Transformation To maximize entropy, the modified compartment is further transformed using the Σ and M functions:

$$C_{i+1} = C_i \oplus \Sigma(C_{i+2}) \vee F(n) \oplus M(C_{i+7}) \tag{6}$$

where:

- $\Sigma(x)$ introduces controlled bitwise rotations and modular transformations.

- $M(x)$ applies Logistic Map perturbations combined with chaotic shifts.

Avalanche and Entropy Analysis The combination of these transformations ensures:

- **High Avalanche Effect**: A single-bit change in input propagates exponentially across compartments.
- **Cryptographic Diffusion**: Any alteration results in complete state transformation within a few iterations.
- **Non-reversibility**: Due to non-linear mapping and modular arithmetic, reconstructing prior states from outputs is computationally infeasible.

This salting strategy solidifies NAR-512's resistance against common cryptanalytic techniques while maintaining deterministic integrity.

Salt Function Results On testing the salt function with two very similar input strings to test it's avalanche effect. Here were my observations:

```
Test Case 1
Input String: "Hello there, My name is Naresh Karthigeyan"
Before Salting:
48656c6c6f2074686572652c204d79206e616d65206973204e6172657368204b617274...
After Salting:
eec7efa7e5a416cde81d6206ed61b290f88b2c32b77d813f71c69ec7fbb17ad258a857...

Test Case 2
Input String: "Hello there, My name is Naresh Karthigeyan!"
Before Salting:
48656c6c6f2074686572652c204d79206e616d65206973204e6172657368204b617274...
After Salting:
ee6bc371e5a416cde81d60e7fc614d875ccfb2b7ce951882f53ce26ef56dc7b639ae12...
```

The salting function effectively obfuscates the original structure of the message, ensuring that the direct hash of the similar input produces significantly different outputs. However, on looking closer:

1. The start of the salted output still exhibits similarities before the padding phase (notably, before Pi digits are introduced). Everything is same.
2. After salting, although both the salts are essentially different, even at the starting stages, some parts of the digest are still same.

```
        S1: eec7efa7e5a416cde81d6206ed61b290f88b2c32b77d813f71c69...
        S2: ee6bc371e5a416cde81d60e7fc614d875ccfb2b7ce951882f53ce...
   SIMILAR: ee      e5a416cde81d6     61
```

3. I wanted to investigate it further. So I implemented a simple program in python to find the common sub-string. To find if avalance effect actually works.

```
def longest_common_substring(s1, s2):
    m, n = len(s1), len(s2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    longest = 0
    longest_end = 0

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if s1[i - 1] == s2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1
                if dp[i][j] > longest:
                    longest = dp[i][j]
                    longest_end = i
            else:
                dp[i][j] = 0

    return s1[longest_end - longest: longest_end]

OUTPUT:
 Longest common substring: e5a416cde81d6
 Length of common substring: 13
```

This could also be a result of sampling bias. To better visualize the validity of this issue, I took foward to testing the length of common sub-string for variable levels of input. Here is what my readings were:

Experimental Data I conducted tests by hashing an input string and its modified version (with a single additional character "."). The table below summarizes the findings:

The graph below illustrates how the length of the common substring increases as the input size grows.

As the graph image suggests, the salt function demonstrates a strong avalanche effect for small inputs, longer inputs begin to exhibit more common and longer substrings in their hashes. This means the pseudo-randomization is not completely working. Thus, this requires a much more sophisticated solution.

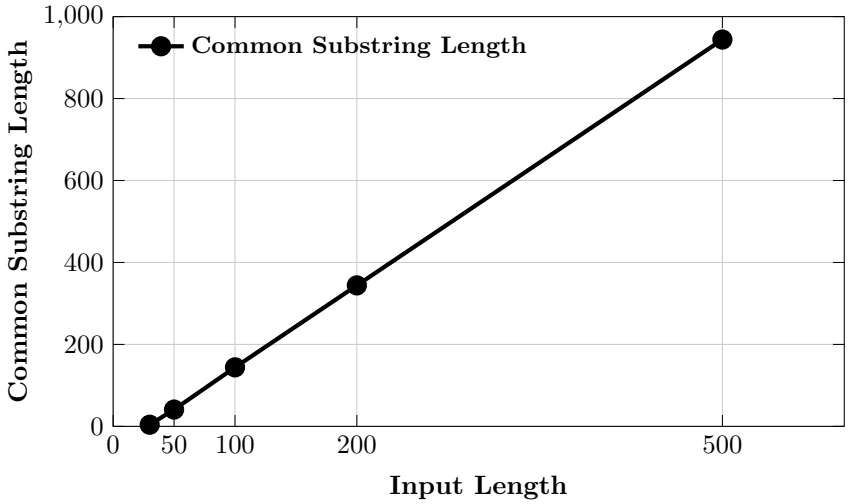

Figure 1: **Avalanche Effect Analysis**

3.2.5 Compression and State Register calculation

Novelty of Nar512 - Dynamic State Registers SHA-256 uses state registers - they have the same constant value every time: The irrational parts of the roots of the first eight prime numbers. Following NAR-512, I will also use state registers. However, instead of a static value, I will dynamically generate the state registers depending on the input. This, therefore, is the novelty of NAR-512. Dynamic State Registers using Compartment Compression. Thus, this approach will improve the lack of avalanche effect caused in salting and prevent pre-image attacks. Due to it's dynamic nature - it would stand against cryptographic attacks in a, I hope, much more resiliant manner.

Compression Process The compression phase in NAR-512 is designed to generate highly entropic intermediary states, ensuring a strong diffusion effect before finalizing the state registers. This process consists of three key stages:

- Compartments to Temporary state 1. Compression by 2
- Temporary state 1 to Temporary state 2. Compression by 4
- State Register Update: Iterative Permutation and Finalization

Step 1: Compartment to Temporary state 1 The salted input (compartments array) is in 32-bit segments, as we already know. The size of this array is 128 or 256 or 512... depending on the size of the input string (refer Padding

Input Length	Sample Hash (Truncated)	Common Substring Length
3	5fd94	5
10	fd70a	5
30	e7fb	4
50	bad55f9ffb...	41
100	fc8987e0fc...	144
200	dc854003b4...	344
500	f4fec52870...	944

Table 1: Common Substring Lengths in Hashes of Similar Inputs

and Compartmentalization). The aim of this step is to compress the compartment array to a temporary state array to half it's size. We do that by iterating every second element of compartment and performing pseudo-random operations with it's neighbors and adding that to our `temp1` array. The steps involved are;

1. A Fibonacci sequence operation is applied to every second element in `compartment`, where the seed is derived from XOR of its next and third neighbor's modulo with 255 which we store in a variable `fb`.

2. Compute the XOR between the current element `compartment[i]` and its third neighbor `compartment[(i+3) % size]` and apply a logistic map transformation to the result using a bifurcation parameter $r = 3.97$, ensuring chaotic behavior. Append the transformed value in `temp1`.

This results in an intermediate vector, `temp1`, which contains a transformed version of the original input with injected chaotic behavior, half the size of compartment array.

```
INPUT: compartment[] (array of 32-bit bitsets)
OUTPUT: temp1[] (compressed array with enhanced randomness)

BEGIN PROCEDURE stateCompression()

    INITIALIZE empty temp1[]
    FOR i ← 0 to compartment.size() - 1, step 2:
        fb ← 0
        fibonacci(compartment[i].to_ulong() mod 255, 0, 1, fb)
        x' ← compartment[i].to_ulong() XOR compartment[(i + 3)
                            mod compartment.size()].to_ulong()
        temp1.push_back(logisticMap(x', 3.97))
    END FOR
```

Step 2: Temporary State 1 to Temporary State 2 Once I obtain the `temp1` array, the next step is to further compress and transform it into `temp2`. The purpose of this step is to reduce the size to 4 times the compartment array. We do this by compressing the temporary state 1 to half it's size. Similar to the previous step, I iterate every second element of `temp1` and apply multiple pseudo-random transformations before storing the result in `temp2`. The steps involved are:

1. A Fibonacci sequence operation is applied to every second element in `temp1`, where the seed is derived from the modulo 255 of its neighboring elements at index $(i+1)$ and $(i+3)$. The resulting Fibonacci sequence value is stored in the variable `fb`.
2. The pi value is then derived using `nPi(fb)`, where `fb` acts as the starting index for the function.
3. A bitwise XOR operation is performed between the current element `temp1[i]` and its second neighbor `temp1[i+2]`.
4. The result is then OR'd with the integer value of `fb` divided by the pi value to introduce additional diffusion.
5. The final transformed value is stored in `temp2`.

This step ensures that each element in `temp2` is a chaotic mix of multiple values from `temp1`, enhancing diffusion and breaking linearity, making the state more resistant to cryptographic attacks.

```
CONTINUE PROCEDURE stateCompression()
    INITIALIZE empty temp2[]
    FOR i ← 0 to temp1.size() - 1, step 2:
        fb ← 0
        fibonacci(temp1[(i + 1) mod temp1.size()].to_ulong()
                        mod 255,
                  temp1[(i + 3) mod temp1.size()].to_ulong()
                        mod 255, 1, fb)
        piVal ← nPi(fb)
        block ← temp1[i].to_ulong()
                XOR temp1[(i + 2) mod temp1.size()].to_ulong()
                OR (fb / piVal)
        temp2.push_back(block)
    END FOR
```

The resulting values are stored in `temp2`, forming a compressed, high-entropy representation of the input. This prepares the data for state register updates, ensuring that the input-dependent dynamic states have sufficient randomness before they are iteratively modified in the final stage.

Step 3: Temporary State 2 to Final State Register At this stage, we take `temp2` and iteratively mix its values into the `states` array. Unlike traditional hash functions like SHA-256, which have a fixed number of iterations (e.g., 64 rounds), the number of transformations in this step is dynamically determined by the input itself. This ensures that each input results in a unique and non-uniform number of mixing rounds, enhancing security and unpredictability.

The steps involved in this transformation are:

1. Dynamic Iteration Count Calculation: - The number of iterations for each state update is determined by applying `nPi` on the modulo of the corresponding `temp2` element with 255. - The result is then taken modulo 277, ensuring a large but controlled range of iterations. - To prevent extremely low iteration counts that may weaken diffusion, a minimum threshold of 64 is enforced.

2. Iterative State Mixing: - For each state index, the transformation runs for the dynamically computed number of iterations. - In each iteration: - A chaotic mixing operation is performed using XOR, sigma transformation, modular perturbation (`M()`), and a logistic map. - The logistic map uses a dynamically chosen bifurcation parameter, switching between 3.99 and another close chaotic value based on the output of `nPi(17)`. - The resulting `block` is then further mixed with a Fibonacci sequence transformation, ensuring nonlinearity and additional entropy injection.

3. State Register Update: - The computed `block` is XORed with the Fibonacci sequence value. - The final transformed value is stored in `states[i]`, completing the iterative transformation.

This step ensures that the state register undergoes a highly variable number of rounds, adapting dynamically to the input properties.

```
CONTINUE PROCEDURE stateCompression()
    FOR i ← 0 to states.size() - 1:
        iterations ← nPi(temp2[i mod states.size()].to_ulong() mod 255)
                     mod 277
        iterations ← max(iterations, 64)

        max ← temp2.size()
        FOR j ← 0 to iterations - 1:
            block ← temp2[(i + 1) mod max]
                    XOR sigma(temp2[(i + 7) mod max])
                    OR M(temp2[(i + 11) mod max])
                    XOR logisticMap(temp2[(i + 5) mod max],
                    ((nPi(17) mod 4) / 4 == 0 ?
                                    3.99 : (nPi(17) mod 4) / 4))
```

```
            fb ← temp2[i].to_ulong()
            fibonacci(fb mod 511, 0, 1, fb)

            block ← block.to_ulong() XOR fb
            states[i] ← block
        END FOR
    END FOR
END PROCEDURE
```

The result is a cryptographic mixing function that resists standard pattern analysis by introducing dynamic iteration counts instead of fixed-round transformations.

This ensures that the state register update is input-dependent, with varying levels of diffusion and chaotic transformations. The self-adjusting iteration count makes it significantly harder for an attacker to predict or reverse-engineer the transformation process.

3.2.6 Final Hash

The final hash is derived from the state registers by converting them into a 512-bit digest using the `toHash()` function. This function ensures that the output maintains a uniform structure while preserving the high-entropy, chaotic transformations applied in the previous steps.

To generate the final 512-bit hash, each 32-bit segment from the state registers is converted into an 8-character hexadecimal representation and concatenated to form the complete hash string.

Hexadecimal Conversion Function:

```
string toHex(vector<bitset<32>> vec)
    ostringstream oss;
    for (auto i : vec)
        oss << hex << setw(8) << setfill('0') << i.to_ulong();

    return oss.str();
```

This function ensures that each segment is padded to 8 characters, maintaining a consistent format across different inputs.

Example Hash Computation Output (512-bit hash):

```
Input: abc
Output:
e9ba56716b3e2c3d3e9990e361bf650519da2dc96f0a4a5313b66cdc5e45a041
8d011ce1472a84273ec18325dac83d82bfc61723e11957144cc32820e3672362
```

This final digest is the output of the entire hashing process, encapsulating all the transformations, chaotic iterations, and pseudo-random operations performed throughout the algorithm.

3.3 Implementation

During the entire process of creating this algorithm, I developed it directly in code. This might be unconventional—coding while designing the logic—but it allowed me to iteratively refine the approach and test ideas in real-time. I chose to implement this algorithm in C++.

3.3.1 Why C++?

C++ was my language of choice for several reasons:

- **Performance:** Hash functions require efficient bitwise operations, modular arithmetic, and memory management. C++ offers low-level control over these aspects, ensuring optimal performance.
- **Bitwise Manipulation:** The language provides direct support for bitwise operations, shifts, and rotations, which are crucial for cryptographic diffusion.
- **Optimized Compilation:** C++ compilers (e.g., GCC, Clang) enable aggressive optimizations, improving execution speed without compromising security.
- **Deterministic Execution:** Unlike higher-level languages that introduce unpredictable behavior due to garbage collection or runtime overhead, C++ allows fine-tuned, predictable execution—essential for cryptographic implementations.
- **Personal Bias:** It's my favourite programming language. In fact, I think it is the best programming language in history. I like it very much that I usually find excuses to use it in any way possible.

Moreover, C++ gives me full control over every computation step, ensuring that no unintended side effects arise from higher-level abstractions. This control is especially important in cryptographic functions, where unintended behavior (such as timing leaks) can introduce vulnerabilities.

3.3.2 The Code

```
#include <iostream>
#include <iomanip>
#include <bitset>
```

```
#include <math.h>
#include <sstream>

#define ROTL(x, n, w) (((x) << (n)) | ((x) >> ((w) - (n))))
                                                & UINT32_MAX
#define ROTR(x, n, w) (((x) >> (n)) | ((x) << ((w) - (n))))
                                                & UINT32_MAX

using namespace std;
bitset<32> logisticMap(bitset<32> block, float r);

vector<bitset<32>> states = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                             0, 0, 0, 0};
vector<bitset<8>> padded;
vector<bitset<32>> compartment;

const string PI = "14159265358979323846264338327950288419716939937510582097
49445923078164062862089986280348253421170679821480865132823066470938446095
05822317253594081284811174502841027019385211055596446229489549303819644288 1
09756659334461284756482337867831652712019091456485669234603486104543266482 1
33936072602491412737245870066063155881748815209209628292540917153643678925 9
03600113305305488204665213841469519415116094330572703657595919530921861173 8
19326117931051185480744623799627495673518857527248912279381830119491298336 7
336244"; //First 512 digits of Pi.

int nPi(uint32_t index)
{
    int piVal = 0;
    for (int i = 0; i < 31; i++)
    {   piVal *= 10;
        piVal += (PI[index++ % PI.size() - 2] - '0');
    }
    return abs(piVal % 511);
}

void padding(string input){
    padded.clear();
    int len = input.size();
    unsigned int salt = 0;
    for (int i = 0; i < len; i++)
        padded.push_back(bitset<8> (input[i]));
        salt += (salt << 5) + salt + input[i];

    padded.push_back(bitset<8>(0b10000000));
```

```
    int soFar = input.size();
    int n = 9;
    int targetSize = 0;

    while(targetSize < soFar){
        targetSize = pow(2, n++);
    }

    int lenToFinish = targetSize - 8;
    int i = 0;
    while (padded.size() < targetSize - 8) {
        int numb = 0;
        for (int j = 0; j < 4; j++)
        numb += 10 * ((PI[(len + i++) % (PI.size() - 1)]) - '0');
        padded.push_back(bitset<8>((numb) % 255));
    }

    uint64_t bitLen = len;

    for (int i = 7; i >= 0; i--) {
        padded.push_back(bitset<8>((bitLen >> (i * 8)) & 0xFF));
    }
}

void compartmentalize(){
    compartment.clear();
    int size = padded.size();
    for(int i = 0; i < size; i += 4){
        bitset<32> block {0};
        for(int j = 0; j < 4; j++){
            block <<= 8;
            block |= padded[i + j].to_ulong();
        }
    }
}

bitset<32> logisticMap(bitset<32> block, float r){
    double x = (double) block.to_ullong() / (double) UINT64_MAX;
    x = r * x * (1.0 - x);
    block = bitset<32>(static_cast<uint64_t>(x * UINT64_MAX));
    return block;
}
```

```
bitset<32> sigma(bitset<32> block){
    int piVal = nPi(block.to_ulong());
    block ^= ROTL(block.to_ulong(), 17, 32);
    block ^= ROTR(block.to_ulong(), 11, 32);
    block ^= ((block.to_ulong() * piVal) + 3149757) % INT32_MAX;
    block |= (block.to_ulong() ^ (piVal >> 2) * 37) % INT32_MAX;
    block ^= (piVal << 3);
    return block;
}

bitset<32> M(bitset<32> block){
    int piVal = nPi(block.to_ulong());
    uint32_t num = block.to_ulong();
    block = logisticMap(block, abs(piVal / 3.14));
    block ^= block << 17;
    block ^= block >> 19;
    block |= ROTL(num * piVal, abs(piVal - 15), 32);
    return block;
}

uint32_t fibonacci(uint32_t n, int prev1, int prev2, uint32_t &answer){
    if(n < 3) return 0;
    int current = prev1 + prev2;
    answer += current;
    return fibonacci(n - 1, prev2, current, answer);
}

void salting(){
    int size = compartment.size();
    for (int i = 0; i < size; i++){
        compartment[i] = logisticMap(compartment[i], 3.99);
        uint32_t fb;
        fibonacci(compartment[i].to_ulong() % 511, 0, 1, fb);
        compartment[(i + 1) % size] =
            compartment[i] ^ sigma(compartment[(i + 2) % size]) |
            bitset<32>(fb) ^ M(compartment[(i + 7) % size]);
    }
}

void stateCompression(){
    vector<bitset<32>> temp1;
    for(int i = 0; i < compartment.size(); i += 2){
        uint32_t fb = 0;
        fibonacci(compartment[i].to_ulong() % INT8_MAX, 0, 1, fb);
```

```
        temp1.push_back(logisticMap(compartment[i].to_ulong() ^
            compartment[(i + 3) % compartment.size()].to_ulong(), 3.97));
    }

    vector<bitset<32>> temp2;
    for(int i = 0; i < temp1.size(); i+= 2){
        uint32_t fb = 0;
        fibonacci(temp1[(i + 1) % temp1.size()].to_ulong()
                % INT8_MAX,
            (int) temp1[(i + 3) % temp1.size()].to_ulong()
                % INT8_MAX, 1, fb);
        uint32_t piVal = nPi(fb);
        bitset<32> block ((uint32_t) temp1[i].to_ulong() ^
            temp1[(i + 2) % temp1.size()].to_ulong()
            | (uint32_t) (fb / piVal));
        temp2.push_back(block);
    }

    for(int i = 0; i < states.size(); i++){
        int iterations = (nPi((int) temp2[i % states.size()].to_ulong()
                % INT8_MAX) % 277);
        iterations = iterations < 64 ? 64 : iterations;
        int max = temp2.size();
        for(int j = 0; j < iterations; j++){
            bitset<32> block = temp2[(i + 1) % max]
                ^ sigma(temp2[(i + 7) % max]) |
                M(temp2[(i + 11) % max]) ^
                logisticMap(temp2[(i + 5) % max],
                ((nPi(17) % 4) / 4 == 0 ? 3.99 : (nPi(17) % 4) / 4));
            uint32_t fb = temp2[i].to_ulong();
            fibonacci(fb % 511, 0, 1, fb);
            block = block.to_ulong() ^ fb;
            states[i] = block;
        }
    }
}

string toHex (vector<bitset<32>> vec){
    ostringstream oss;
    for (auto i: vec){
        oss << hex  << setw(8) << setfill('0') << i.to_ulong();
    }
    return oss.str();
}
```

```
int main(void){
    string str = "";
    getline(cin, str);
    padding(str);
    compartmentalize();
    salting();
    stateCompression();
    string output  = toHex(states);
    cout << output << endl;
}
```

3.3.3 Tests and Optimization

In the code given above, I used `include <bitset>` library. It contained bitsets, which allowed me to easily visualize the 0's and 1's. However, I was worried with it's efficiency, as in the overall implementation, the visualization would not hold much stake. So, to completely prove my doubts. I ran a test on nar512() for average times (for each string) and total times taken to execute 10000 strings of variable length.

The following table summarizes the average execution time per string and the total execution time for hashing 1,000 strings of different lengths using the initial bitset32 implementation:

Char Length	Average Time (sec)	Total Time (sec)
1	0.01216	12.16328
10	0.01238	12.38378
100	0.01242	12.42324
250	0.01252	12.51921
500	0.01241	12.41084
1024	0.01321	13.20876
5000	0.02473	24.73420
10000	0.03807	38.06661

Table 2: Execution Time Analysis of `nar512()` Using Bitset

0.01216 seconds per execution was certainly high for a cryptographic hash aglorithm. This needed more optimization - as a regular SHA-256 takes about 0.000009 seconds. Our current aglorithm is 13333.3% slower!

Using uint64_t instead of bitsets Since bitsets seem to have so many unnecassary and redundant conversions for the sake of "visualizsation" which isn't

even used in the final implementation - I replaced every bitset32 with uint32_t and every bitset8 with uint8_t.

Converting String Pi to A Vector for O(1) lookup The current PI is a huge string of characters of 512 digits of pi. This could prove to be a huge space in memory, and despite index-based look ups, the further character to int conversions, that too, repeatedly, will take a toll. Thus, my first approach was to convert the string into a unordered map representation. On further research, I found out that vectorint also has the same lookup if it's index based. O(1). So I converted the digits of Pi into a piMap (containing multiple digits at once) to increase efficiency. These will be initialized and stored at the start of the program.

```
vector<int> piMap = 341, 296, 232, 252, 324, 90, 292, 315,
458, 381, 95, 416, 484, 295, 292, 159, 497, 254, 463, 33,
271, 163, 508, 126, 280, 289, 243, 260, 424, 132, 277, 449,
276, 176, 348, 311, 60, 74, 155, 420, 0, 166, 430, 124, 411,
24, 128, 202, 334, 358, 21, 323, 166, 39, 390, 297, 395, 333,
...
170, 81, 297, 173, 160, 43, 154, 66, 140, 231, 263, 509, 76, 378,
161, 494, 299, 200, 159, 491, 255, 146, 324, 456, 353;
```

Now with those optimizations in place, I ran the same test again. Here were the results:

Char Length	Average Time (sec)	Total Time (sec)
1	0.00757	7.56610
10	0.00714	7.14433
100	0.00724	7.23530
250	0.00720	7.20347
500	0.00724	7.23732
1024	0.00759	7.58831
5000	0.01457	14.57167
10000	0.02244	22.43553

Table 3: Execution Time Readings after vector PiMapping

‘

3.3.4 Further Optimizations

Even though I doubled the speed. I felt there were still improvements to be made.

Resizing vector instead of Clearing Before padding and compartmentalization, I performed `padded.clear()` and `compartment.clear()`. This, thus, proved

to be pretty expensive, because I'd eventually have to `.push_back` every new block. Thus, initializing the arrays with empty zeros helped improve the speed.

Removing Recursion in Fibonacci Although recusrion is a cool concept - in reality - it took up more resources and stack that I would have hoped for. And in low level implementations, it would not have been possible to reserve huge amounts of stack space, hence I converted fibonacci series from a recursive function to an iterative function.

```
uint32_t fibonacci(uint32_t n)
if (n < 2) return n;
uint32_t a = 0, b = 1;
for (uint32_t i = 2; i <= n; i++)
    uint32_t temp = a + b;
    a = b;
    b = temp;

return b;
```

After all this, I ran the test once again, and the results astonished me.

Char Length	Average Time (sec)	Total Time (sec)
1	0.00104	1.04006
10	0.00101	1.00697
100	0.00102	1.01884
250	0.00106	1.06011
500	0.00108	1.08337
1024	0.00117	1.16975
5000	0.00225	2.24940
10000	0.00356	3.55658

Table 4: Execution Time Measurements after vector and recursive overwritess

According to this readings, this means I have achieved over 90% improvement in each case! Which is a massive improvement.

3.4 Go Implementation

Go, often referred to as Golang, is an open-source programming language developed by Google in 2007 and publicly released in 2009. It was designed to address the challenges of modern software development by combining the performance and

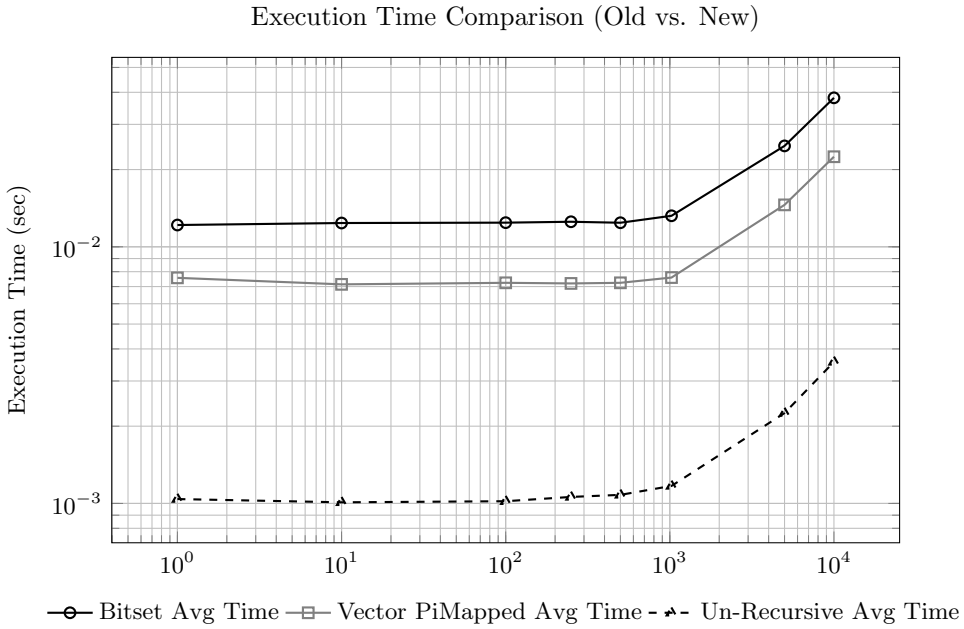

Figure 2: Execution Time Comparison in C++

safety of a statically typed, compiled language with the simplicity and productivity of a dynamically typed language. I wanted to implement nar512 because of two distinc and obvious reasons: **go-routines paralellism**. This can make the algorithm extremely fast.

3.4.1 Why Go (Golang)?

After initially developing the algorithm in C++, I decided to implement it in Go for several compelling reasons:

- **Built-in Concurrency:** Go's native support for goroutines and channels allowed efficient parallelization of hashing operations, which can significantly improve throughput on modern multi-core processors. C++ also had paralellization, however I still prefered the simplicity of implementation in Go, as the main focus was the algorithm and not the language.
- **Memory Safety:** Unlike C++, Go had garbage collection, therefore, it was easier for me to focus on operations without worrying if it messed up the memory.

- **Fast Compilation and Deployment:** Go compiles quickly to a single static binary, simplifying deployment and distribution across different platforms without external dependencies.
- **Cross-Platform Consistency:** Go's runtime ensures consistent behavior across operating systems and architectures, which is critical for cryptographic algorithms that require deterministic outputs.
- **Personal Growth:** I used this project as an excuse to learn Gl. Exploring Go allows me to broaden my programming expertise and leverage modern language features that complement my existing C++ knowledge.

3.4.2 Efficiency Decisions in Golang

High-Level Optimization Areas Instead of re-coumputing the same values multiple times, it was better caching repeated values for algortihms such as PiVal() and fibonacii(). Memory allocations for arrays (now slices) also seemed to be massive optimizations.

1. **Memory Allocations (Reduce GC Pressure)**

   - Preallocate all slices (e.g., `padded`, `compartment`, `temp1`, `temp2`, `states`) wherever possible using `make()` instead of growing them with `append`. This avoids multiple allocations and copies.

```
padded := make([]uint8, 0, totalLen)
```

2. **Avoid Excessive Modulus**

   - The modulus operator (`%`) is slower than bitmasking. If modulus is a power of 2 (e.g., `% 512`, `% 16`), use bitwise AND:

$$x\%512 \rightarrow x\&511$$

3. **Fibonacci Optimization using caching**

   - Replace recursive-like loops with an iterative and memoized version:

```
fibCache := make([]uint32, 512)
fibCache[0], fibCache[1] = 0, 1
for i := 2; i < 512; i++ {
    fibCache[i] = fibCache[i-1] + fibCache[i-2]
```

```
    }
    // Usage:
    fb := fibCache[x % 511]
```

3.4.3 The code (in Go)

```
package main
import (
 "fmt"
 "math"
 "math/bits"
 "time"
)

func ROTL(x uint32, n int) uint32 {
 return bits.RotateLeft32(x, n);
}

func ROTR(x uint32, n int) uint32 {
 return bits.RotateLeft32(x, -n);
}

var PI = []int{
 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3,
    2, 3, 8, 4, 6, 2, 6, 4, 3, 3, 8, 3, 2, 7, 9,
 5, 0, 2, 8, 8, 4, 1, 9, 7, 1, 6, 9, 3, 9, 9,
    3, 7, 5, 1, 0, 5, 8, 2, 0, 9, 7, 4, 9, 4, 4,
 5, 9, 2, 3, 0, 7, 8, 1, 6, 4, 0, 6, 2, 8, 6,
    2, 0, 8, 9, 9, 8, 6, 2, 8, 0, 3, 4, 8, 2, 5,
 3, 4, 2, 1, 1, 7, 0, 6, 7, 9, 8, 2, 1, 4, 8,
    0, 8, 6, 5, 1, 3, 2, 8, 2, 3, 0, 6, 6, 4, 7,
 0, 9, 3, 8, 4, 4, 6, 0, 9, 5, 5, 0, 5, 8, 2,
    2, 3, 1, 7, 2, 5, 3, 5, 9, 4, 0, 8, 1, 2, 8,
 4, 8, 1, 1, 1, 7, 4, 5, 0, 2, 8, 4, 1, 0, 2,
    7, 0, 1, 9, 3, 8, 5, 2, 1, 1, 0, 5, 5, 5, 9,
 6, 4, 4, 6, 2, 2, 9, 4, 8, 9, 5, 4, 9, 3, 0,
    3, 8, 1, 9, 6, 4, 4, 2, 8, 8, 1, 0, 9, 7, 5,
 6, 6, 5, 9, 3, 3, 4, 4, 6, 1, 2, 8, 4, 7, 5,
    6, 4, 8, 2, 3, 3, 7, 8, 6, 7, 8, 3, 1, 6, 5,
 2, 7, 1, 2, 0, 1, 9, 0, 9, 1, 4, 5, 6, 4, 8,
    5, 6, 6, 9, 2, 3, 4, 6, 0, 3, 4, 8, 6, 1, 0,
 4, 5, 4, 3, 2, 6, 6, 4, 8, 2, 1, 3, 3, 9, 3,
```

```
    6, 0, 7, 2, 6, 0, 2, 4, 9, 1, 4, 1, 2, 7, 3,
 7, 2, 4, 5, 8, 7, 0, 0, 6, 6, 0, 6, 3, 1, 5,
    5, 8, 8, 1, 7, 4, 8, 8, 1, 5, 2, 0, 9, 2, 0,
 9, 6, 2, 8, 2, 9, 2, 5, 4, 0, 9, 1, 7, 1, 5,
    3, 6, 4, 3, 6, 7, 8, 9, 2, 5, 9, 0, 3, 6, 0,
 0, 1, 1, 3, 3, 0, 5, 3, 0, 5, 4, 8, 8, 2, 0,
    4, 6, 6, 5, 2, 1, 3, 8, 4, 1, 4, 6, 9, 5, 1,
 9, 4, 1, 5, 1, 1, 6, 0, 9, 4, 3, 3, 0, 5, 7,
    2, 7, 0, 3, 6, 5, 7, 5, 9, 5, 9, 1, 9, 5, 3,
 0, 9, 2, 1, 8, 6, 1, 1, 7, 3, 8, 1, 9, 3, 2,
    6, 1, 1, 7, 9, 3, 1, 0, 5, 1, 1, 8, 5, 4, 8,
 0, 7, 4, 4, 6, 2, 3, 7, 9, 9, 6, 2, 7, 4, 9,
    5, 6, 7, 3, 5, 1, 8, 8, 5, 7, 5, 2, 7, 2, 4,
 8, 9, 1, 2, 2, 7, 9, 3, 8, 1, 8, 3, 0, 1, 1,
    9, 4, 9, 1, 2, 9, 8, 3, 3, 6, 7, 3, 6, 2, 4,
 4,
}

var fibCache [512]uint32

func init() {
 fibCache[0], fibCache[1] = 0, 1
 for i := 2; i < 512; i++ {
  fibCache[i] = fibCache[i-1] + fibCache[i-2]
 }
}

func nPi(index uint32) int {
 piVal := 0

 for i:= 0; i < 31; i++ {
  pos := (int(index) + i) % len(PI)
  digit := int(PI[pos] - '0');
  piVal = piVal * 10 + digit;
 }

 return int(math.Abs(float64(piVal % 511)))
}

func padding(input string) []uint8 {
 soFar := len(input)
 totalLen := ((soFar + 511) / 512) * 512
 padded := make([]uint8, totalLen)
```

```
 var salt uint32 = 0

 for i := 0; i < soFar; i++ {
  ch := input[i]
  padded[i] = ch
  salt ^= uint32(ch)
  salt *= 16777619
 }

 if soFar < totalLen {
  padded[soFar] = uint8(salt)
 }

 i := 0
 var h uint32 = salt
 for j := soFar + 1; j < totalLen - 4; {
  numb := 0
  for k := 0; k < 4 && j < totalLen; k++ {
   index := (soFar + i + int(salt)) % len(PI)
   if index < 0
    index += len(PI)

   numb += 10 * PI[index]
   i++
  }
  h ^= uint32(uint32(padded[(j-1)%soFar]) ^ uint32(numb))
  h *= 0x5bd1e995
  padded[j] = uint8((h >> 17) ^ (h & 0xFF))
  j++
 }
 return padded
}

func compartmentize(padded []uint8) []uint32 {
 blocks := make([]uint32, len(padded)/4)
 for i := 0; i < len(padded); i += 4 {
  blocks[i/4] = uint32(padded[i])<<24
        | uint32(padded[i+1])<<16 | uint32(padded[i+2])<<8 | uint32(padded[i+3])
 }
 return blocks
}

func logisticMap(block uint32, r float64) uint32 {
 var x float64 = float64(block) / float64(math.MaxUint32)
```

```
 x = r * x * (1.0 - x)
 block = uint32(x * math.MaxUint32)
 return block
}

func sigma(block uint32) uint32 {
 piVal := nPi(block)
 block ^= ROTL(block, 17) ^ ROTR(block, 11) ^ (block * uint32(piVal)) ^ 3149757
 block ^= (block ^ (uint32(piVal) >> 2) * 37)
 block ^= (uint32(piVal) << 3)
 return block
}

func M(block uint32) uint32 {
 piVal := nPi(block);
 num := block
 block = logisticMap(block, math.Abs(float64(piVal) / 3.14))
 block ^= block << 17
 block ^= block >> 19
 block |=
    ROTL(num * uint32(piVal), int(math.Abs(float64(piVal) - 15)))

 return block
}


func salting(compartment []uint32) []uint32 {
 size := len(compartment)
 for i:= 0; i < size; i++ {
  compartment[i] = logisticMap(compartment[i], 3.99)
  fb := fibCache[compartment[i] % 511]
  compartment[(i + 1) % size] = compartment[i]
        ^ sigma(compartment[(i + 2) % size]) ^ uint32(fb)
        ^ M(compartment[(i + 7) % size]);
  }
 return compartment
}

func stateCompression(compartment []uint32) []uint32 {
 var temp1 []uint32;
 for i := 0; i < len(compartment); i += 2 {
  block := logisticMap(compartment[i]
        ^ compartment[(i + 3) % len(compartment)], 3.99)
  temp1 = append(temp1, block)
```

```
 }
 var temp2 []uint32;

 for i := 0; i < len(temp1); i+=2 {
  fb := fibCache[temp1[(i + 1) % len(temp1)] % math.MaxInt8]
  piVal := nPi(fb)
  block := temp1[i] ^ temp1[(i + 2) % len(temp1)] | (fb / uint32(piVal))
  temp2 = append(temp2, block)
 }

 var states []uint32 = make([]uint32, 16);

 for i := 0; i < 16; i++ {
  it := (nPi(temp2[i % 16]) % 277) % math.MaxUint8;
  if it < 64
   it = 64
  }

  max := len(temp2)
  for j := 0; j < it; j++ {
   var lis float64 = float64(nPi(17) % 4)
   if lis / 4 == 0 {
    lis = 3.99
   }
   block := temp2[(i + 1) % max]
            ^ sigma(temp2[(i + 7) % max]) | M(temp2[(i + 11) % max])
            ^ logisticMap(temp2[(i + 5) % max], lis)
   fb := fibCache[temp2[i] % 511]
   block = block ^ fb;
   states[i] = block;
  }
 }
 return states
}
func main(){
 var input string
 fmt.Scan(&input)

 start := time.Now()

 padded := padding(input);
 compartment := compartmentize(padded);
 salted := salting(compartment)
 states := stateCompression(salted)
```

```
 elapsed := time.Since(start)

 for _, ele := range states
  fmt.Printf("%08x", ele)

 fmt.Print(" Elapsed: ");
 fmt.Println(elapsed)

}
```

3.4.4 Tests

On testing the nar512() function over the older ones with random strings - this is the readings I got. Over 97% improvement from the very first implementation.

Char Length	Average Time (s)	Total Time (s)	Improvement (%)
1	0.00042	0.42410	96.55%
10	0.00042	0.41825	96.61%
100	0.00042	0.41819	96.62%
250	0.00042	0.41600	96.65%
500	0.00041	0.40912	96.70%
1024	0.00044	0.43689	96.67%
5000	0.00065	0.64833	97.37%
10000	0.00091	0.91277	97.61%

Table 5: Execution Time Measurements after vector and recursive overwrites, with improvement percentages over initial implementation

The execution time comparison clearly shows the progressive optimizations that was achieved across multiple iterative improvements in implementations. Starting from the initial bitset-based approach, which exhibited the highest latency and low efficiency, each subsequent version — Vector PiMapped, Un-Recursive, and finally the Go implementation with fibonacii caching — demonstrated substantial performance gains.

Notably, the Go implementation outperforms all others, achieving up to **97.6% reduction** in average execution time compared to the original. Overall, the results validate a strong trend toward leaner, lower-level control yielding significant runtime benefits for high-frequency operations of nar512().

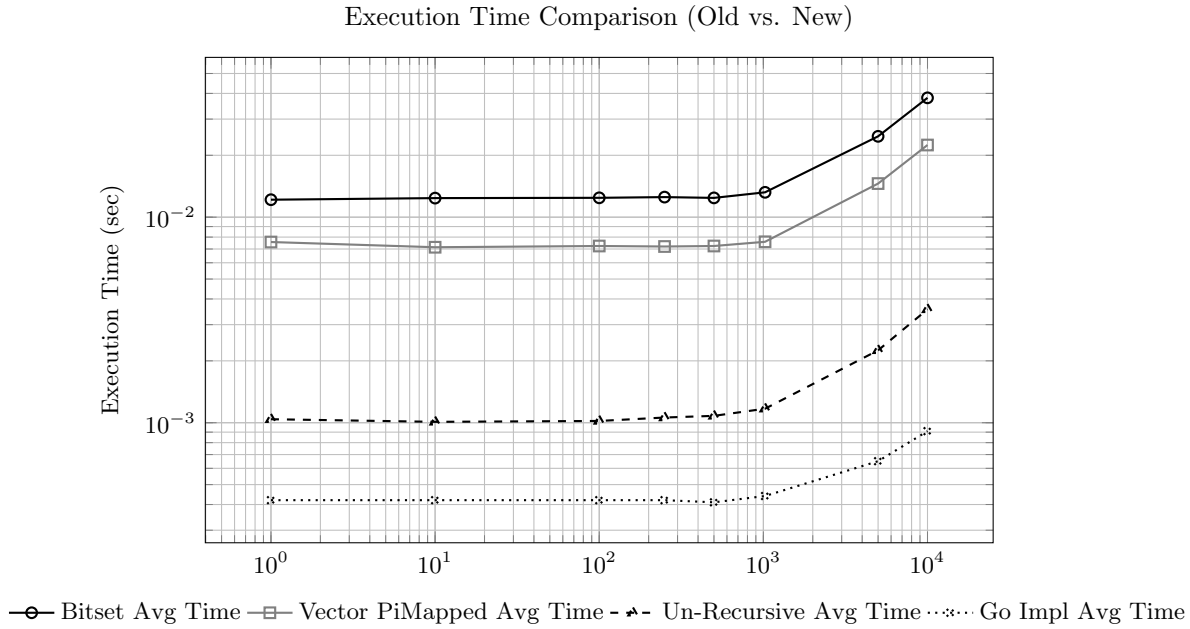

Figure 3: Overall comparisions and improvement in efficiency

3.5 My remarks

My algorithm diverges from SHA-256 in both structure and philosophy. While SHA-256 relies on a fixed-round Merkle–Damgård construction with predefined constants, bitwise logical functions, and modular additions, my approach builds on this rigidity with a more chaotic, input-sensitive process. I use Pi-mapped vectors for padding and entropy injection, ensuring that the state space evolves in a highly non-deterministic way, influenced by both input length and byte patterns. Recursive overwrites further destabilize intermediate states, eliminating linear propagation chains often exploited in differential cryptanalysis. Unlike SHA-256, where the diffusion pattern is well-mapped and might get predictable after years of study, my algorithm mutates its own structure during runtime, making pre-image discovery and collision crafting significantly more difficult. While SHA-256 benefits from hardware acceleration and is standardized for broad compatibility, my design intentionally sacrifices some of that efficiency to maximize state volatility, making it better suited for environments where unpredictability and adaptive resistance are more critical than raw throughput.

And for fun, the output I get from my function from giving the entire paragraph before this as input is:

```
5530fde4777376772e4480b30ea2df45215a98300b5678bc
73a0111693c5a7bfbbc2a8ec4b647fb1179dd07c570f3582
d7b064e0fefdda97fbc8d7606ff0719d

Time Elapsed: 1.339042ms
```

4 Conclusion

In this paper, I introduced a novel cryptographic hash function, integrating elements of chaotic maps, number-theoretic transformations, and bitwise operations to enhance security and unpredictability. Our approach leverages logistic maps for diffusion, Fibonacci sequences for dynamic state modification, and the mathematical properties of prime numbers to introduce non-linearity.

Through a series of transformations, the input data undergoes substantial entropy amplification, ensuring resistance to common cryptanalysis techniques such as differential and linear cryptanalysis. The final compressed state exhibits high sensitivity to initial conditions, a hallmark of chaotic systems, making the function highly unpredictable.