Collatz Hash: Hash Algorithm Using $3x + 1$ Conjecture

Shaurya Pratap Singh[1], Bhupendra Singh,[2] and Alok Mishra[3]

[1] Centre for Artificial Intelligence & Robotics
Defence Research and Development Organisation(DRDO),
Bengaluru, India
shauryapratap2114@gmail.com
[2] Centre for Artificial Intelligence & Robotics
Defence Research and Development Organisation(DRDO),
Bengaluru, India
bhupendra.cair@gov.in
[3] Centre for Artificial Intelligence & Robotics
Defence Research and Development Organisation(DRDO),
Bengaluru, India
alokmishra.cair@gov.in

Abstract. In this paper, we introduce a new hash algorithm in that we used the Collatz problem, focusing on its conditional branching structure, an element often overlooked despite the fame of the $3x + 1$ conjecture. This hash algorithm takes an arbitrary-length input and produces a fixed-length 512/384/256 bit output. The presented hash algorithm is in the category of strong one-way Hash Function (OWHF), and this hash algorithm is designed by focusing on its use in password storing and Pseudo-Random Number Generator (PRNG).
Until now, quantum computers also didn't show any speedup in the Collatz conjecture problem, so the proving and disproving of the Collatz conjecture is still a big question, so we believe that there is no structural attack on this hash algorithm.

Keywords: Collatz Conjecture, Hash Algorithm, Hash Function, One-way Hash Function (OWHF).

1 Introduction

The Collatz conjecture is a very famous problem of mathematics, first introduced by German mathematician Lothar Collatz in 1937. Almost all the famous mathematicians till now have tried this conjecture and have not been able to show a useful growth toward a solution and suggest others also not try to solve this problem. In this paper, we are also not trying to solve this problem; instead, we are using this conjecture in our hash algorithm and presenting a new cryptographic hash algorithm that uses the Collatz conjecture function in its internal structure and produces a fixed-size output.

In this document, we are proposing Collatz Hash, a new hash algorithm, which is going to give output of sizes 512/384/256 bits. In the construction of the Collatz hash, we designed a new compression function that took 1024 bits message block and 512 bit H_{i-1} as input and produced a fixed size H_i, later we explained it in more detail. Below we mention a basic introduction about hash algorithms and their type.

1.1 Cryptographic Hash Algorithms

Hash algorithms are one of the most important building blocks of modern security. Most of the digital security is based on the hash algorithm; some are like message authentication code (MAC) for secure message communication, digital signature for identifying the right sender, etc. Hash algorithms take a variable-size input and give a fixed-size output, and a small change in input changes the output completely, meaning it satisfies the avalanche effect [1].

Definition 1.1. A hash function is a function of the form

$$h(x) = y \quad \text{where } x \in \{0,1\}^* \text{ and } y \in \{0,1\}^n$$

here x is an arbitrary-length input, and y is a fixed-size output, which has a wide variety of names in the cryptography literature: hash result, imprint, checksum, compression, authenticator, fingerprint, Message Integrity Code (MIC), etc.

In this section we divide hash functions into two categories: One-Way Hash Function (OWHF) and Collision-Resistant Hash Function (CRHF). They are used in Manipulation Detection Code (MDC).

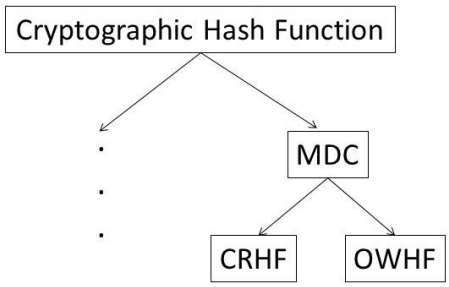

Fig. 1. : Categories of Hash Function

1.1.1 One-Way Hash Function (OWHF)

By the name, it's clear that the hash function h produces an output y by passing a input x then it is "hard" to find the **x** by **y**, meaning with the help of the hash value, it is difficult to find the message that is hashed. This property of the hash function is called **One Wayness**. The informal description of the hash function is provided by M. Rabin [4] & R. Merkle [2], [3].

Formally, a hash function h satisfying the following property then it's a one-way hash function:

1. The algorithm of a hash function is publicly known by everyone and should not require any hidden information for generating a hash result (Kerckhoff's principle).
2. Input of hash algorithm (x) should not have any restriction about the size of x.
3. Computing the hash result with input x must be "easy".
4. The function that produces the hash needs to be one-way in the sense that, given a y in the image of h, it is "hard" to track down a message x when $h(x) = y$ given x, and $h(x)$ it is "difficult" to find a message $x' \neq x$ such that $h(x') = h(x)$.

In the above point, it is explained that when we have a hash value, it must be very difficult to find the input of the hash algorithm; this is called one-wayness or preimage resistance. And when we have an input x and we are finding another input x' such that $h(x') = h(x)$ it must be at least exponentially hard.

1.1.2 Collision Resistance Hash Function (CRHF)

In CRHF all the properties of OWHF must be satisfied, and there is one extra property called collision resistance, as mentioned in [5], [3], [6].

1. The algorithm of a hash function is publicly known by everyone and should not require any hidden information for generating a hash result (Kerckhoff's principle).
2. Input of hash algorithm (x) should not have any restriction about the size of x.
3. Computing the hash result with input x must be "easy".

4. The function that produces the hash needs to be one-way in the sense that, given a y in the image of h, it is "hard" to track down a message x when $h(x) = y$ given x, and $h(x)$ it is "difficult" to find a message $x' \neq x$ such that$h(x') = h(x)$.

5. The hash algorithm should be collision resistant, meaning it is "difficult" to find two input x & x' whose hash values are going to be the same.

Examples of cryptographic hash functions include the MD family (MD2, MD4, MD5) and the SHA family (SHA-0, SHA-1, SHA-224, SHA-256, SHA-384, SHA-512), [1]. However, numerous other hash functions exist, each serving different purposes.

1.2 Collatz Problem Overview

The Collatz Problem, also known as the $3x + 1$ problem, the Ulam conjecture, or the Syracuse problem, is an unsolved mathematical conjecture that is easy to state but has perplexed mathematicians for decades. It is named after the German mathematician Lothar Collatz, who introduced the problem in 1937 [7], [8], [9].

– **Statement of the Problem**

The Collatz Problem is defined as follows:

1. Start with any positive integer x.
2. If x is even, divide it by 2: $x \to \frac{x}{2}$.
3. If x is odd, multiply it by 3 and add 1, then divide by 2: $x \to \frac{3x+1}{2}$.
4. Repeat the process indefinitely.

The conjecture asserts that no matter what positive integer x you start with, you will eventually reach the number 1.

Example 1.1

Let's consider an example to understand how the process works:

– Start with $x = 7$ (which is odd so we apply $3x + 1$)

$$7 \to 3 \times 7 + 1 = 22$$

– 22 is even, so we apply x divide by 2

$$22 \to \frac{22}{2} = 11$$

– 11 is odd:

$$11 \to 3 \times 11 + 1 = 34$$

– 34 is even:

$$34 \to \frac{34}{2} = 17$$

– 17 is odd:

$$17 \to 3 \times 17 + 1 = 52 \to \frac{52}{2} = 26 \to \frac{26}{2} = 13 \to 3 \times 13 + 1 = 40 \to \frac{40}{2} = 20 \to \frac{20}{2} = 10$$

$$\to \frac{10}{2} = 5 \to 3 \times 5 + 1 = 16 \to \frac{16}{2} = 8 \to \frac{8}{2} = 4 \to \frac{4}{2} = 2 \to \frac{2}{2} = 1$$

Thus, starting with $x = 7$, the sequence goes 7, 11, 17, 13, 5, 1. As predicted, we eventually reach to 1.

Mathematically, the Collatz function $C : \mathbb{N} \to \mathbb{N}$ can be defined as

$$C(x) = \begin{cases} \frac{x}{2} & \text{if } x \text{ is even,} \\ \frac{3x+1}{2} & \text{if } x \text{ is odd.} \end{cases}$$

The sequence generated by repeatedly applying C to an initial integer x is known as the *Collatz sequence* of x. The Collatz conjecture asserts that, for all $x \in \mathbb{N}$, there exists some $k \in \mathbb{N}$ such that $C^k(x) = 1$, where $C^k(x)$ denotes the k-th iterate of C on x [8], [10].

1.2.1 Current Status and Open Questions

Despite its simple definition, the Collatz Problem remains unsolved. No one has yet been able to prove or disprove the conjecture. The problem has been verified for large numbers through computational methods, but a general proof for all positive integers has not been found [9].

Some open questions related to the Collatz Problem include:

- **Does the sequence always reach 1?** While this has been verified for many numbers, a formal proof remains elusive.
- **Are there cycles other than 1?** The conjecture suggests that every starting value eventually reaches 1, implying no other cycles exist. However, this has not been proven.
- **What is the longest sequence before reaching 1?** For any given x, the number of steps it takes to reach 1 varies, and the maximum number of steps before reaching 1 is of interest.

1.2.2 Why It is Hard to Prove Collatz Conjecture

We know that if we have a positive integer, x then to verify Collatz conjecture, we repeat the step of odd and even until x reaches to 1. But still there is no proof for every number. Why? Let's see.

We are going to take dummy conjecture, which is defined as:

$$D(x) = \begin{cases} \frac{x}{2} & \text{if } x \text{ is even,} \\ \frac{x+1}{2} & \text{if } x \text{ is odd.} \end{cases}$$

So if a number x is even, then we divide it by 2, and if a number is odd, then we add 1 and divide it by 2 until we reach to one. If we compare the orbits of $C(x)$ and $D(x)$, then for $x = 10$ $C(x)$ going to be

$$10 \to 5 \to 8 \to 4 \to 2 \to 1$$

and $D(x)$ going to be

$$10 \to 5 \to 3 \to 2 \to 1$$

We see here that the orbit of $D(x)$ reach to 1 much faster than $C(x)$ and each element of $D(x)$ the orbit is less than the previous one. So we can say that the orbit under $D(x)$ always reaches to one. Why?

Dummy conjecture, with the fact that half of the integer is always less than the integer itself. If x is even, then by the Dummy conjecture definition, it is going to be $D(x) = x/2 < x$ mean if the number is even, then the next number is always less than the x.

If x is odd, then in $D(x)$ we add 1 and again divide it by 2. For odd x orbit will look like this:

$$x \to x+1 \to \frac{x+1}{2} \to ...$$

Notice that $\frac{x+1}{2} = \frac{x}{2} + \frac{1}{2}$. We know that $\frac{x}{2} < x$ and $\frac{1}{2}$ is small, so $\frac{x+1}{2}$ is also less than then x.

So when an orbit is under $D(x)$ and gets an odd number greater than 1, we will always be at a smaller number after two steps. We will observe here if x odd or even; it is always going to some smaller number in each step. We can say that after some steps it always going to reach 1.

But in the case of Collatz conjecture, when $C(x)$ get an odd number, like 5 it's going to triple the number and add 1 then divide it by 2, which is going to 8 which is greater than, 5 so here we can't say that in each step it's going to get some smaller number than x. That's why here it is very hard to prove it always going to reach 1. The image below describes the structure of Collatz conjecture.

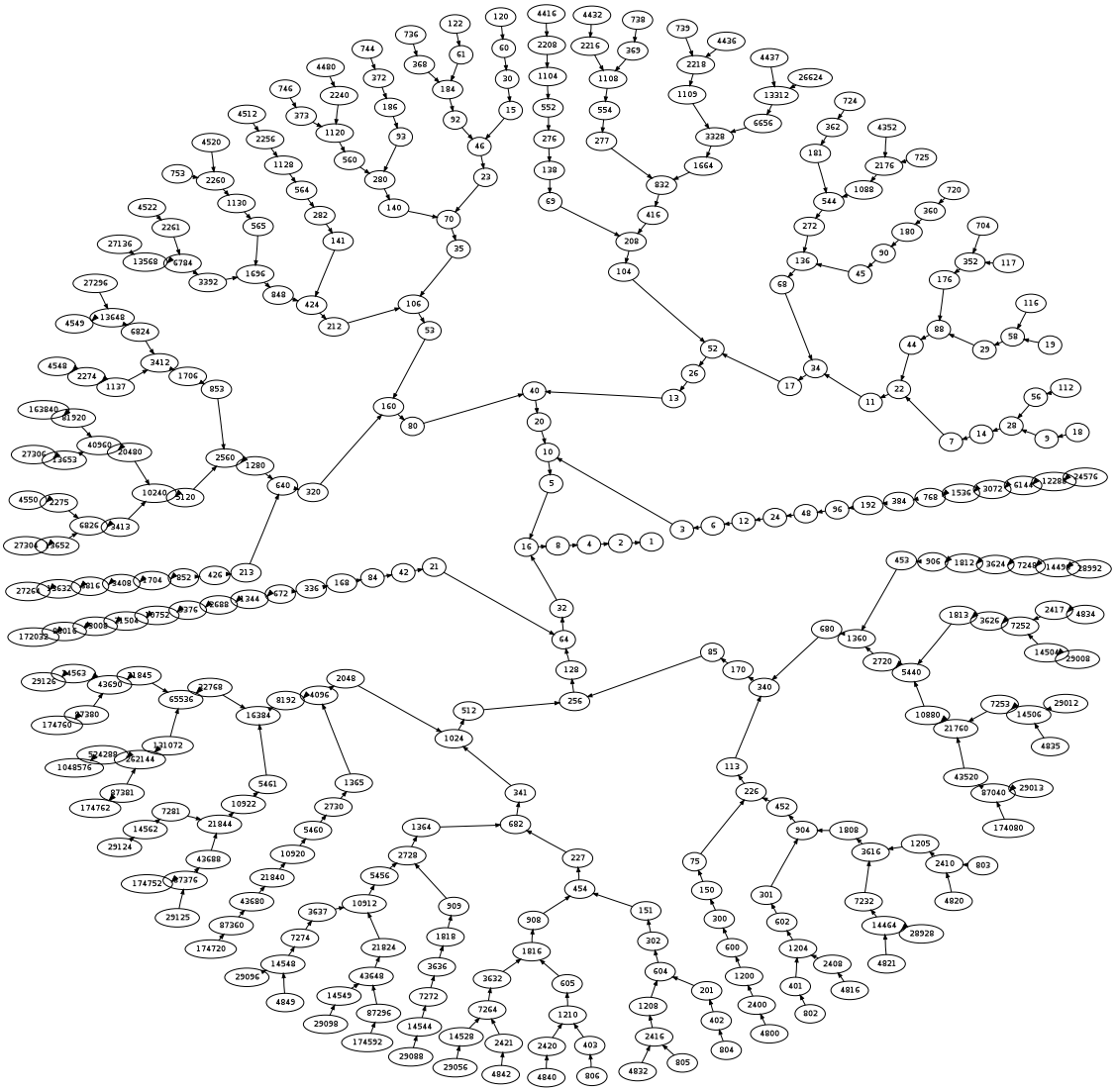

Fig. 2. : Collatz Conjecture Structure

Significance in Mathematics

The Collatz Problem is often cited as an example of a problem that is deceptively simple to state but difficult to solve. It illustrates the unpredictability and complexity that can arise in iterative processes. The problem has connections to various fields, including number theory, dynamical systems, and computer science [10]. Although the Collatz Problem has not been solved, studying it has led to insights into other areas of mathematics and has inspired the development of various computational techniques, and we are also using the structure in the proposed cryptographic hash algorithm.

2 Design Goals

In this section, we give a brief motivation of the Collatz hash proposal. The overall objectives are to demonstrate desirable qualities, make the design elegant, and make the analysis simple.The fact that the compression function proposed in this is based on the $3x + 1$ conjecture and it is very hard to revert.

2.1 Overall goal of the hash

Here we gave the overall design goal for Collatz hash.

- Simplicity of analysis: hence, Collatz conjecture is already studies from many decades, and it is proven that it is hard to revert.
- No special preference for a particular platform or word size, and good performance in a very wide range of platforms.
- Prevention of length-extension attack, which is very common in Merkle-Damgard construction.

2.2 Failure-tolerant design

- The internal state is significantly larger than the final output, so all the known generic attacks are thwarted.
- An attack on the Collatz conjecture function may not lead to an attack on the hash function because there are some other components also outside the conjecture iteration.
- There is no known attack better than brute-force on the compression function, by the fact of $3x + 1$ conjecture.

2.3 The compression function's design consideration

Some of the most used and famous hash functions are based on the Merkle–Damgård construction, e.g., MD5, SHA-1, SHA-256 [20], Tiger [21], etc. So the block cipher designs are well explored. The recent attacks [[22], [16], [17], [18], [23]] on the SHA-1 and Tiger illustrate lots of issues. Keeping this thing in mind, our proposal uses few input-based deterministic shuffling of numbers and the world-famous Collatz conjecture for strong one-wayness. The advantages of such a design methodology are as follows:

- Unlike MD5, SHA-1, and SHA-2, we are not taking any constant K values.
- One-wayness hardness easily reduces to a known hard problem.
- Most of the decisions of the presented compression function, like selection of word, rotation of word, etc., are based on the input directly or indirectly, so it reduces the probability of collision.

3 Description of the Collatz Hash

3.1 The Hash Function construction

The Collatz conjecture is an iterative hash function that is based on the Merkle-Damgard design. The main component of the Merkle-Damgard design is their compression function. In the presented hash algorithm there are two parts in the compression function(f) ,f_1 and f_2 explain below:

$$f = \begin{cases} f_1 : \{0,1\}^{512} \times \{0,1\}^{1024} \rightarrow \{0,1\}^{512} \\ f_2 : \{0,1\}^{512} \rightarrow \{0,1\}^{640} \end{cases} \tag{1}$$

$\rightarrow f_1$ will take $1024 - bit$ message block and $512 - bit$ previous hash digest(H_{i-1}) and give $512 - bit$ output.

$\rightarrow f_2$ will take $512 - bit$ output of f_1 and it will give $640 - bit$ output, and after truncating, it will give the final digest. In section 3.2 and 4.3 explain in more detail.

To compute the $H(M)$ message, it is first divided into N 1024 bit blocks. Then the compress function is used to compress the message blocks and the internal state:

$$H_{i+1} = f(H_i, M_i) \tag{2}$$

There is a padding rule to fill the last message block if it is not 1024-bit; for the padding we are using Merkle-Damgard strengthening. The initial value of the state H_{i-1} is called IV and is fixed in the description of the hash function.

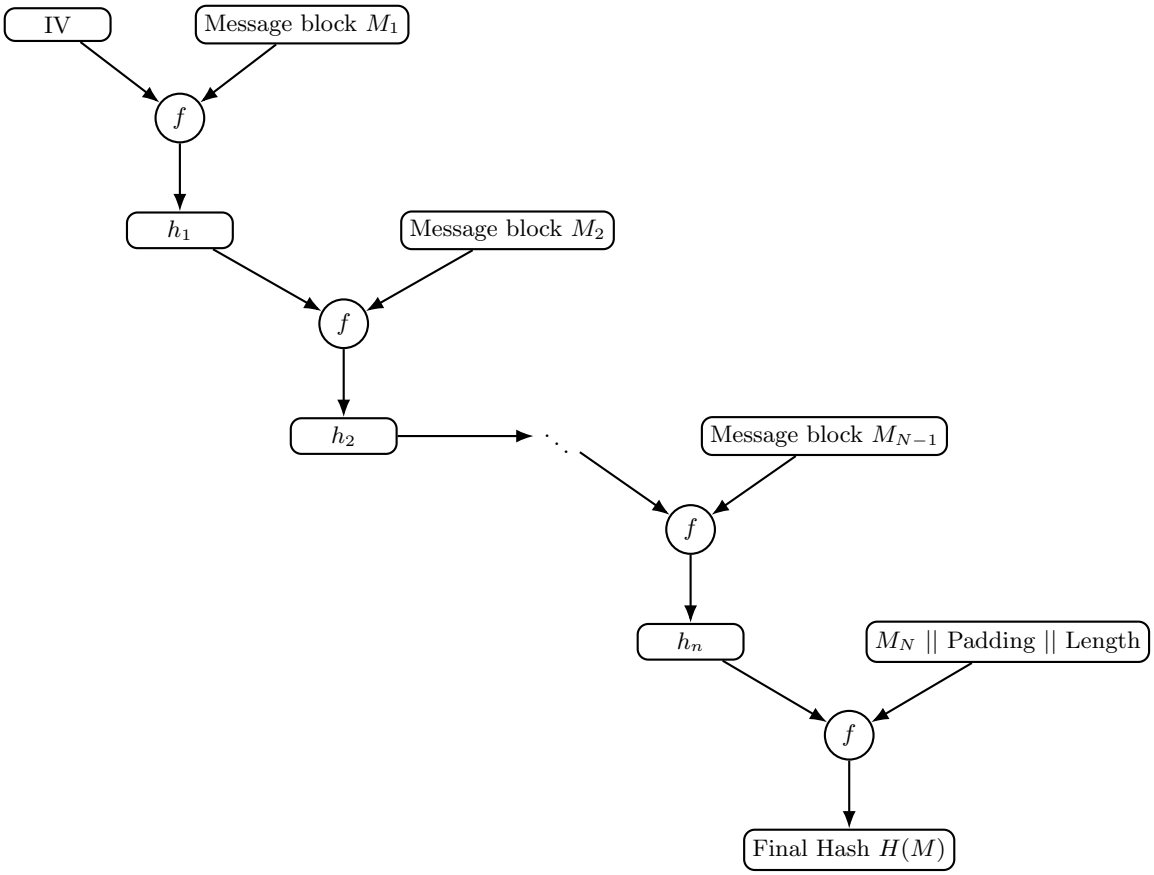

Fig. 3. Merkle–Damgård Construction

3.2 New Compression Function f

We are following Merkle–Damgård construction for the hash algorithm and designing a new compression function in the presented hash algorithm. There are 7 rounds, and each round has two main operations to perform. The first operation performed inside the iteration bounded by $3x+1$ conjecture under that iteration variable a, b, c, d, e, f, g, h is going to update as explained in figure-4:

$x \in X[16]$, $X[\,] = \{a_0, a_1, ..., h_0, h_1\}$.

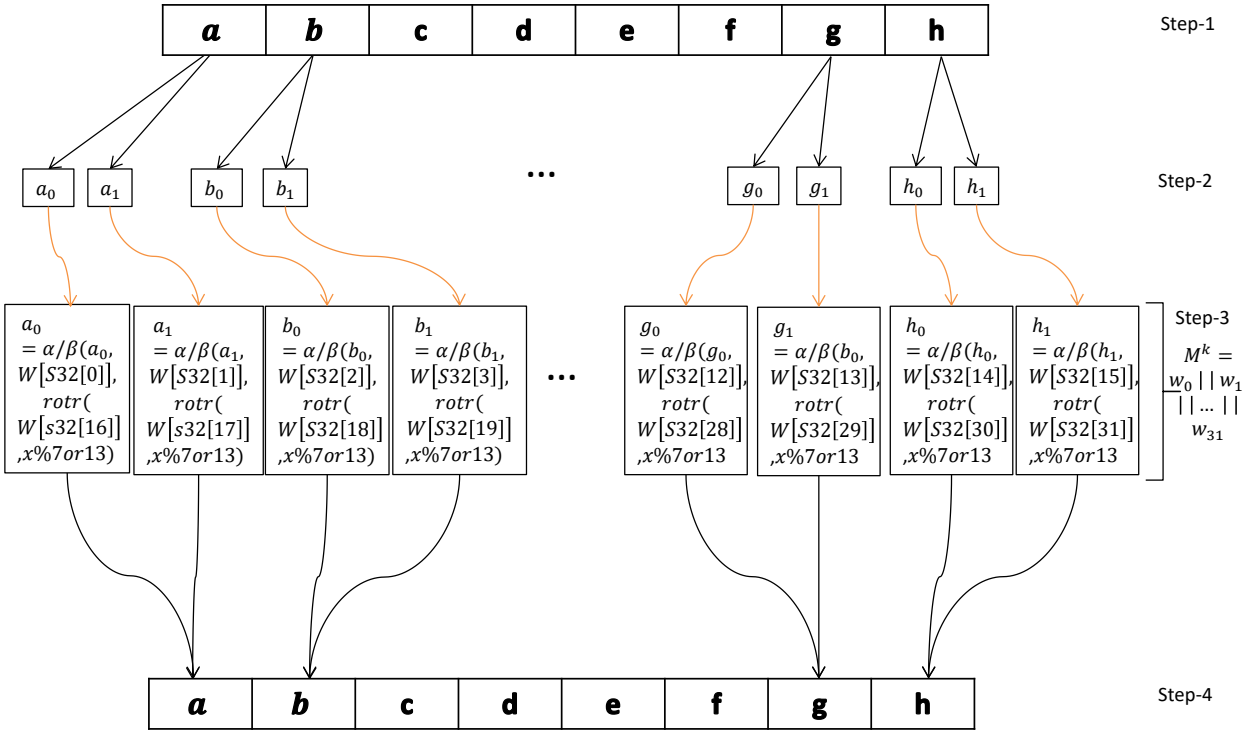

Fig. 4. : 1^{st} Operation of Compression Function

Step-1: We have a, b, c, d, e, f, g, h , each variable is 64-bit and when we concatenate these variable become 512- bit , so for the first message block, M^1 we define these variables as the initial vector(IV) or H^0.

Step-2: We divide each variable into 2 parts; each part is 32-bit.

Step-3: We update the each 32-bit part of variable with operation perform by function $alpha()$ or function $beta()$ which is define as

Algorithm 1: Function $\alpha(j, k, l)$

1: $t \leftarrow (j + k + (l \ll 1))$ // addition + left shift
2: $r \leftarrow \text{ROTR}_{16}(t, 7)$ // rotate right by 7
3: **return** $(r \oplus j) + (k \oplus \text{ROTR}_{16}(l, 3))$

Algorithm 2: Function $\beta(j, k, l)$

1: $choose \leftarrow (j \wedge k) \oplus ((\neg j) \wedge l)$
2: $maj \leftarrow (j \wedge k) \oplus (j \wedge l) \oplus (k \wedge l)$
3: $t \leftarrow (choose + maj + \text{ROTR}_{16}(j \oplus k, 5))\&0xFFFF$
4: **return** $t \oplus \text{ROTR}_{16}(l, 11)$

The message block is divided into 32 words, and each word size is 32-bit.

$$M^k = w_0||w_1||...||w_{31}$$

If we see the variable, a then it divided into two sub-variables, a_0 & a_1 and, with the help of function $alpha()$ or $beta()$ they are going to update. The choice of function and modulo operator depends on the value of x, if x is even, then $alpha()$ and $x\%9$ otherwise, $beta()$ and $x\%13$, then again, the concatenation of 32-bit sub-variables into a 64-bit variable is applied.

$$a+ = a_0||a_1$$

Step: 4 The same operation of step-3 is perform with a, b, c, d, e, f, g, h and at the end of the conjecture loop, we have updated a, b, c, d, e, f, g, h.

The second operation is performed on the updated version of a, b, c, d, e, f, g, h as explained in Figure-5.

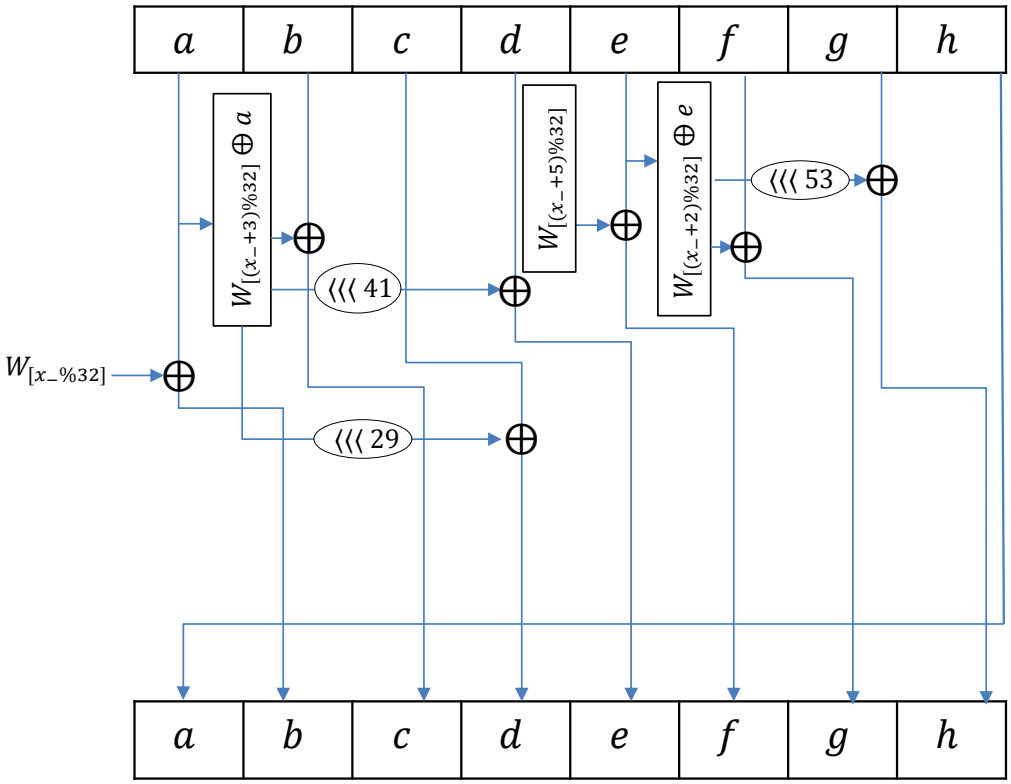

Fig. 5. : 2^{nd} Operation of Compression Function

In Figure-5 operation, "$a \oplus W[(x_ + 3)\%32]$" and "$e \oplus W[(x_ + 2)\%32]$" are stored in variable T_1 and T_2 so we have total 10 variable.

So in every round, first we update variables inside the conjecture loop by function $alpha()$ and $beta()$, after that, on the variables a, b, c, d, e, f, g, h the 2^{nd} operation of the compression function is performed as explained in Figure-5.

3.3 Final Mixing

After the N^{th} message block we have the final updated variables a, b, c, d, e, f, g, h, after that we perform the mixing operation as explained in Figure-6; after that we have the final digest output.

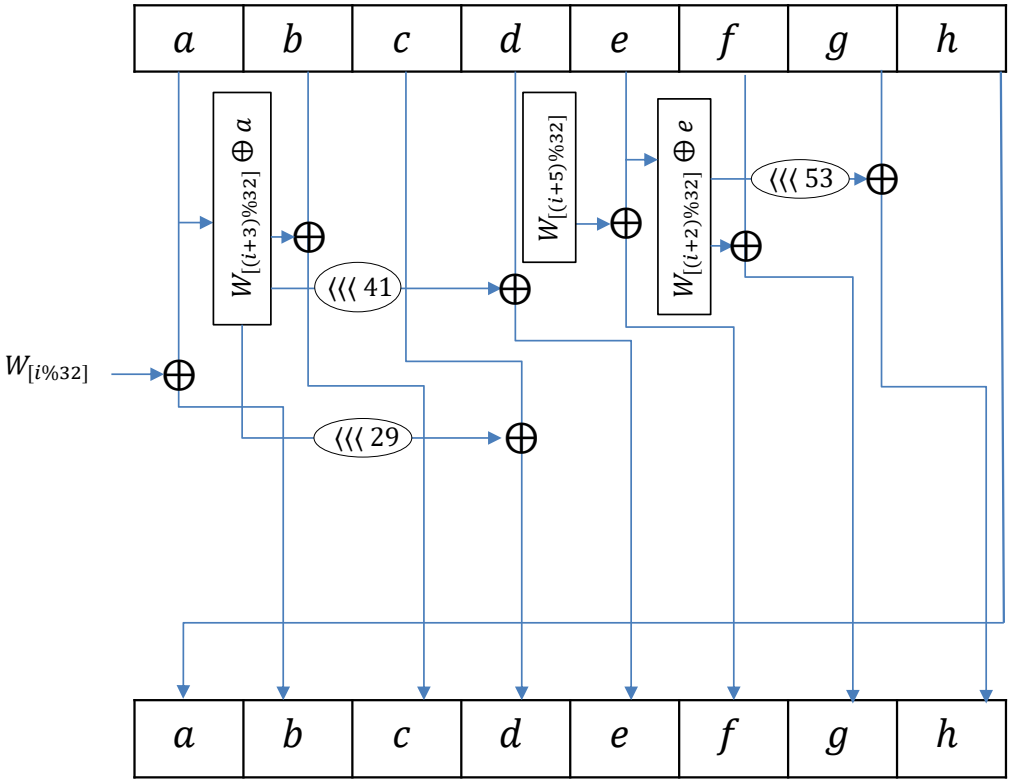

Fig. 6. : Variable Mixing

In figure-6 operation $a \oplus W[i\%32]$ and $e \oplus W[(i+2)\%32]$ is stored in variable T_1 and T_2 so we have total 10 variables.

* $x_$ is the same as x.
* i is the variable that iterates through each message block, so when this variable is at the last block, then the function explained in Figure-6 is going to execute.
* <<< is left circular rotation.

3.4 Generating Message Digest

The message digest is generated by truncating 640-bit(10 variables).

3.4.1 Collatz Hash-256

The message digest is generated by truncating 256-bits(4 variables) from 640-bit(10 variables).

Algorithm 3: Selecting 4 Variables from 10

Data: Array of 10 variables: $[a, b, c, d, e, f, g, h, T_1, T_2]$
Data: Array $shuffled10[]$ containing a permutation of $\{0, \ldots, 9\}$
Result: An array $Digest[0..3]$ containing 4 selected variables

```
for i ← 0 to 3 do
    Digest[i] ← vars[shuffled10[i]];
return Digest[0..3];
```

$$H = Digest[0]||Digest[1]||Digest[2]||Digest[3]$$

3.4.2 Collatz Hash-384

The message digest is generated by truncating 384-bits(6 variables) from 640-bit(10 variables).

Algorithm 4: Selecting 6 Variables from 10

Data: Array of 10 variables: $[a, b, c, d, e, f, g, h, T_1, T_2]$
Data: Array $shuffled10[]$ containing a permutation of $\{0, \ldots, 9\}$
Result: An array $Digest[0..5]$ containing 6 selected variables

```
for i ← 0 to 5 do
    Digest[i] ← vars[shuffled10[i]];
return Digest[0..5];
```

$$H = Digest[0]||Digest[1]||Digest[2]||Digest[3]||Digest[4]||Digest[5]$$

3.4.3 Collatz Hash-512

The message digest is generated by truncating 512-bits(8 variables) from 640-bit(10 variables).

Algorithm 5: Selecting 8 Variables from 10

Data: Array of 10 variables: $[a, b, c, d, e, f, g, h, T_1, T_2]$
Data: Array $shuffled10[]$ containing a permutation of $\{0, \ldots, 9\}$
Result: An array $Digest[0..7]$ containing 8 selected variables

```
for i ← 0 to 7 do
    Digest[i] ← vars[shuffled10[i]];
return Digest[0..7];
```

$$H = Digest[0]||Digest[1]||Digest[2]||Digest[3]||Digest[4]||Digest[5]||Digest[6]||Digest[7]$$

* shuffled32[] stores the 32 shuffled numbers, and shuffling is performed based on the original input message.
* shuffled10[] stores the 10 shuffled numbers, and shuffling is performed based on the original input message.
* $rotr16()$ function take 16-bit input and an integer number and performs right rotation.
* $\|$ this sign shows concatenation.

4 Proposed Hash Algorithm

4.1 Terminology and Notation

4.1.1 Operations

The following operations are used in the Collatz hash specifications:

- & Bitwise AND operation.

– << Bitwise left shift operation.

– $\oplus$ Bitwise XOR ("exclusive–OR") operation.

– + Denote the addition.

– $\neg$ Bitwise complement operation.

– $\|$ Concatenation operation.

4.1.2 Parameters

The following parameters are used in the Collatz hash specifications:

– H Number of bits in a hash value. $h = 512/384/256$.

– H_i The ith hash value, with a size of h bits. H_0 is the initial hash value; H_N is the hash value after N^{th} message block.

– l Length of the original message before padding in bits.

– k Number of zeros appended in the original message.

– M^n Message to be hashed.

– N Number of blocks after padding the message.

4.1.3 Notations

The following notations are used in the Collatz hash specifications:

– **Word** w_i group of bits.

– w_i The i^{th} word of the message block.

$$M^n = w_0 \parallel w_1 \parallel w_2 \parallel \cdots \parallel w_{32-1}.$$

4.2 Collatz Hash Algorithm Description

We begin by supposing that we have a l-bit message as input and that we wish to find its message digest. Here l is an arbitrary nonnegative integer; l may be zero.

$$m_0 \; m_1 \; \ldots \; m_{l-1}$$

The following six steps are performed to compute the hash value of the message.

Step 1. Append Padding Bits

The message is "padded" (extended) so that its length (in bits) is going to be equal to block length$(1024 - bits)$. Let message(m) length is len. Then append the bit "1" to the end of the message, followed by k zero bits, so $len + 1 + k \equiv 1024$.

Step 2. Append Length

A 128-bit representation of len, the length of the message before the padding bits were appended to the result of the previous step. In the unlikely event that $len > 2^{64}$, then only the low-order 128 bits len are used.

At this point the resulting message (after padding with bits and with len) has a length that is an exact multiple of 1024 bits.

Step 3. Parsing the padded message

After a message has been padded, it is parsed into N 1024-bit block, M^1, M^2, ..., M^N . The 1024-bit message block is expressed as 32 words($w_0, w_1, ...w_{31}$) each word is size of $32 - bit$.

Step 4. Setting the initial hash value(H_0) or initial vector(IV)

The initial hash value H_0 is set for the first message block computation; each variable a, b, c, d, e, f, g, h is $64 - bit$.

$H_0 = a||b||c||d||e||f||g||h$

```
a=0x6a09e667f3bcc908;
b=0xbb67ae8584caa73b;
c=0x3c6ef372fe94f82b;
d=0xa54ff53a5f1d36f1;
e=0x510e527fade682d1;
f=0x9b05688c2b3e6c1f;
g=0x1f83d9abfb41bd6b;
h=0x5be0cd19137e2179;
```

Step 5. Computing the Hash value H_N

The compressor function f is applied to each M^1, M^2, ..., M^N to obtain the H_N. Based on the original input, we generate two shuffled arrays by the algorithm-6:

Algorithm 6: Deterministic Shuffle

Input : key (string), UpperLimit (integer)
Output: result (list of integers)

```
sum1 ← 0, sum2 ← 0
for i ← 0 to length(key) − 1 do
    if i mod 2 = 0 then
        sum1 ← sum1 + ASCII(key[i])
    else
        sum2 ← sum2 + ASCII(key[i])
    end
end
s ← ((sum1 mod (UpperLimit/2)) × 2) + 1
// Odd step size: 1 to UpperLimit-1
o ← sum2 mod UpperLimit
// Offset: 0 to UpperLimit-1
result ← empty list
for k ← 0 to UpperLimit − 1 do
    val ← (o + k × s) mod UpperLimit
    Append val to result
end
return result
```

Shuffled10[10]=Contain shuffled 10 numbers from 0 to 9 generated by algorithm-6.
Shuffled32[32]=Contain shuffled 32 numbers from 0 to 31 generated by algorithm-6.

For $i = 1$ to N

// we store the 32-bit parts of each a, b, c, d, e, f, g, h into an array $X[\,]$ like:

$X[0] = a_0, X[1] = a_1, X[2] = b_0, ..., X[13] = g_1, X[14] = h_0, X[15] = h_1$

and then perform the mixing operation:

Algorithm 7: Mixing Step with Rotation and XOR

Input: Array $X[0 \dots 15]$
Output: Updated array X

for $i \leftarrow 0$ **to** 15 **do**
  $left \leftarrow X[(i + 15) \bmod 16]$;
  $right \leftarrow X[(i + 1) \bmod 16]$;
  $mix \leftarrow \big(left + \text{rotr16}(right,\ (i \,\&\, 3) + 3)\big) \bmod 2^{16}$;
  $X[i] \leftarrow X[i] \oplus \text{rotr16}(mix,\ (i \,\&\, 15))$;

after that divide 1024-bit message block into 32 words, each word of size 32-bit.

$M^n = w_0||w_1||...||w_{28}||w_{29}||w_{30}||w_{31}$, {sizeof($w_l$)=32 bit, $32 > l > -1$ }

Algorithm 8: Compression Function

Input: X, w, $shuffled32$
Output: Updated state variables $a0, \dots, h1$

for $i \leftarrow 0$ **to** 6 **do**
  $x \leftarrow (X[shuffled32[i] \mod 16] >> (48 - i \mod 2 * 16) \& 0xFFFF)$;
  $x_ \leftarrow x$;
  while $x > 2$ **do**
    if $x \bmod 2 = 0$ **then**
      foreach *pair* (u, v) *in* $\{(a_0, a_1), (b_0, b_1), \dots, (h_0, h_1)\}$ **do**
        update (u, v) with α and rotr16($w[\cdot]$, $x \bmod 7$);
      $x \leftarrow x/2$;
    else
      foreach *pair* (u, v) *in* $\{(a_0, a_1), (b_0, b_1), \dots, (h_0, h_1)\}$ **do**
        update (u, v) with β and rotr16($w[\cdot]$, $x \bmod 13$);
      $x \leftarrow (3 \times x + 1)/2$;
  $a \leftarrow a + a_0 + a_1; \quad b \leftarrow b + b_0 + b_1;$
  $c \leftarrow c + c_0 + c_1; \quad d \leftarrow d + d_0 + d_1;$
  $e \leftarrow e + e_0 + e_1; \quad f \leftarrow f + f_0 + f_1;$
  $g \leftarrow g + g_0 + g_1; \quad h \leftarrow h + h_0 + h_1;$

  $T_1 \leftarrow a \oplus W[(x_ + 3) \bmod 32]$;
  $b \leftarrow a \oplus W[x_ \bmod 32]$;
  $c \leftarrow b \oplus T_1$;
  $d \leftarrow c \oplus \text{ROTL}(T_1, 29)$;
  $e \leftarrow d \oplus \text{ROTL}(T_1, 41)$;
  $T_2 \leftarrow e \oplus W[(x_ + 2) \bmod 32]$;
  $f \leftarrow e \oplus W[(x_ + 5) \bmod 32]$;
  $g \leftarrow f \oplus T_2$;
  $h \leftarrow g \oplus \text{ROTL}(T_2, 53)$;
  $a \leftarrow h \oplus \text{ROTL}(T_2, 13)$;

This process is repeated for every message block, and the value of the variables a, b, c, d, e, f, g, h after N^{th} message block passes to step-6 for the final message digest.

Step 6. Final Hash

In this step we perform operation on variable $a, b, c, d, e, f, g, h.$ as explained in figure-6 as:

Algorithm 9: Final Update

Input: $a, b, c, d, e, f, g, h, T_1, T_2, W[0..31]$
Output: Updated $a, b, c, d, e, f, g, h, T_1, T_2$

$T_1 \leftarrow a \oplus W[i \bmod 32]$;
$b \leftarrow a \oplus W[(i+5) \bmod 32]$;
$c \leftarrow b \oplus T_1$;
$d \leftarrow c \oplus ROTL\{T_1, 29\}$;
$e \leftarrow d \oplus ROTL\{T_1, 41\}$;
$T_2 \leftarrow e \oplus W[(i+1) \bmod 32]$;
$f \leftarrow e \oplus W[(i+7) \bmod 32]$;
$g \leftarrow f \oplus T_2$;
$h \leftarrow g \oplus ROTL\{T_2, 53\}$;
$a \leftarrow h \oplus ROTL\{T_2, 13\}$;

after step-6, the final digest is:

Algorithm 10: Hash Output

Input: Array(containing variable $a, b, c, d, e, f, g, h, T_1, T_2$), S
Output: $S - bit\ digest$; `// S is size of the digest`

$S \leftarrow \frac{S}{64}$
$H \leftarrow \epsilon$; `// Initialize as empty string`
for $i \leftarrow 0$ **to** S **do**
  $H \leftarrow H$ || toHexString($Array[shuffled10[i]]$);

Table 1. Hash Algorithm Input and Output

Input Pair	Output
Collatz Hash-256(“ ”)	0x f00e471f1fb38091cd1cc5e625f5a4fc3c6ef3747e94f82bc97554f28153d9d8
Collatz Hash-256(“The quick brown fox jumps over the lazy dog”)	0x 3c6ef3773daf03a67d5033441df51de92d20d7aac8737c64e888862f67d44320
Collatz Hash-384(“ ”)	0x f00e471f1fb38091cd1cc5e625f5a4fc3c6ef3747e94f82bc97554f28153d9d894c 31fa49553c0b6deba35b112abaa04
Collatz Hash-384(“The quick brown fox jumps over the lazy dog”)	0x 1955d4a3a215f71b4fc521a09e2759b90b2cd027938f3425cf5ebe969e80f77243 f097262f4ef51733e3c704cbb866e4
Collatz Hash-512(“ ”)	0x 09d60fab146298df1c992e0fdd0534651416094c157e08d175b57543b320d414 ba4dc6be06d62cd95e0d1f5735d5f21f3c6ef3749714f82bc307b6cc38978600
Collatz Hash-512(“The quick brown fox jumps over the lazy dog”)	0x 1955d4a3a215f71b4fc521a09e2759b90b2cd027938f3425cf5ebe969e80f77243f 097262f4ef51733e3c704cbb866e4b914e072a930c2211f83d9aecb23b1cf

5 Security

5.1 One-Wayness or Pre-Image Resistance

This is one of the main properties of a hash algorithm. where a hash function is easy to compute the output for every input but hard to revert from the output to input.

Proving and disproving one-way functions is still an open conjecture. The proving of the existence of a one-way function proves the $P \neq NP$ and disproving proves the $N = NP$. So the best we can do is we can reduce our function to a known problem, and then the hardness of the problem becomes the hardness of reverting the function [12], [11], [13].

So here, if we see the structure of Collatz conjecture in Figure 2, then we notice that every number reaches to 1, but if we see from 1, then there are lot of branches that are denoting some starting x value. So if we have x then reaching to 1 is easy; we just repeatedly do $C(x)$ (define above), then we reach to 1, but if x reach to 1 then going back to real x is really hard because of the branching structure there are lot of possibilities for x from 1. We are taking 16-bits x each time, so the largest number we can represent is 56535 if we exclude 1 itself, then we have 65534 possibility of a number.

The Collatz function is defined as:

$$x \mapsto \begin{cases} x/2, & \text{if } x \text{ is even,} \\ 3x+1, & \text{if } x \text{ is odd.} \end{cases}$$

Reversing this process from a given value m yields two possible predecessor cases:

- **Even reverse:** m may come from $2m$ (always valid).
- **Odd reverse:** m may come from $(m-1)/3$ only if $m \equiv 1 \pmod 3$ and $(m-1)/3$ is an odd integer.

Exploring backwards from 1 therefore produces a tree of candidates. While the *even* reverse is always possible, the *odd* reverse is only occasionally valid, resulting in an average branching factor β in the range of 1.26 to 1.32.

If the original number x reached 1 in k steps (known as the total stopping time), then brute-force reversal requires examining approximately:

$$O(\beta^k)$$

candidates. Empirical studies estimate $k \approx 6.64 \log_2(x)$ for random x, where $\log_2(x)$ is the bit-length b of the input.

Thus, the reverse search complexity becomes:

$$O\big(\beta^{6.64b}\big) \approx O\big((1.3^{6.64})^b\big) \approx O\big((6.46)^b\big)\,,$$

which grows exponentially with b. For large b, this makes naive inversion computationally expensive. However, no formal proof of hardness exists, and potential algorithmic shortcuts cannot be ruled out.

So we believe that reversing the presented hash function is not computationally possible.

5.2 Truncated Birthday Attack

In the presented cryptographic hash algorithm, we have an output size 512/384/256 bit, so for performing birthday attack, we need to compute the hash for $2^{\frac{512}{2}}, 2^{\frac{384}{2}}, 2^{\frac{256}{2}}$ which is computationally not feasible, so we test it, $20 - million$ times for 256-bit digest, $20 - million$ times for 384-bit digest and $20 - million$ times for 512-bit digest, and we didn't find any collision.

Computation performed by the program Collatz-Hash-Birthday-Attack.cpp. In that program there is flexibility to fix input length and max attempts. Characters are choose randomly from charter set define as

$$\text{“}abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789''$$

5.3 Length Extension Attack Resistance

Most of the algorithms based on the Merkle–Damgård construction face length extension attacks like SHA-512, SHA-256, MD-5, etc. In the presented hash algorithm, we introduce a final mixing as explained in figure-5.

In Merkle–Damgård construction, length extension attack is possible because it follows the chaining rule to compute the next hash value, so if anyone has the hash value and knowledge of the message length, then the attacker can continue the chaining and be able to crack the system.

So in Collatz Hash, the first thing is that the internal state is larger than the final output, and we are truncating the internal state and then giving the output, so this thing also avoids the length extension attack see the section 3.4.

Secondly, after computing the digest for the last message block M^n then we again perform final mixing as explained in figure-5, then give the final digest value.

5.4 Differential Analysis

Differential analysis is a standard cryptanalytic technique used to evaluate the sensitivity of a hash function and block cipher to small changes in its input [14] . The goal is to determine whether a single-bit modification in the input leads to a significant and unpredictable change in the hash output, thereby ensuring strong diffusion properties. Let x and x' be two messages differing by a single bit, i.e., $x' = x \oplus \Delta x$, where Δx represents the one-bit difference. The corresponding hash outputs are $H(x)$ and $H(x')$. If the Hamming distance between $H(x)$ and $H(x')$ is close to half the output length, the algorithm satisfies the *avalanche criterion* [[18], [15], [16], [17]].

For our analysis, we implemented a C++ program that performed both differential distribution and avalanche effect testing on the proposed hash function. The input length was fixed at 64 bytes, and we used $100,000$ random samples for differential analysis and 100 trials per input bit for avalanche testing. In each trial, one input bit was flipped while keeping the others constant, and the average probability of each output bit flipping was computed.

The avalanche matrix summary for the first 8 input bits is shown below:

Input Bit	**Average Flip Probability**
0	0.426
1	0.484
2	0.482
3	0.479
4	0.477
5	0.459
6	0.461
7	0.468

An ideal hash function should exhibit an average flip probability close to 0.5, implying that each output bit has a 50% chance of flipping when a single input bit changes. Our results show that the proposed hash function achieves values close to this ideal criterion, indicating strong avalanche properties and resistance against differential cryptanalysis. Furthermore, the absence of low-probability output bits suggests good diffusion and no exploitable patterns in the hash computation.

5.5 Joux's Multicollision Attack

Joux's multicollision attack relies on a fundamental assumption: an iterated hash with a *fixed* compression function F and a *fixed* initialization value (IV). Under these conditions, once a single collision for F is found for a given chaining value, the attacker can ladder such collisions across consecutive blocks to build 2^k distinct messages sharing the same final digest with total work roughly $k \cdot 2^{n/2}$ for an n-bit hash.

The proposed Collatz-based design breaks the "fixed-F" precondition in two distinct ways evident from the implementation:

1. **Message-dependent internal permutation.** Before processing any block, the algorithm derives permutations via `deterministic_shuffle(key, 32)` and `deterministic_shuffle(key, 10)` where `key` is the entire input message string. These permutations steer word selection, rotations, and mixing inside the per-block processing (e.g., accesses using `shuffled32[...]`), making the effective compression mapping depend on the *whole message* rather than being fixed across messages. Consequently, two different candidate messages at the same iteration point are processed by *different* effective compression functions, so the standard multicollision "pairing tree" cannot be constructed.

2. **Message-dependent digest extraction.** After all blocks are processed, the digest is formed by concatenating selected 64-bit buffer words indexed by the message-derived permutation `shuffled10` (i.e., the output uses `buffers[shuffled10[i]]` for $i = 0, \ldots, S/64 - 1$). Thus, even if two messages were engineered to collide on the final chaining state, their differing `shuffled10` (because their messages differ) would map the internal state to *different* output coordinates, preventing a shared final digest through Joux's construction.

For completeness, we note that the scheme otherwise follows Merkle–Damgård-style block processing with MD-like padding to 1024-bit blocks and a fixed IV array of 64-bit words; however, the global, message-dependent permutations injected into both the compression rounds and the finalization step violate the fixed-function assumption essential to Joux's attack. Therefore, the classical multicollision strategy of iteratively splicing colliding block pairs does *not* carry over to this design.

5.6 Resistance to Fixed-Point Attack

A *fixed-point attack* on a hash function aims to find an input message M^* such that after processing all its blocks, the internal state of the hash function returns to its original initialization vector (IV), i.e.,

$$H(M^*) = H(\epsilon)$$

where ϵ denotes the empty message. If such a fixed point exists, it may be reused to build multiple colliding messages or facilitate second-preimage attacks.

In classical Merkle–Damgård constructions with a fixed compression function, fixed points can sometimes be derived algebraically if the compression function has a simple linear or algebraic structure. However, in the proposed Collatz-based hash algorithm, the following properties break the assumptions required for a fixed-point attack:

1. **Message-dependent permutations:** Each round of the algorithm relies on permutations `shuffled32` and `shuffled10` generated deterministically from the entire message. This implies that the effective compression mapping is *message-dependent*, so the same internal state processed under different messages corresponds to different transformations. Consequently, no global fixed point across all messages exists.

2. **Non-linear Collatz-based mixing:** The internal state update involves iterative Collatz-like transformations and modular rotations, which are non-linear and message-dependent. Finding a message that exactly returns the internal state to the initial IV would require solving a large system of nonlinear equations over multiple rounds, which is computationally infeasible.

3. **Finalization step with message-specific output selection:** Even if an adversary could, hypothetically, engineer a message to return the chaining variables to the IV, the final hash value is constructed using a message-dependent selection of buffer words through `shuffled10`. This further prevents the reuse of any internal fixed point to produce a valid hash collision.

Therefore, the proposed algorithm does not admit a practical fixed-point attack in the classical sense because the conditions necessary to exploit the iterative structure of the hash function are deliberately broken by message-dependent permutations and non-linear state updates [19].

5.7 Resistance to Algebraic Attacks

Algebraic attacks on hash functions attempt to exploit simple algebraic structures (e.g., linearity, low-degree polynomial representations) in the compression or round function to model the entire hash computation as a system of algebraic equations. If such equations are sufficiently low-degree or sparse, modern solvers such as *Gröbner basis* methods or SAT solvers can potentially find collisions, preimages, or second preimages more efficiently than brute-force search.

For example, in Merkle–Damgård-based hashes with simple linear XOR and modular addition layers, an attacker might try to represent the full hash function as a multivariate polynomial system over $\mathbb{F}_2$ and then solve it using algebraic cryptanalysis techniques. Successful algebraic modeling requires:

1. A *fixed* and *low-degree* round function,
2. Predictable composition of identical rounds,
3. Absence of data-dependent or non-linear control flow.

5.7.1 Applicability to the Collatz-based hash The proposed Collatz-based hash breaks all three preconditions:

- **Message-dependent round structure:** Each message block generates its own permutations (`shuffled32`, `shuffled10`), so the round transformation is *message-dependent*. Two different messages induce two different algebraic systems, preventing a universal equation system for all inputs.
- **Collatz-based non-linear dynamics:** The inner loop of each round uses the Collatz-like mapping,

$$x \mapsto \begin{cases} x/2, & \text{if } x \text{ even}, \\ (3x+1)/2, & \text{if } x \text{ odd}, \end{cases}$$

  which introduces *data-dependent branching*. The number of iterations and the mixing functions (α or β) depend on the evolving state. This makes it impossible to model the entire round function as a fixed low-degree polynomial.
- **Bit-level rotations and modular operations:** The use of modular rotations and XORs on 64-bit words further raises the algebraic degree of any polynomial representation. After only a few rounds, the degree growth becomes exponential, making Gröbner basis or SAT-based modeling infeasible.

5.8 Time Attack Analysis

A timing side-channel attack becomes possible when the execution time of an algorithm depends on *secret data*. In the Collatz Hash algorithm, the outer loops for message blocks, word mixing, and finalization run for a fixed number of iterations. However, the inner loop implementing the Collatz mapping

$$x \mapsto \begin{cases} x/2 & \text{if } x \text{ is even}, \\ (3x+1)/2 & \text{if } x \text{ is odd}, \end{cases}$$

continues until $x < 2$, so the number of iterations depends on the *Collatz stopping time* of x.

Importantly, the value of x in each round is derived solely from the *public message input*, not from any secret key or hidden state. Hence, any timing variation reflects only the input-dependent dynamics of the Collatz conjecture, which are fully visible to the adversary. Since side-channel attacks require *secret-dependent* timing behavior to leak information, such variations do not reveal additional information beyond what is already known from the public input.

Moreover, for fixed-length inputs, the adversary can simulate the exact Collatz iteration counts offline, so observing execution time provides no advantage in practice. Therefore, **time-based side-channel attacks remain infeasible for the Collatz Hash algorithm**.

5.9 Security of Padding

If message is empty, meaning the input is empty, then Merkle–Damgård strengthening is performed, and first 1 is appended then $o's$ until it reach to 896- bit long then 128-bit message size is appended. This padding help to avoid some basic collision like

Consider a simplified example with a toy hash function having:

- Block size: 8 bits
- Padding: a single '1' bit followed by zeros, without a length field

Let two different messages be:

$$M_1 = \text{"abc"} \quad (24 \text{ bits}), \quad M_2 = \text{"abc"} \parallel \text{"00000000"} \quad (32 \text{ bits})$$

Without the length field in padding, both M_1 and M_2 could result in the same padded block structure, leading to identical intermediate states in the compression function. This would create a trivial collision despite the messages being different. With Merkle-Damgard strengthening, the padding scheme becomes:

1. Append a single '1' bit.
2. Append enough zeros to reach the block boundary minus the length field.
3. Append the original message length in bits as a fixed-size field (e.g., 64 bits in SHA-256).

Now:

$$\text{Length}(M_1) = 24 \quad \text{bits}, \quad \text{Length}(M_2) = 32 \quad \text{bits}$$

The final blocks differ in their length fields, ensuring that M_1 and M_2 produce different padded messages and cannot collide due to block boundary ambiguity alone.

This padding method does not eliminate cryptographic collision attacks entirely, but it prevents trivial structural collisions and is an essential part of secure hash function design.

6 Design Rationale

Collatz Hash is designed to obtain better security, better hardware implementation, and low-cost security evaluation. In the presented design we follow Merkle–Damgård construction, but we propose a completely new compression function that takes a 1024-bit(M^n) message block and previous hash value (H_{i-1}) as input and gives a 512/384/256 bit sequence as output.

Here we design our hash algorithm like the iteration of the loop, and the rotation of bits directly or indirectly depends on the input, so even a small change in the input changes the complete iteration and rotation of the compression function. This thing helps to achieve better collision resistance.

6.1 Why Collatz Conjecture?

If we see the branching structure of Collatz conjecture in figure 4, then we notice that there is always a unique path from x to 1 so there is only one loop documented in Collatz conjecture that $4 \to 2 \to 1 \to 4 \to 2...$ we take care of this loop in our iteration, so it never gets stuck in the loop. And till now there is no good algorithm in the quantum computing field that disproves or proves that it always reaches to 1 or not, but for all the numbers that researchers have tested so far, it is always reaching to 1.

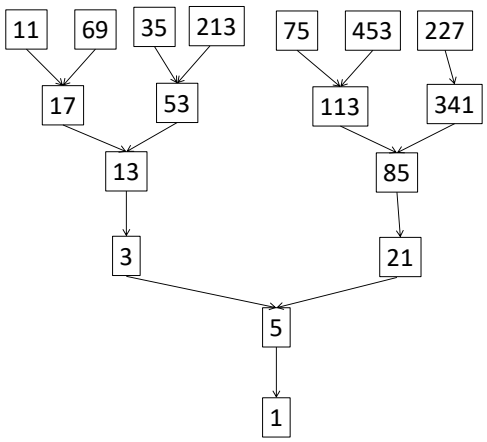

Fig. 7. : Collatz Conjecture Branches

So because of the unique structure for every input, like if we took 11, then we know that $11 \to 17 \to 13 \to 3 \to 5 \to 1$, $69 \to 17 \to 13 \to 3 \to 5 \to 1$, both these paths are different from 11 to 1 and 69 to 1, may be they merge in between, but if we see them end to end, then they are different.

6.2 Round Structure and the Collatz (Conjecture) Loop Effect

The compression routine of the proposed Collatz-based hash proceeds in a small fixed number of *rounds* per message block, but each round internally performs a variable-length sequence of operations driven by a Collatz-like iteration on a 16-bit seed derived from the current working state. This section describes the round count, the per-round Collatz loop, and the impact on diffusion and efficiency.

6.2.1 Round count per block For each 1024-bit message block, the algorithm performs the following principal steps (see source for exact indexing and permutations). After an initial arrangement of 32-bit words into the working array $X[0..15]$ and a short mixing stage, the core update executes a **fixed outer loop of seven iterations** indexed by $w = 0, \ldots, 6$. Inside each outer iteration the algorithm:

1. Extracts a 16-bit value x from a selected word of the working state using a message-derived permutation (code: `x = (X[shuffled32[w] % 16] >> ...) & 0xFFFF`).
2. executes a *Collatz-like inner loop* while $x \geq 2$. If x is even, it is replaced with $x \leftarrow x/2$; if x is odd, it is replaced with $x \leftarrow (3x + 1)/2$. Depending on the parity of the current x the round uses two different nonlinear mixing functions (`alpha` when x even, `beta` when x odd) to update the 32-bit halves of the eight 64-bit buffer words.
3. After the inner loop finishes, the updated halves are recombined to 64-bit words, and a short ARX-style transformation (XORs and rotated versions of two temporaries T_1, T_2) is applied to produce the per-iteration contribution to the 10-word buffer state.

Because the outer loop has seven iterations and each outer iteration executes a variable-length inner loop, the effective number of internal update steps per block is variable but bounded and depends on the 16-bit seed values extracted from the working state. See the implementation for the exact code path.

6.2.2 The Collatz inner loop: behavior and effect The inner loop uses the mapping

$$x \mapsto \begin{cases} x/2, & \text{if } x \text{ is even,} \\ (3x+1)/2, & \text{if } x \text{ is odd,} \end{cases}$$

and repeats while $x \geq 2$. Two important consequences follow:

- **Variable per-round workload.** The number of inner iterations varies with the 16-bit seed x. This yields message-dependent computation effort per round and removes a simple fixed-step model from the compression mapping.
- **Nonlinearity and unpredictability.** Because the branch (even/odd) controls whether `alpha` or `beta` is applied, and the choice of rotation offsets depends on $x \bmod k$ inside the round, the inner loop drives nonlinear, data-dependent mixing that increases diffusion in a highly non-uniform but hard-to-predict way.

6.2.3 Empirical stopping-time statistics (16-bit seeds) The 16-bit seed used by the inner loop is formed by extracting 16 bits from the working state (see code). To quantify the typical loop length, we sampled *all* initial seeds in the range $0 \ldots 2^{16} - 1$ and counted iterations of the exact inner rule above (stop when $x < 2$). The measured statistics are

- **Average inner-loop iterations:** ≈ 69.0
- **Median:** 64
- **Maximum observed:** 214

These numbers come from the same Collatz-like transition used in the implementation and indicate that, on average, each outer iteration executes roughly 69 inner steps for a uniformly distributed 16-bit seed. Combined with seven outer iterations, a typical block performs on the order of $7 \times 69 \approx 480$ inner Collatz-driven updates (plus the inexpensive ARX/XOR steps and recombination for every inner iteration). The maximum-case per-block inner-step count (across all seeds) was observed to be 214 per outer iteration, implying an absolute worst-case (per block) of about $7 \times 214 = 1498$ inner iterations. These empirical numbers were computed directly from the inner loop rule used in the source implementation.

6.3 Implications for diffusion and efficiency

Diffusion & security

- The message-dependent seed selection together with the variable-length Collatz inner loop makes the per-block compression function effectively *message-dependent* (the exact sequence of α/β updates, rotation offsets and word indices depends on the seed trajectory). This increases resistance against simple differential laddering and fixed-round analytic attacks because the attacker cannot assume a fixed deterministic sequence of operations for two different messages at the same iteration point.
- The repeated application of non-linear 32-bit updates (α, β) interleaved with 64-bit ARX-like mixing steps promotes bitwise diffusion: even a small change in input tends to cause different parity patterns for the seed x and thus diverging inner-loop behaviors across messages.

Efficiency

- The variable inner-loop cost is a trade-off: it increases per-block computational work on average (roughly a few hundred inner updates per block, per above), which is acceptable for many security-focused applications but must be considered when assessing throughput.
- The worst-case bound is finite for 16-bit seeds (empirically ≤ 214 iterations per outer iteration in our sample). Thus, the algorithm does not have an unbounded runtime, but adversarial message construction could aim to produce seeds with large stopping times to increase cost.

7 Advantages and Comparison

The proposed Collatz-based hash algorithm introduces several unique design features that distinguish it from conventional hash functions such as SHA-2, SHA-3, or BLAKE2. These features arise primarily from the use of *message-dependent permutations*, *Collatz-inspired variable-length mixing loops*, and *non-linear diffusion* integrated into each compression round.

7.1 Advantages over conventional hash algorithms

1. **Message-dependent internal structure:** Traditional hash functions such as SHA-2 use a fixed sequence of round constants and operations for all inputs. In contrast, the Collatz-based design derives internal permutations (`shuffled32`, `shuffled10`) directly from the input message, so the effective per-round compression mapping is *message-dependent*. This breaks assumptions required for standard multicollision attacks (e.g., Joux's attack) and complicates differential path construction since no single fixed round function applies across all inputs.

2. **Collatz-inspired variable-length loops:** Each round includes a variable-length inner loop governed by the Collatz-like rule

$$x \mapsto \begin{cases} x/2, & x \text{ even}, \\ (3x+1)/2, & x \text{ odd}, \end{cases}$$

   with non-linear update functions α and β triggered by parity. This data-dependent branching ensures that small input changes affect both the number of inner iterations and the mixing pattern, thereby enhancing the avalanche effect compared to fixed-round designs such as SHA-2.

3. **Resistance to fixed-point and laddering attacks:** Since both the number of iterations and the rotation offsets inside each round depend on the evolving message state, attackers cannot construct predictable fixed points or ladder collisions across multiple blocks as in Merkle–Damgård-based designs.

4. **High diffusion with simple arithmetic:** The combination of Collatz branching, modular rotations, and XORs ensures that each output bit depends in a complex way on many input bits, achieving strong diffusion with low implementation complexity (no large S-boxes or lookup tables).

7.2 Compression efficiency and trade-offs

The Collatz-based algorithm processes input in 1024-bit blocks with a fixed number of outer rounds (seven per block) but variable-length inner iterations per round. Empirically, for 16-bit seeds extracted from the message state:

- Average inner iterations per round: ~ 69
- Median: 64
- Maximum observed: 214

Thus, a typical block requires roughly $7 \times 69 \approx 480$ inner iterations, which is higher than the fixed 64 rounds of SHA-2 but provides significantly more message-dependent diffusion per block. Unlike SHA-3, which achieves security through wide-pipe sponge construction, the Collatz-based design leverages variable-time nonlinear dynamics to break structural symmetries.

7.3 Comparison summary

Table 2 summarizes key differences between the proposed algorithm and representative conventional hash functions.

Table 2. Comparison of the Collatz-based hash with conventional hash algorithms.

Feature	**Collatz Hash**	**SHA-2**	**SHA-3 / BLAKE2**
Block size	1024 bits	512 bits	1600-bit state (SHA-3)
Rounds per block	7 outer, var. inner	64 fixed	24 fixed (SHA-3)
Message-dependent perms	Yes	No	No
Variable-time mixing	Yes (Collatz)	No	No
Non-linear dynamics	High	Moderate	High (Keccak S-box)
Attack resistance (Joux, fixed points)	Strong	Weaker (MD-based)	Strong (sponge)
Implementation cost	Moderate	Low	Moderate

Overall, the Collatz-based hash trades some computational efficiency for significantly increased diffusion, message-dependent variability, and resistance to certain structural attacks inherent in classical Merkle–Damgård constructions.

8 Conclusion

In this document, we proposed the Collatz hash algorithm, which follows the Merkle-Damgård construction, and we designed our own compression function. Our analysis shows that Collatz hash is secure and can be used for password storing, PRNG, etc. However, the extensive security analysis of any new design requires a lot of effort from many researchers. We thus invite and encourage researchers to analyze the security of Collatz hash. Collatz hash is not covered by any patent, and Collatz hash is freely available.

9 Complete Implementation

The complete source code and implementation details for the proposed hash algorithm, experimental setup, and analysis scripts are publicly available at:

GitHub Link: `https://github.com/shaurya010/Collatz-Hash__Hash-Algorithm-Using-Collatz-Conjecture`

10 Future Work

This paper introduced the Collatz Hash-512/384/256 algorithm. In future publications, we intend to deploy this algorithm within a Pseudorandom Number Generator (PRNG) framework and provide its corresponding randomness proof.

References

1. A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1996. Available: `https://theswissbay.ch/pdf/Gentoomen%20Library/Cryptography/Handbook%20of%20Applied%20Cryptography%20-%20Alfred%20J.%20Menezes.pdf`
2. R. C. Merkle, *Secrecy, Authentication, and Public Key Systems*, UMI Research Press, 1979.
3. R. C. Merkle, "One Way Hash Functions and DES," in *Advances in Cryptology — CRYPTO'89*, G. Brassard, Ed., Lecture Notes in Computer Science, vol. 435, Springer-Verlag, 1990, pp. 428–446.
4. M. O. Rabin, "Digitalized Signatures," in *Foundations of Secure Computation*, R. Lipton and R. DeMillo, Eds., Academic Press, New York, 1978, pp. 155–166.
5. I. B. Damgård, "Collision Free Hash Functions and Public Key Signature Schemes," in *Advances in Cryptology — EUROCRYPT'87*, D. Chaum and W. L. Price, Eds., Lecture Notes in Computer Science, vol. 304, Springer-Verlag, 1988, pp. 203–216.
6. I. B. Damgård, *The Application of Claw Free Functions in Cryptography*, Ph.D. thesis, Aarhus University, Mathematical Institute, 1988.
7. T. Tao, "Almost all orbits of the Collatz map attain almost bounded values," *arXiv preprint arXiv:1909.03562* (revised version v5, Feb152022).
8. J.-P. Allouche, "Sur la conjecture de Syracuse-Kakutani-Collatz," *Séminaire de Théorie des Nombres*, 1978–1979, Exp. No. 9, 15 pp., CNRS, Talence, 1979.
9. R. E. Crandall, "On the '3x+1' Problem," *Mathematics of Computation*, vol. 32, 1978, pp. 1281–1292.
10. R. Terras, "A Stopping Time Problem on the Positive Integers," *Acta Arithmetica*, vol. 30, 1976, pp. 241–252.
11. Leonid A. Levin, "The Tale of One-Way Functions," arXiv:cs/0012023v5 [cs.CR], August 2003. Available online: `https://arxiv.org/pdf/cs/0012023v5`
12. Alexander Russell, "Necessary and sufficient conditions for collision-free hashing," *Journal of Cryptology*, vol. 8, pp. 87–99, March 1995. DOI: 10.1007/BF00190757.
13. R. Impagliazzo, L. A. Levin, and M. Luby, "Pseudo-random generation from one-way functions," in *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing (STOC)*, pp. 12–24, 1989.
14. E. Biham and A. Shamir, "Differential Cryptanalysis of DES-like Cryptosystems," *Advances in Cryptology – CRYPTO'90, Lecture Notes in Computer Science*, vol. 537, pp. 2–21, Springer-Verlag, 1991.
15. F. Chabaud and A. Joux, "Differential Collisions in SHA-0," *Advances in Cryptology – CRYPTO'98*, pp. 56–71, Springer-Verlag, 1998.
16. E. Biham and R. Chen, "Near-Collisions of SHA-0," *Advances in Cryptology – CRYPTO 2004*, pp. 290–305, Springer-Verlag, 2004.

17. X. Wang, X. Lai, D. Feng, H. Chen, and X. Yu, "Cryptanalysis of the Hash Functions MD4 and RIPEMD," *Advances in Cryptology – EUROCRYPT 2005*, pp. 1–18, Springer-Verlag, 2005.
18. X. Wang and H. Yu, "How to Break MD5 and Other Hash Functions," *Advances in Cryptology – EUROCRYPT 2005*, pp. 19–35, Springer-Verlag, 2005.
19. I. Dinur and A. Shamir, "Cube Attacks on Tweakable Black Box Polynomials," *IACR ePrint Archive*, 2008. Available at `http://eprint.iacr.org/2008/385`.
20. National Institute of Standards and Technology (NIST), "Secure Hash Standard (SHS)," FIPS PUB 180-4, U.S. Department of Commerce, August 2015. Available at `https://csrc.nist.gov/publications/detail/fips/180/4/final`.
21. R. L. Rivest, R. Anderson, and E. Biham, "Tiger: A Fast New Hash Function," *Fast Software Encryption (FSE 1996), Lecture Notes in Computer Science*, vol. 1039, pp. 89–97, Springer-Verlag, 1996.
22. N. Alamgir, S. Nejati, and C. Bright, "SHA-256 Collision Attack with Programmatic SAT," *arXiv preprint*, Jun. 2024.
23. T. Hatanaka, R. Fushio, M. Watanabe, W. J. Munro, T. N. Ikeda, and S. Sugiura, "A Quantum-Resistant Photonic Hash Function," *arXiv preprint*, Sep. 2024.