# Cryptanalysis of SHA2 based on Perturbation Technique

**Munawar Hasan**

**Thesis Supervisor: Dr. Somitra Kumar Sanadhya**

Thesis submitted to the Indraprastha Institute of Information Technology
in partial fulfillment of the requirements
for the award of
Master of Technology

Cryptographic Research Group
Indraprastha Institute of Information Technology
New Delhi - 110020

# Contents

# Chapter 1

# Introduction

A cryptographic hash function is a mathematical algorithm that maps arbitrary size data, called the message to a fixed size value, called the hash or message digest or digest. The hash so obtained is one way and computationally hard to invert. Hence, for a given hash value; it is computationally infeasible to retrieve its message. The security of any hash function is given by the degree of its resistance to following properties - *pre-image resistance*, *second pre-image resistance* and *collision resistance*. Additionally, a hash function should be fast and efficient.

A hash function is pre-image resistant if it is computationally infeasible to obtain the corresponding message, $m$ from the given hash value $h$ such that $Hash(m) = h$ i.e., given a hash $h$; it is difficult to find a value $m$ that hashes to $h$. In mathematical terms, given the range value ; it is difficult to find the domain point which hashes to the given range value. For a second pre-image resistant hash function, given a message $m_1$, it is computationally hard to obtain another message $m_2$ such that they both produce the same hash value $h$. In other words, given a message $m_1$, it computationally hard to find $m_2 \neq m_1$ such that $h(m_1) = h(m_2)$. In a collision resistant hash function, it is computationally hard to find two messages that hash to same value, i.e. it is hard to produce two different inputs (say $m_1$ and $m_2$) such that they have the same hash value. One notable property is that the hash size is fixed and is much smaller that the domain. Therefore, theoretically, it is impossible to prevent the preimage, second preimage and collision attacks on any hash functions. *Figure 1.1* shows the inequality in the size of the range and the domain. The computational hardness of problems in hash function domain depends on the amount of computation that is needed to break the various properties discussed previously. Since the output size of a hash function is a fixed number $2^n$ (assuming that the hash function has $n$ bit output), it is easy to show that a generic preimage attack can be constructed for an attacker who can call the hash function $2^n$ times. Similarly, a collision attack can succeed with probability more than 50% if the attacker can call the hash function more than $2^{n/2}$ times. Finally, a generic second preimage attack exists for an attacker who can call the hash function $2^n$

times. Thus, it is no surprise that an attacker who has the power to call a hash function as many times as mentioned earlier can break any or all of the security properties of the hash function.

The security of a hash function is therefore defined in terms of the computational power of an adversary. If an adversary who makes lesser than $2^{n/2}$ calls to an $n$-bit hash function can produce a collision for the hash function then we call this function to be broken with respect to its collision resistance. Similar arguments can be given for preimage and second preimage attack resistance as well.

For example, the hash function SHA-1 produces 160 bit digests. With $2^{80}$ calls to it, one should expect to find collisions with high probability. However, a shortcut attack requiring only $2^{69}$ calls to the hash function was shown by X. Wang and her team. However, this attack has not yet been demonstrated in practice due to the high number of calls to the hash function. Attacks like this are called "theoretical attacks" and attacks which can be exhibited in practice are called "practical attacks". Hash functions MD4, MD5, and SHA-0 suffer from practical collisions attacks, for example.

Further, it is also desirable that a hash function is secure from any length extension attacks. In a length extension attack, an attack uses the previous instances of the computation and generates the new computation by simply adding one or more new message blocks. This attack does not breaks any basic properties of hash function but violates another basic property of hash function i.e. *randomness*, according to which a hash function must produce random output even a one bit change is done in the message. A compression function is used at the end to prevent this attack. *Figure 1.2* shows the compression function diagrammatically.

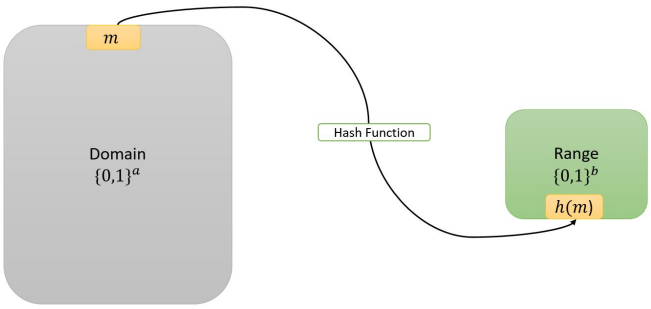


Figure 1.1: Size of domain and the range of Hash Functions

## 1.1 Applications of Hash Functions

Following are few domains where hash functions are used:

- Public Key Algorithms
  - Password Logins
  - Encryption Key Management
  - Digital Signatures
- Integrity Check
  - Virus and Malware scanning
- Authentication
  - Secure Web Connection (PGP, SSL, SSH, S/MIME)

## 1.2 SHA2

The SHA2 is a cryptographic hash function designed by the National Institute of Standards and Technology (NIST). SHA2 is the successor of SHA1. It describes an improved version of algorithm which brings more security as compared to SHA1. SHA2 works on eight 32 bit words. Following are the hash sizes of SHA2 - *SHA-224*, *SHA-256*, *SHA-384* and *SHA-512*. We are interested in study of *SHA-256* which produces a hash size of *256 bits*.

SHA2 is based on Merkle-Damgård (MD) construction. The MD construction was first described in [5]. It was independently proven in [6] and [2] that the structure is secure and collision resistant if appropriate padding scheme is used and the compression function is collision resistant. *Figure 1.2* describes MD the design principle:

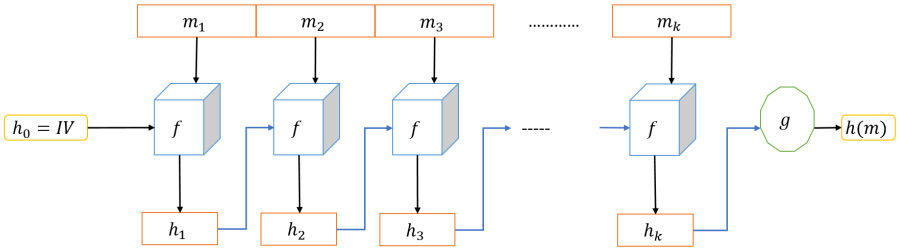


Figure 1.2: Merkle-Damgård design for hash functions

The computation of hash is based on repeated application of some mathematical function, $f$ in *Figure 1.3*. *Figure 1.4* describes the iteration of the hash

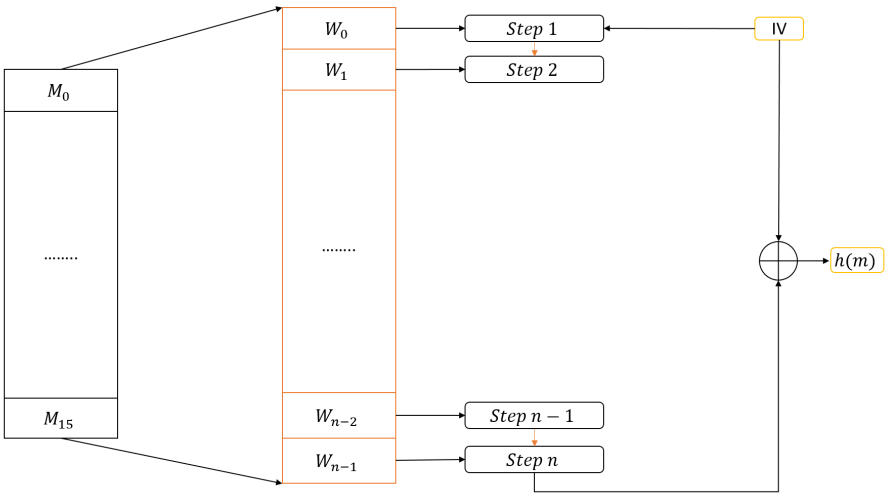


Figure 1.3: Iteration based hash function

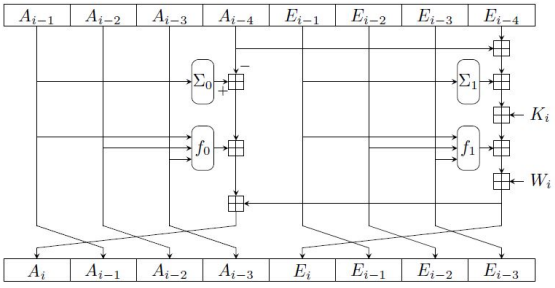


Figure 1.4: SHA2 Step Function

for SHA2. As clear from the figure that the original message is first expanded and then step transformation is performed on the expanded message. The final value or the *digest* is the *xor* of $IV$ and $n^{th}$ step.

As our interest lies in analysis of *SHA256*, hence from hereon we refer SHA2 to *SHA256*. Further we describe the internal structure and bit manipulation of *SHA256* only omitting the details of other variants.
*Figure 1.4* shows the details of single step of SHA2. Each step function or step transformation operate on 8 state variables each of which is 32 bits; $A_i$, $B_i$, $C_i$, $D_i$, $E_i$, $F_i$, $G_i$ and $H_i$, where $i$ denotes the $i^{th}$ step. The initial value of the state variables are given by the initialization vector, $IV$. The $IV$ are predetermined and are listed below:

$\langle A_0 = 0x6a09e667 \rangle$
$\langle B_0 = 0xbb67ae85 \rangle$

$\langle C_0 = 0x3c6ef372 \rangle$
$\langle D_0 = 0xa54ff53a \rangle$
$\langle E_0 = 0x510e527f \rangle$
$\langle F_0 = 0x9b05688c \rangle$
$\langle G_0 = 0x1f83d9ab \rangle$
$\langle H_0 = 0x5be0cd19 \rangle$

As stated above and shown in figure; calculation of hash consists of two phases, the message expansion and the step transformation. In case of SHA2, theses phases are described as follows:

1. Message Expansion
   There are 16, $W_0$ to $W_{15}$ base message words in SHA2. The rest 64 message words are obtained from the base using the following recurrence relation:

   $$W_i = \begin{cases} W_i & \text{for } 0 \leq i \leq 15, \\ \sigma_1(W_{i-2}) + W_{i-7} + \sigma_0(W_{i-15}) + W_{i-16} & \text{for } 16 \leq i \leq 63 \end{cases}$$

   where
   $\sigma_0(x) = ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x)$
   $\sigma_0(x) = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x)$

2. State Update
   The state update transformation takes place in span of 64 steps (0 to 63) using the message word and the round constant $K_i$. We have used alternative description of SHA2 as described in [4]. In this description there are only two updates namely the $A_i$ and the $E_i$; rest all the state variables takes the values from these two in form of chaining input, due to this chaining value of state update function, we can represent all the state variables in terms of $A$ and $E$ as described below:

   $A_i = A_i$, $B_i = A_{i-1}$, $C_i = A_{i-2}$, $D_i = A_{i-3}$, $E_i = E_i$, $F_i = E_{i-1}$, $G_i = E_{i-2}$ and $H_i = E_{i-3}$
   SHA2 uses two boolean functions as described below:

   - $f_0(x, y, z) = (x \wedge y) \oplus (x \wedge y) \oplus (y \wedge z)$
   - $f_1(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z)$

   SHA2 also use two auxiliary functions given below:

   - $\sum_0(X) = (X >>> 2) \oplus (X >>> 13) \oplus (X >>> 22)$
   - $\sum_1(X) = (X >>> 11) \oplus (X >>> 6) \oplus (X >>> 25)$

   From the *Figure 1.4* it clear that step transformation is dependent only on two message words, $A_i$ and $E_i$. Any other words, $B$, $C$, $D$, $F$, $G$ and $H$ can be obtained by knowing previous value of $A_i$ and $E_i$ .i.e. $A_{i-1}$ and $E_{i-1}$.

## 1.3 History and Related Work

There are very few theoretical references explaining the full round differential trail of the *SHA256* while the practical ones haven't reached full round differential. There are two different approaches of performing the attack on any hash function. The *local collision* technique has provided some major break through in studying the attack on hash functions. In [3] and [7], the author showed how local collisions can be used for attacking iterated hash functions. These works are centered around the study of non-linearity of message expansion and its correlation in subsequent steps. Using these technique *Sanadhaya and Sarkar* formulated *SS local collision* technique and produced 21 round collision of SHA2 in [8] which was later improved and increased to 24 rounds [9]. Using similar concept, a 24 round attack was also given in [7]. It was computationally hard to increase the rounds using this technique. These attacks were based on combinatorial analysis of internal functions of SHA-2. Due to the further complexity and cost of the internal structure, the $\sigma_0$ and the $\sigma_1$, extension to more rounds using this approach was not possible.

The second technique is based on the disturbance vector. *Canniere and Rechberger* in [1] showed the trails on SHA1. They described a generalized characteristic of SHA1 and presented the trail using the automation. This technique was later used by *Mendel et.al* [4] and a 28 round attack was presented on SHA2. This attack was later improved and increased to 46 rounds.

We found that though work done by *Mendel et.al* was considerable and remarkable yet it lacked an efficient back tracking scheme. During the last few years there has been lots of improvements in the implementation and also the research has shifted more on the finding of the impossibility conditions that occur in SHA2 during the generation of automated trail. These various research and their formulation do provide some improvements on the previous ones but none of them is able to define all the set of conditions for generating the automated trails of SHA2.

## 1.4 Motivation

Our aim is to study the second technique for the attack on hash function and come up with an open source implementation that provides a framework for generating automated trails. There are very few implementation on generating the automated trails of *SHA256*, hence it becomes our motivation to work on this area.

We examine the paper [1] which is completely based on SHA-1 trails. SHA2 is more complex than SHA-1. There are several conditions in SHA2 that causes inconsistency behaviour in generating trails. Several research are still on going over this field and new conditions are being discovered. Recently, one such condition called the "*double bit condition*" was published in [4]. There seems to be no standard approach or any pre-defined framework to generate the automated trails of SHA-256 and hence it still remains open problem and interesting topic

of research. Our aim is to formulate an open source implementation for finding trails that is able to handle the inconsistent values automatically.

# Chapter 2

# Collision Attack on Reduced Round SHA2

Our problem specification begins on the footprints of SHA-1. We carry forward the described sub-routine in [1].
We begin by defining basic foundations of the implementation into following factors:

- Design of algorithm
- Formulation of algorithm into the data structure

The design of algorithm is the first step in any implementation task. Further as our implementation is computationally intensive and lot of backtracking is inherent, we need to incorporate these factors into our algorithm. The data structure must also be redesigned and should incorporate the condition described in the algorithm. We begin by describing the algorithm and discussing constraints on data structure. Later we show *non-random* properties of the $\sigma$ function and its graphical plot. Analyzing all these things, in *section 2.4* we start formulating the trails. Later in *section 2.6*, we provide complete analysis of various data structure used and studied. Then in *section 2.6.5*, we present the best data structure for our work. Then we present the description of the implementation, the algorithm and the generated trails from the algorithm. We research on following points that are discussed in this chapter:

- Overview of algorithm design
- Overview of data structure
- Non-Randomness in $\sigma$
- Strategy for developing differential trail
- Design and Analysis of Data Structure

- Description of the implementation
- Theoretical bounds
- Precomputation
- Algorithm

## 2.1 Algorithm Design

The algorithm design is based on satisfying two conditions namely the two bit condition and the consistency check. The consistency check is done randomly over several steps but two bit condition is done at each and every step. The algorithm must fill each value that is consistent and correct value. If this is not the case, then backtracking occurs that removes the conflicting values and populates the correct one.
Few helper sub-routines are also used like the depth first search etc. Further, there is also lot of auxiliary manipulations and looksups like the bit vector etc. Detail algorithm is presented in *section 2.10*.

## 2.2 Data Structure

Choice of data structure is very important for any computation based task. The choice is formulated on the type of computation and its limitations. Most often we are required to make decisions on the acceptance of values or whether to drop the value or rectify the value. This decision making involves frequent lookups on the previous values. Also, a single value is capable of generating a trail and therefore providing a path. Henceforth, we use a tree data structure for implementing the algorithm.
The tree data structure makes our few operation very efficient like the establishment of path (hence, giving us the trails), previous value lookups (using the parent of the node) and the backtracking (using the parent and the siblings).
Thus, from here on we discuss and analyze the tree data structure only.

The algorithm needs complete redesign and redesign it to suit the SHA-2. We perform several tests on different data structures We formulate several conditions and design our approach into set of algorithms as described in *section 2.10*.

## 2.3 Non-Randomness in $\sigma$

### 2.3.1 Analysis of $\sigma_0$

The analysis of $\sigma_0$ shows that the range of values of $\sigma_0$ when plotted over the entire domain of word W, takes a periodic sequence. Any simple c-code, like one as mentioned below can be used to generate the values. The plot is shown

in figure 2.

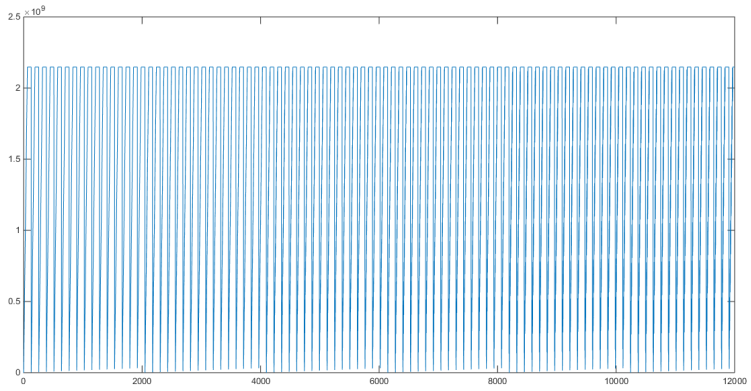


Figure 2.1: $\sigma_0(\omega)$

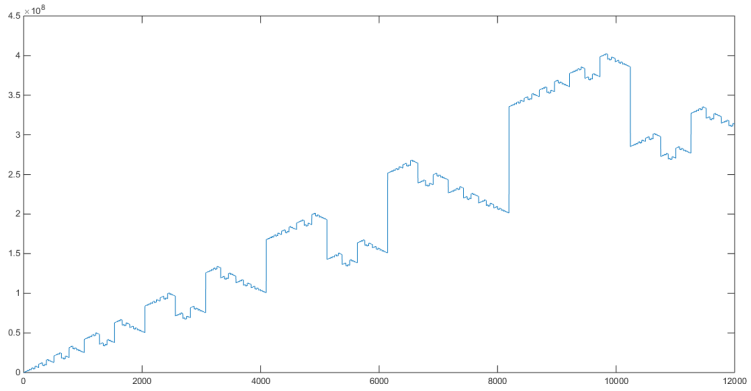


Figure 2.2: $\sigma_1(\omega)$

**c-snippet for $\sigma_0$**

```
#define ROTRIGHT(a,b) (((a) >> (b)) | ((a) << (32-(b))))

#define SIG0(x) (ROTRIGHT(x,7) ^ ROTRIGHT(x,18) ^ ((x) >> 3))
#define SIG1(x) (ROTRIGHT(x,17) ^ ROTRIGHT(x,19) ^ ((x) >> 10))


int main(){
    unsigned int sig_0;
    unsigned int W;
    unsigned int i;

    for(W = 0; W < LIMIT; W++)
        sig_0 = SIG0(W);
}
```

*Figure 2.1* shows the graph representing first 12000 values of $\sigma_0(W)$ on Y-axis and $W$ on X-axis. From the plot, we can infer that whatever value we take on the Y-axis .i.e. $\sigma_1(W)$, we will get solution with probability one, that is we will get an intersection on the graph when drawn a line from the chosen point.

Given $\delta$, number of solution pairs $(\omega, \alpha)$ satisfying
$\sigma_0(\omega+\alpha) - \sigma_0(\omega) =$ (number of times the *upper line* cuts the curve) $*$ (number of times the *lower line* cuts the curve)
where $\delta$ is taken from Y-axis and $\alpha$ from X-axis.
Let $\sigma_0(\omega + \alpha) - \sigma_0(\omega) = \Delta$
Further, once we fix a $\delta$, the number of solutions of the above equation do not differ to much from $\Delta$.

### 2.3.2 Analysis of $\sigma_1$

In the similar way, we can show the analysis of $\sigma_1$. Below is the C code similar to $\sigma_0$. *Figure 2.2* shows the plot of values. Similar analysis as that of $\sigma_0$ can be done here.

**c-snippet for $\sigma_1$**

```
#define ROTRIGHT(a,b) (((a) >> (b)) | ((a) << (32-(b))))

#define SIG0(x) (ROTRIGHT(x,7) ^ ROTRIGHT(x,18) ^ ((x) >> 3))
#define SIG1(x) (ROTRIGHT(x,17) ^ ROTRIGHT(x,19) ^ ((x) >> 10))


int main(){
```

```
    unsigned int sig_1;
    unsigned int W;
    unsigned int i;

    for(W = 0; W < LIMIT; W++)
        sig_1 = SIG1(W);
}
```

### 2.3.3 Differential Analysis of $\sigma_0$

We perform local analysis of $\sigma_0$ by running the simulation on random $\delta$ for each of the message values. The equation can formulated as below:

$$\sigma_0(W + \delta) - \sigma_0(W) = \Delta$$

The distribution is almost random and non-conclusive. The below plot in figure 4, shows the values of the corresponding equation. The x-axis represents first 16000 values while the y-axis solves the equation - $\sigma_0(W + \delta) - \sigma_0(W) = \Delta$

### 2.3.4 Differential Analysis of $\sigma_1$

The differential analysis of $\sigma_1$ could also be performed in the similar way. The corresponding equation is:

$$\sigma_1(W + \delta) - \sigma_1(W) = \Delta$$

The distribution is seems similar to the $\sigma_0$ plot. In the *Figure 2.4*, the values shown for the corresponding equation. The x-axis represents first 16000 values while the y-axis solves the equation - $\sigma_1(W + \delta) - \sigma_1(W) = \Delta$

## 2.4 Building Characteristics

The building of characteristic for SHA2 can be demonstrated using following flow chart in *Figure 2.5*:

Following points describes the flowchart:

- Choose a random message pair $(m, m^*)$
- Build differential characteristic of the chosen message pair $(m, m^*)$
- Store the intermediate results in augmented data structures and define the time based heuristics, $T_h$
- Drill the search for the conforming message to the SHA2 rounds limited by $T_h$.

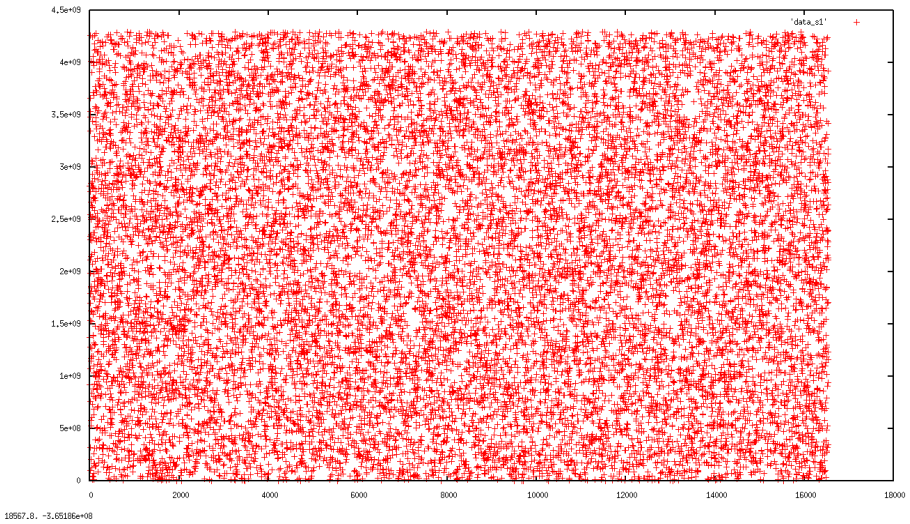


Figure 2.3: Differential Analysis of $\sigma_0$

- Once $T_h$ expires, perform backtracking

Two Bit Condition and Cyclic Anomaly in SHA2: Conditions of type $(A_{(i-2),j} = A_{(i-3),j}) \wedge (A_{(i-2),j} \neq A_{(i-3),j})$ are called two bit conditions. Such condition occur very frequently in SHA2. They have a deep impact on the search heuristics. As setting of one bit affects a total of 4 steps,, we must carefully mine out two bit condition and fix it. Conditional branching spanning over 5 steps for each set bit should be sufficient. Though this would decrease the searching efficiency but is vital in generating valid differential characteristic.

Conditional Checks: Checking each bit and assigning values becomes complete test for generating the differentials. This process becomes very costly and hence some times abrupt omissions are done. Each equal bit bit is assigned with value 0 or 1. Each unequal bit is assigned a value $u$ or $n$ and otherwise (check table 1).
To define the algorithm for conforming message words, let us first define set all possible values that a message difference can take (Table 1).

## 2.5 Strategy and Initial Setup

### 2.5.1 Starting Pair

For getting the differentials extending over 24 steps we need to choose the initial characteristic spanning between $9 \leq t \leq 16$. At $t = 16$, the attack can be

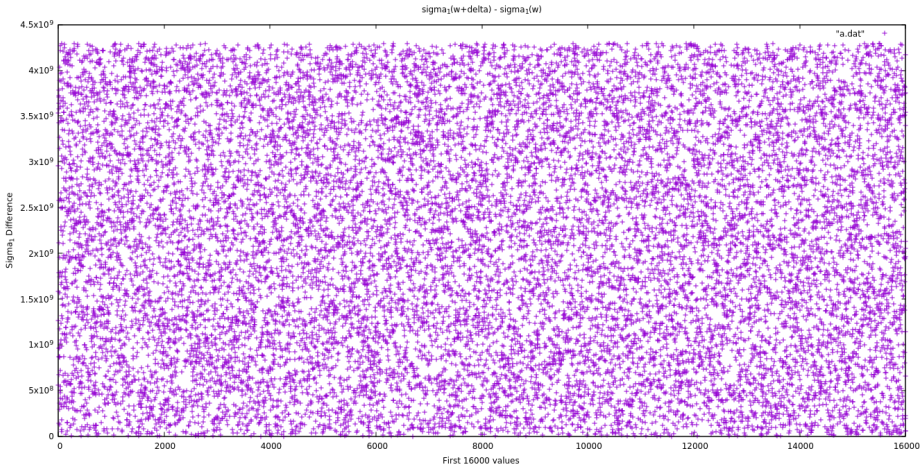


Figure 2.4: Differential Analysis of $\sigma_1$

Table 2.1: Set of all possible condition on a pair of bits

| $(x_i, x_i^*)$ | (0, 0) | (1, 0) | (0, 1) | (1, 1) |
|---|---|---|---|---|
| ? | ✓ | ✓ | ✓ | ✓ |
| - | ✓ | - | - | ✓ |
| x | - | ✓ | ✓ | - |
| 0 | ✓ | - | - | - |
| u | - | ✓ | - | - |
| n | - | - | ✓ | - |
| 1 | - | - | - | ✓ |
| # | - | - | - | - |
| 3 | ✓ | ✓ | - | - |
| 5 | ✓ | - | ✓ | - |
| 7 | ✓ | ✓ | ✓ | - |
| A | - | ✓ | - | ✓ |
| B | ✓ | ✓ | - | ✓ |
| C | - | - | ✓ | ✓ |
| D | ✓ | - | ✓ | ✓ |
| E | - | ✓ | ✓ | ✓ |

extended to 32 steps. Any starting point will begin by fixing the value of $t$. Once the span is fixed then we try to choose the message word that makes the SHA2 rounds consistent. One more metric for choosing the message word is the

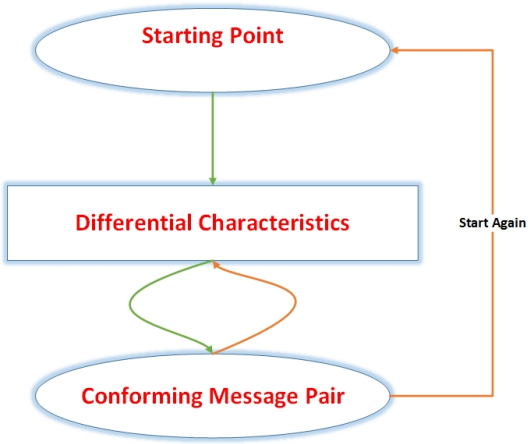


Figure 2.5: Flowchart

sparsity of the message word so that the search time is greatly reduced. Further, the higher the value of $t$, more difficult it becomes to choose a good message word.

### 2.5.2 Valid Differentials and Conforming Pair

For a given starting point, we perform the state update together with the message expansion. During this phase if we get any inconsistency we move a step backward and fix a bit to remove the consistency. The huge amount of inconsistency in SHA2 is due to the update of two state variables together with the diffusion of $\Sigma_i$. A generalized framework cannot be determined which covers all the inconsistent properties. The idea is to put checkpoints on most common factors (that cause inconsistencies) that occur in SHA2 like the two bit condition, cyclic dependency and the complete condition check. These three checks seems sufficient to generate a valid trail. Due to the high probability of inconsistent behavior of the differentials, merely creating differentials will not help. A strategy for searching conforming message pair has to be to combined to the above process. Doing this will detect critical bits (the bits that cause two bit condition and hence the critical bits).

### 2.5.3 Improving Computation

We divide the $W_i$, $A_i$ and $E_i$ into 3 sub steps. We take each individual bit of $\sigma_i$, $\Sigma_i$ and $f_i$ and perform the modular additions. The bit computation reduces to 3.
Memoization: While we are creating the trail of differential, we can use the previous computed values in backtracking. Hence, we need to store the trails in some hash map or some fast data structure. These values are very big and

provides an upper bound on the stored value.

## 2.6 Design and Analysis of Data Structure

Selecting the most appropriate data structure for the implementation of the cryptanalytic technique is very important. The should be an optimal solution between the storage, retrieval and updates. Hence, in our current problem domain following points represents a metric for choosing any data structure:

- Insertion and Deletion
- Retrieval
- Space complexity
- Update

Further, in any cryptanalytic technique the trails provides a path from the root node to the leaf node. Each node is dependent on its parent and its ancestors and thus our data structure must additionally put following constraints to handle these relations:

- It must be able to represent the relationship between different trails and among the individual values. For example - a value is dependent on the parent value which itself is dependent on several previous values. These value forms a path starting from initial random chosen value. Hence, a tree data structure would be appropriate to represent such scenario.
- It must be simple so that the data can be processes easily whenever required.

Let us study some data structure in detail and check its applicability for our application. As already mentioned above about the relevancy of tree data structure, we are going to study and formulate our problem domain on tree data structure.

### 2.6.1 Binary Search Tree

*Binary Search Tree* or *BST* is a lightweight data structure. The design of *BST* depends on the assumption that the data inserted is random in nature. This assumption is often not correct and the resulting *BST* may become skewed to some degree. If we consider the *BST* created via random permutation, the total number of *BST* is given below.

**Number of *BST* on a given number of nodes, '*n*'**

*Let there be n nodes to construct a BST.*
*At a given instance, let i be the root of the BST.* Thus, there are two subtrees spanning over *(n-i)* and *(i-1)*. As $1 \leq i \leq n$, the span of $i$ takes all the possibilities recursively defines by both the *sub-BST's. Thus, the recurrence relation is given by the following equation:*

$$T(n) = T(n-i) * T(i-1)$$

*Summing i over n nodes gives us the total number of BSTs*

$$T(n) = \sum_{i=1}^{i=n} T(n-i) * T(i-1)$$
$$T(n) = \frac{\binom{2n}{n}}{(n+1)}$$

The above expression is called the $CatalanNumber$.

In our area of applicability, the construction of trails occurs by analyzing and changing few bits. These results in values that are not random and some times very nearer and thus makes the *BST*, a skewed one.
We perform an experiment to prove our claim. We generate a random number to mimic the message word pair. Then we try to generate differential trail by making one bit change. The bit flip is repeated 10 times and the generated values are populated into a *Binary Search Tree. Figure 7* shows the plot of such trail. From the observation it is obvious that the value is highly skewed. We insert the generated values into a *BST* using following C - code in *List 1.*
The analysis shows that the values are populated first in increasing order and then in decreasing order, there by creating a skewed tree. The height of left subtree is six and that of right subtree is two making the difference between right subtree and left subtree to 4. Out of possible 16796 tree, we get a tree that is skewed one.
Thus, we conclude that there is necessity of balancing the height of the tree.

```
int insert(struct btree **q, int value){
        if(*q == NULL){
                struct btree *temp = NULL;
                temp = (struct btree *)malloc(sizeof(struct btree));
                temp->left = NULL;
                temp->right = NULL;
                temp->key = value;
                *q = temp;
        }
        else if((*q)->key < value)
                insert(&((*q)->right), value);
        else if((*q)->key >= value)
                insert(&((*q)->left), value);
        return 0;
```

}

List 1:*BST* insert code

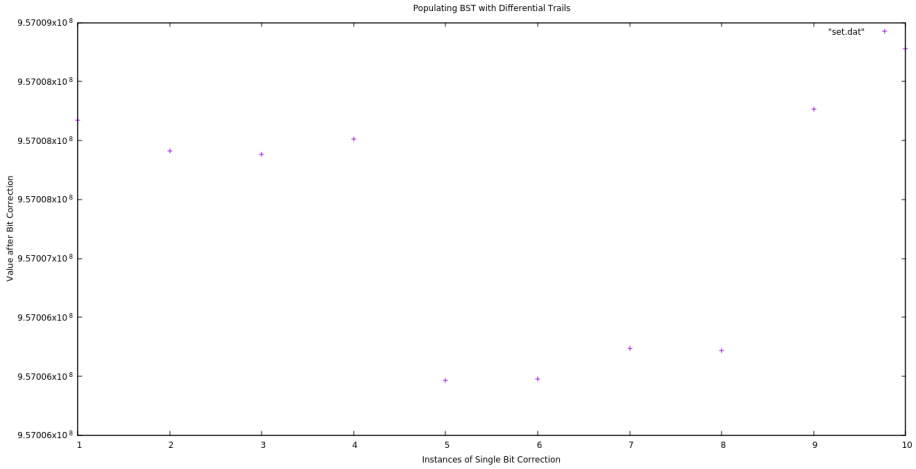


Figure 2.6: Differential Trail having one bit difference

### 2.6.2 AVL Tree

*AVL Tree* is a height balanced version of a *BST*. The height of the two left and right subtrees differ by atmost one. Let $L_h$ and $R_h$ be the height of left and right subtree at any given node. Then following equation holds:

$$|L_h - R_h| \leq 1$$

The *AVL Tree* uses rotations to balance its height. There are three rotation when we insert the above data into an *AVL Tree*. Due to self balancing nature, this data structure is very fast in data retrieval but the problem arises when there are lots of updates and deletion. The rotation goes on increasing. In our analysis a ten node tree made three rotation, so in the nodes increase the rotation also increases. Hence, a lot of computation is wasted in rotation and shifting the tree.

### 2.6.3 Red Black Tree

*RB Tree* is also height balanced tree. Each node of *RB Tree* carries an extra information, i.e. the color of the node; which can be either *Red* or *Black*. The main idea behind *RB Tree* is to make it balanced during insertion and deletion. There are several properties that must be maintained in a *RB Tree*. Due to

the consequences of these properties, the path from the root to the farthest leaf is no more than twice as long as the path from the root to the nearest leaf. Henceforth, the tree is roughly height balanced.
One further advantage of *RB Tree* is in the parallel programming paradigm. *RB Tree* are very efficiently constructed and managed on parallel hardwares. A sorted list of items takes about $O(loglog(n))$.
A redblack tree is similar in structure to a *B-tree* of order 4, where each node can contain between 1 and 3 values and (accordingly) between 2 and 4 child pointers. In such a B-tree, each node will contain only one value matching the value in a black node of the redblack tree, with an optional value before and/or after it in the same node, both matching an equivalent red node of the redblack tree.

**Note:** From the above walk through we conclude that among the available data structures, the *RB Tree* is one of the better option. The only disadvantage of *RB Tree* is the amount of space it takes. A *BST* or an *AVL Tree* would additionally need an auxiliary vector array to keep the track of the previous nodes.

### 2.6.4 Augmentation

Any custom *RB Tree* comes with three naive pointers, the left child, the right child and the parent pointer (we refer this as threading). The threading is very important and must be explicitly implemented. Apart from that there are few more augmentation as described below.

There exists one more constraint on data structure .i.e. huge backtracking with very frequent inserts and updates. The insert and delete can be handled by the RB Tree while the frequent backtracking requires access of the parent node in constant time. Hence an extra augmentation of the data structure is required to access parent in constant time. For this we have implemented the threaded tree. In a threaded tree the parent of the node can be accessed in $O(1)$ time. Thus, the definition of the used data structure is given below, $RB_t$:

```
struct btree{
        struct btree *thread;
        struct btree *left;
        unsigned int data;
        unsigned int id;
        char color;
        struct btree *right;
};
```

Each augmented node of $RB_t$ will branch according to the number of conditions imposed. A unique id is also associated with the node. This id will uniquely identify the node and will be populated in auto-increment manner. Figure 5 shows the detail of single node of the augmented $RB_t$.
The node begins with a parent pointer. The pointer points to its immediate parent. The parent pointer is followed by a left child pointer that contains the

address of its immediate left child. Then comes the storage of the differential trail. Here we store data that are currently generated using the algorithm described in section 9. The values here are not static and changes frequently according to the backtracking. Upon backtracking the value is simply overwritten with any memoization to save space. The node requires an id to uniquely identify itself to distinguish it from any other node on a different path that contains same value as the current node. If id is not used the algorithm 3 and 5 will make whole trails inconsistent by changing values for other paths as well. The probability of same value occurrence is very bleak but when occurs will must be correctly dealt. Further, to maximize the automation of consistency checks, id must be used. Any node has a *char* that identifies it as a red or black node. This information is used to balance the tree. In the end is the pointer to the right child that keeps the track of immediate right child.

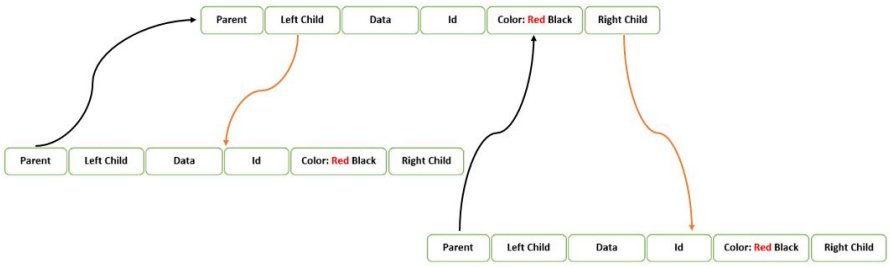


Figure 2.7: Detail of one node of $RB_t$

### 2.6.5 Time and Space Trade off

Though we have achieved a decent complexity to mine out the differential trail using the $RB_t$, yet the space consumption is the biggest let down of the above approach. The implementation using the $RB_t$ takes over one hour for generating the differential trail of 18 steps on an average. There were few instance when the trail went as far as 22 steps but never exceeded that. The algorithm used for the above approach is described in section 2.10 under Algorithm 1, 2, 3, 4, 5 and 6.
The bottleneck in the above approach is the bulky data structure. The serialization and the de-serialization becomes very time consuming task and hence we need some simple data structure. Further this new design must also keep the track of values that are obtained as they will be used for resetting the trail (like the path in tree).
One solution is to use linear array. Using the linear array enforces us to use additional following arrays:

- LB - a linear array that keeps track of branching to facilitate *backtracking*

- DB - a linear array that keeps track of *dirty bit*
- TBC - an optional linear array to keep track of *two bit condition*. This array will provide us with all inconsistent two bit conditions that occur randomly on the given data for the given number of steps. Analysis of this array would lead to further improvement of the trails in future work.

## 2.7 Brief Description of the Implementation

The implementation producing differentials spans over two phases. The first phase is static and does not require inconsistency checks. The other phase is dynamic and is completely based on the inconsistency checks. The achievement of number rounds are completely dependent on how efficiently we handle the dynamic phase. If we use non-linear data structure we get less differential rounds as compared to the linear data structure.

- Phase 1: Unrestricted Step Transformation
  The step transformation happens without any restriction. The is no condition check required; be it the two bit condition or the consistency check. The phase 1 on an average occurs two times during the complete trails. The first time is the initial start trail itself. On an average it goes to 3-4 rounds. Few times it goes to $5^{th}$ round. There were also instance where the phase 1 spanned only for two rounds. The second time phase 1 occurs randomly (no specific order or consistency) but very few times. Most of the times it occurred between $10^{th}$-$11^{th}$ and $12^{th}$-$13^{th}$ round. In case of linear approach the results were better. The initial trail went constantly till $5^{th}$ and $6^{th}$ round. Further, inconsistency still occurred around the $13^{th}$ and $14^{th}$ steps.
- Phase 2: Restricted Step Transformation
  All the step transformations that does not follow phase 1, comes under the phase 2. Here a valid differential is found only using the backtracking. The backtracking spans ranging 1-2 steps to several steps. Any bit whose backtracking order is four or more is considered as critical bit. Such bit occur in random order. Once occurred, the priority is to fix this bit first. Fixing this bit results in changing of the trails to a large extent. The critical bits start occurring from round 6-7 on an average for non linear and 8-10 on an average for the linear approach. Many a times the critical bits made the complete trails inconsistent and whole process had to be repeated again.
  The phase 2 is constructed around the two bit condition. We constantly check each and every bit that has been set in previous steps for its consistency in current step. Further, a separate consistency check is required. This check validates each an every bit against the current set bit. This check is very costly and is not done at each step. Instead, this is done only few times. During the analysis, we never found any conflict using the

extra consistency check and hence it was pure overhead for our analysis. One of the reason for being unsuccessful could be due to not checking each and every transformation. Such analysis could be done but is very costly and would need good hardware support.

Using the above approach, we were able to get to 17 to 19 rounds on an average for non linear and 21 to 23 steps on an average for linear approach. There was one instance that the trail went as far as 22 steps for non-linear and 26 steps for linear approach. After the average number of rounds the backtracking gets into some kind of deadlock and keeps running and changing the differential values.

## 2.8 Computational Complexity

We define the computational complexity as the time taken to produce valid differential upto 18 rounds on an average. On our laptop, with intel i7 dual core processor clocked at 2.6 GHz with hypervisor technology (two threads per core); it takes about one hour and fifteen minutes to generate 18 round differential. In case of linear implementation we produced differential upto 21 rounds on an average in about 40 minutes.
Further, we have implemented SHA2 in different way as described above. The bounds are not very stiff and few work around should bring it down.

## 2.9 Precomputation

Let $W$ and $W'$ be two message words. We randomly populate these message words and define them on the basis of table 1, as described above. The state variables $A$ and $E$ have fixed initial value. $A_0 = 0x6A09E667$ and $E_0 = 0x510E527F$. Thus, we define the characteristic of A and E as follows:
$\nabla A_i = 01101010000010011110011001100111$, where $i = 0$
$\nabla E_i = 01010001000011100101001001111111$, where $i = 0$
Define $\phi$ as the set containing undetermined bits .i.e. $\phi$ will contain 32 instances of ? or x. Let $\phi_i$ denote the $i^{th}$ bit of $\phi$

## 2.10 Algorithm

We have defined a total set of 6 algorithms. The first algorithm is the implementation using the $RB_t$ while the $7^{th}$ algorithm implements the characteristics using the Linear array $L$; rest 4 algorithms are the sub-routine to the first one and the seventh one and finds two bit condition and the performs the consistency checks.
Note: DFS aka depth first search used in algorithm 5 is self explanatory and naive one and hence did not required separate sub-routine. Also the $6^{th}$ algorithm is the naive RB tree insert/delete and update and does not require separate mention.

The table 2 shows trail for 17 rounds. It took about 1hr 15mins for generating the given trail. We started with the algorithm 1 with the values of $\nabla A$ and $\nabla E$ as given in the algorithm 1. Then we proceeded as in algorithm 1 by randomly taking a value from the $W$ and update in to $'-'$ or $'n'$ or $'u'$. Then we calculate the trails using the step transformation by using the subroutine CA - Checking the two bit conditions and cyclic anomaly. The subroutine for consistency check is also done randomly (not at each and every step). In case of backtracking we directly call the thread pointer as defined above and in the algorithm "$RB_t \rightarrow thread \rightarrow id$". Then we repeat the whole process again (step no. 27). A separate variable called the time stamp defined as $T_h$ is also used. This keeps track that we are not unnecessarily looping on a deadlocked state. We have taken its value between 15 min to 20 mins. This timeout occurs more often on consistency check rather than on two bit condition.

Alternatively; when we used the linear approach, we were able to generate a 26 round trail on an average with around 40mins for generating the trail. The process and procedure for consistency check and the two bit condition remains the same. The implementation difference comes in the path traversal where in $RB_t$ path was obvious and automatically constructed as we generated trails. Here, we must explicitly provide a different linear data structure that keeps the track of the trails. There is extremely low trade off in case of serializing and de-serializing a linear data structure as we just need to write the values and parse it while retrieving. Comparing this implementation with any non-linear implementation following two efficiency enhancements are obvious:

**Red Black Tree**

- Serialize and de-serialization of path
- Huge trade off in reconstruction of tree in case of backtracking

**Linear Array**

- No serialize or de-serialization of path
- Minimal trade of in reconstruction of values while backtracking

**Algorithm 1** Generating Differential Characteristics Part 1

**procedure** GDC 1
  Set $\nabla A_i = 01101010000010011110011001100111$, where $i = 0$
  Set $\nabla E_i = 01010001000011100101001001111111$, where $i = 0$
  Initialize $RB_t$
  srand(time(NULL))
  Randomly populate $W$ and $W^*$
  Define $\nabla W$ according to table 1
  Define $\phi$ and set it as the root of the $RB_t$
  Pick $v \in \phi_i$ and perform following check
  **if** $(v == ?)$ **then**
    Assign $v$ as '-' and update $\phi_i$
    Create a branch from the node of $RB_t$
  **else**
    Assign $v$ as 'u' or 'n'and update $\phi_i$
    Create a branch from the node of $RB_t$
  ConsistencyCheck()
  Perform the step transformation and update the state variables
  **while** $(v, z) \in \mathrm{CA}(v, z) \ \ \forall\ i, j | v \in \phi_i \wedge z \in \phi_j$ **do**
    Perform backtrack, Set $RB_t \rightarrow thread \rightarrow id$
    **if** $(v == ?)$ **then**
      Assign $v$ as 'x' and update $\phi_i$
    **else**
      **if** $(v ==$ n$)$ **then**
        Assign $v$ as 'u' and update $\phi_i$
      **else**
        Assign $v$ as 'n' and update $\phi_i$
    Recurse on the path from $id(\phi_i)$ to $id(\phi_0)$ | $(id(\phi_i), id(\phi_0)) \notin$ CA
    **if** (steps bactracked $> 3$) **then**
      Mark $v$ as critical
      Resolve $v$ first
    **if** $timestamp \geq T_h$ **then**
      Discard the current differential
      **goto** *Step 9.*

**Note:** *ConsistencyCheck()* gets called after random amount of steps

**Algorithm 2** Generating Differential Characteristics Part 2

1: **procedure** GDC 2
2: Pick $v \in \phi$ randomly
3: **if** $v \neq$ '-' **then**
4: *goto 2*
5: Assign $v$ '0' or '1'
6: Compute the propagation
7: ConsistencyCheck()
8: **if** *consistency fails* **then**
9: *goto 5 and pick and pick a different value*

**Algorithm 3** Two Bit Condition and Cyclic Anomaly $(v_i, v_j)$

1: **procedure** CA
2: **if** $(\text{Path}(v_i, v_j))$ **then**
3: **if** $(id(\phi_i) == id(\phi_j) \wedge id(\phi_i) \neq id(\phi_j))$ **then**
4: return true
5: **else**
6: return false
7: **else**
8: return false

**Algorithm 4** Consistency Check

1: **procedure** ConsistencyCheck
2: Assign a value to the bit and check for consistency with all other bits

**Algorithm 5** Check if nodes are in same path

1: **procedure** Path $(v_i, v_j)$
2: **if** $(\text{DFS}(v_i, v_j))$ **then**
3: return true
4: **else**
5: return false

**Note:** DFS is Depth First Search sub-routine

**Algorithm 6** Augmented RB Tree

1: **procedure** Insert
2: Create a new threaded node with color *red*
3: Add the node to the $RB_t$
4: Call the auxiliary routine to re-color nodes to maintain Red-Black properties
5: **procedure** Update
6: Change the value of the bit $u$ at the current node

**Note:** This algorithm uses normal RB Tree operation

**Algorithm 7** Generating Differential Characteristics using Linear Array

**procedure** LINEARARRAY
  Set $\nabla A_i = 01101010000010011110011001100111$, where $i = 0$
  Set $\nabla E_i = 01010001000011100101001001111111$, where $i = 0$
  Initialize $L$
  srand(time(NULL))
  Randomly populate $W$ and $W^*$
  Define $\nabla W$ according to table 1
  Define $\phi$ and set it as the current value of $L$
  Pick $v \in \phi_i$ and perform following check
  **if** $(v == ?)$ **then**
    Assign $v$ as '-' and update $\phi_i$
    Initialize and Populate $LB$ as the current branching factor
  **else**
    Assign $v$ as 'u' or 'n'and update $\phi_i$
    Populate the next available space of crossponding $LB$
  ConsistencyCheck()
  Perform the step transformation and update the state variables
  **while** $(v, z) \in \mathrm{CA}(v, z)$ $\forall\, i, j | v \in \phi_i \wedge z \in \phi_j$ **do**
    Perform backtrack, fetch last value of $LB$ and update the sentinel
    **if** $(v == ?)$ **then**
      Assign $v$ as 'x' and update $\phi_i$
    **else**
      **if** $(v ==$ n$)$ **then**
        Assign $v$ as 'u' and update $\phi_i$
      **else**
        Assign $v$ as 'n' and update $\phi_i$
    Recurse from $L$ to $LB$ from $id(\phi_i)$ to $id(\phi_0)$ | $(id(\phi_i), id(\phi_0)) \notin$ CA
    **if** (steps bactracked $> 4$) **then**
      Mark $v$ as critical
      Resolve $v$ first
    **if** *timestamp* $\geq T_h$ **then**
      Discard the current differential
      **goto** *Step 9.*

**Note:** *ConsistencyCheck()* gets called when backtracking occurs more than two times

Table 2.2: 17 round differential trail using $RB_t$

| $i$ | $\nabla A_i$ | $\nabla E_i$ | $\nabla W_i$ |
|---|---|---|---|
| 0 | 01101010000010011110011001100111 | 01010001000011100101001001111111 | x???????xx?x?xxxxxx?x?xx?x???xxx |
| 1 | 11010110101000110101010101110001 | 10110110001110110001100101011101 | x??-????xx?x?xxxxxx?x?xx?x???xxx |
| 2 | 00101100111100000110101100110000 | 01000100000111010100100111111011 | x?–????xx?x?xxxxxx?x?xx?x???xxx |
| 3 | 00110100011100001111110110010010 | 00111001111110010111101111110101 | x?–????xx?x?xxxxxx?x?xx?x-??xxx |
| 4 | 01000101000011010000111000100001 | 01110010100011011111000111011011 | x?–????xx?x?xxxxxx?x?xx?x-?-xxx |
| 5 | 01000010011110000101111001110111 | 00001011101100001010011000100110 | x?–????xx?x?xxxxx0?x?xx?x-?-xxx |
| 6 | 01011001010011100100110000111001 | 00111110100010000100001011101000 | x?–??-?xx?x?xxxxx0?x?xx?x-?-xxx |
| 7 | 10101001110101100111101111001100 | 11001111011110000001001010000001 | x?–??-?xx?x?xxxxx0?x?xx?x—xxx |
| 8 | 01111000010100101101100111111100 | 01101010010100110010111111010100 | x?–??-?xx?x?0xxxx0?x?xx?x—xxx |
| 9 | 11100111111100001111110011000111 | 11111101110011000011100101001111 | x?–??-?xx?x?0x1xx0?x?xx?x—xxx |
| 10 | 00111010110101111000111111111111 | 01111001101101110000100000001101 | x?–?–?xx?x?0x1xx0?x?xx?x—xxx |
| 11 | 10100000110111001000110011001111 | 11101011011001011001000110111100 | x?–?–?xx?x?0x1xx0-x?xx?x—xxx |
| 12 | 00010001011011011110000111000111 | 01011101000101001100101110000111 | x?–?–?xx?x?0x1xx0-x?x1?x—xxx |
| 13 | 11111101101001000101010110001011 | 10000000010000100010101101110110 | x?–?–?xx?x?0x1xx0-x?x1?x—xxx |
| 14 | 11110100011100100001010101011000 | 10010111011011111100011011111001 | x?–?–?xx?x?011xx0-x?x1?x—xxx |
| 15 | 11101111101111110000100110111100 | 11101000111001110111010100101011 | x?–?–?xx?x?011xx0-x?x1?x—xx0 |
| 16 | 00001110110100011011101100010100 | 00010011111111110011001101001011 | 0?–?–?xx?x?011xx0-x?x1?x—xx0 |

Table 2.3: 26 round differential trail using $L$

| $i$ | $\nabla A_i$ | $\nabla E_i$ | $\nabla W_i$ |
|---|---|---|---|
| 0 | 01101010000010011110011001100111 | 01010001000011100101001001111111 | ??????x?xxxxx?x?x??xxxxxxxx?xx?? |
| 1 | 01010011100111001000010000111001 | 00110011000010101111000000010011 | ??????x?xxxxx?x?x??xxxxxxxx?x1?? |
| 2 | 11110001111000011011011110001001 | 10110111010000001001110101010111 | ??????x?x1xxx?x?x??xxxxxxxx?x1?? |
| 3 | 10101111010001000101111011101000 | 11001101111001100001110111001101 | ??????x?x1xxx?x?x??x1xxxxxx?x1?? |
| 4 | 01001111100101001110111011011000 | 01001111111000110100111100111111 | ??????x?x1xxx?x?x??x1xxxxxx?x1?- |
| 5 | 00110111100010110010110111110110 | 00110000000001111111000100001011 | ??????x?x1xxx?x?x-?x1xxxxxx?x1?- |
| 6 | 01101010100000000100000100111110 | 01011100100010110111011010110011 | ??????x?x1xxx?x?x-?x1xxxxx0?x1?- |
| 7 | 10010110011000001000001111100000 | 10111101110000110000001101001100 | ??????x?x1xxx?x?x-?x10xxxx0?x1?- |
| 8 | 00001001001011011010101101011001 | 00000101001101001010100001110101 | ??????x?x1xxx?x?x-?x10xxxx0?x1?- |
| 9 | 11101001001100111110010000101110 | 10101101101110000100101110000100 | ??????x?x1xxx?x?x-?x10xxxx0?x1?- |
| 10 | 00010010101010101010110111100100 | 00111101000011101000101010101011 | ???-??x?x1xxx?x?x-?x10xxxx0?x1?- |
| 11 | 11111101010100010001111101011011 | 11100011010110111010010110101110 | ???-??x?x1xxx-x?x-?x10xxxx0?x1?- |
| 12 | 00011101000010001111010110010011 | 01001011101101111011000011110010 | ???-??x?x1xxx-x?x-?x10xxxx0?x1?- |
| 13 | 10000110000010110011000111101100 | 11010000100110111101111010111101 | ???–?x?x1xxx-x?x-?x10xxxx0?x1?- |
| 14 | 11001100111100011100111010011101 | 11010100110110100000011010110111 | ??—?x?x1xxx-x?x-?x10xxxx0?x1?- |
| 15 | 01001100100111011011110101100100 | 00010011111001111101101100110000 | ??—?x?x1xxx-x?x-?110xxxx0?x1?- |
| 16 | 11111111101001101111111110110000 | 11100000111001011001001100101110 | ??—?x?x1xxx-x?x-?110xxxx0?x1?- |
| 17 | 11111000100011100100111000111111 | 11110001101101101101010010111110 | ??—?x?x1xxx-x?x-?110xxxx0?x1?- |
| 18 | 10110111111111110100101111111111 | 11110100111101010011110101110001 | ??—?x?x1xxx-x?x-?110xxxx0?x1?- |
| 19 | 11000000000101100100110010010011 | 10101011111010001101110010001001 | ??—?x?x1xxx-x?x-?110xxxx0-x1?- |
| 20 | 11011011110000001010101011010010 | 10100001011011010111010011010010 | ??—?x?x1xxx-x?x-?110xxxx0-x1?- |
| 21 | 11010111001111111100110110101110 | 11101000111100101100110001111100 | ??—?x?01xxx-x?x-?110xxxx0-x1?- |
| 22 | 11101110011011100000110111001101 | 11010111011001010001110010100001 | ??—?x?01xxx-x?x-?110xxxx0-x1?- |
| 23 | 11111110011000001000000011001011 | 10101101011011110101010001101110 | ??—?x?01xxx-x?x-?110xxxx0-x1?- |
| 24 | 10101001000011011010011101100011 | 11110011011101001010010001010010 | ??—?x?01x0x-x?x-?110xxxx0-x1?- |
| 25 | 10110001100101000000001000100101 | 11101001011111100101010001001001 | ??—?x?01x0x-x?x-?110xx1x0-x1?- |
| 26 | 11111111010011000001101010010101 | 10100001011101000010110001100001 | ??—?x?01x0x-x?x-?110xx1x0-x1?- |

# Chapter 3

# Conclusions - Future Works and Directions

The future scope of our work is centered around studying the inconsistent behaviour of the trails and omitting them in the initial rounds. Finding the bits that do not contribute for valid trail is not easy to find in the run time. One of the way to mine out those dirty bits is to store all the inconsistent values in an array and later exhaustively study each and every value.
The linear array $TBC$ in our algorithm represents the inconsistent values. We need to grill down each and every value together with its path and all the conditions. The research on array should center on following points:

- Initial trail - There would be no impact on the initial trails. As described in paper this is unrestricted trails without any condition and hence our array provides no improvement here
- Span of *First Inconsistency* and *Second Inconsistency* - The study of array should directly affect the steps covered between the first and the second inconsistent values. Here we should get significant improvements as more and more values from $TBC$ is removed out. The number of steps between them should increase on an average. A condition might occur where we might get a decreased efficiency if the values from $TBC$ have some cyclic dependency on the future values.
- Final Inconsistency - Here we also get lots of improvement as we tackle the array. The arguments remains similar as above. The only difference the large span in the number of steps.
- Backtracking - There is not much significant improvement in the degree of backtracking though number of backtracking instances do decrease.

# Bibliography


[1] Christophe De Canniere and Christian Rechberger. Finding SHA-1 characteristics: General results and applications. pages 01–20. Advances in Cryptology - ASIACRYPT, 2008.

[2] I. Damgard. *A Design Principle for Hash Functions.* Lecture Notes in Computer Science Vol. 435, 1989.

[3] Henri Gilbert and Helena Handschuh. Security analysis of SHA-256 and sisters. Lecture Notes in Computer Science, Vol 3006, 2003.

[4] Florian Mendel, Tomislav Nad, and Martin Schlaffer. Finding sha-2 characteristics: Searching through a minefield of contradictions. pages 288–307. Advances in Cryptology - ASIACRYPT, 2011.

[5] Ralph Charles Merkle. Secrecy, authentication, and public key systems, 1979.

[6] R.C. Merkle. *A Certified Digital Signature. In Advances in Cryptology.* Lecture Notes in Computer Science Vol. 435, 1989.

[7] Ivica Nikolic and Alex Biryukov. Collisions for step-reduced sha-256. Fast Software Encryption, Springer, 2008.

[8] Somitra Kumar Sanadhya and Palash Sarkar. Collisions for step-reduced sha-256. Lecture Notes in Computer Science, Springer, 2005.

[9] Somitra Kumar Sanadhya and Palash Sarkar. *New Collision attacks Against Up To 24-step SHA-2.* Lecture Notes in Computer Science, pages 7895. Springer, 2008.