THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

Automated Searching for Differential Characteristics in SHA-2

Mitchell Grout

COMP520-18Y (HAM)

This report is submitted in partial fulfilment of the requirements for the degree of Bachelor of Computing and Mathematical Sciences with Honours (BCMS(Hons)) at The University of Waikato.

Abstract

The SHA-2 family of cryptographic hash functions are a cornerstone of modern security. They have found a use in data integrity, secure storage of sensitive information, and many other security-critical applications. Many of these applications depend on the notion that the SHA-2 family is collision-resistant, meaning it is difficult to find distinct values that share the same hash value. To ensure the security of these functions, it is necessary that researchers cryptanalyze these functions so that they may be recommended for deprecation before it is possible to subvert these properties.

One of the main techniques used for cryptanalyzing SHA-2 is differential cryptanalysis, which relies on a special value Δ_0 to efficiently find collisions. However, due to a lack of public knowledge, including relevant literature and a large barrier to entry, it is not clear to aspiring cryptanalysts how this value is found. Furthermore, this lack of knowledge may potentially hamper further research in the field. To help lower this barrier, we have explored the field in-depth, attempting to find efficient ways to compute this value, and hence find useful results for the SHA-2 family.

To this extent we have created our own SHA-2 variant named MAW32, which shares a similar internal structure to the SHA-2 family. This variant allows for rapid testing of ideas, while still allowing for easy extension to the SHA-2 family. We have attempted to find suitable values for Δ_0 in MAW32 by applying three optimization techniques, including random search, genetic algorithms, and a hybrid approach. Applied testing of these results indicates that these techniques are promising in the search for Δ_0, and may be applied in conjunction with existing techniques.

We hope that the results of this report can help shed light on how differential cryptanalysis works. Furthermore, we hope that the results of this report can be potentially applied to the current cryptanalysis of the SHA-2 family and other cryptographic hash functions that share a similar structure.

Acknowledgements

This project was only possible with the help of many members from the Faculty of Computing and Mathematical Sciences, and I would like to take this time to personally thank each and every one of them. I would firstly like to give my thanks to my supervisors, Associate Professor Ryan Ko, Aleksey Ladur, and Cameron Brown. Your suggestions, support, and advice has given me the confidence to research this difficult field and helped make this project as successful as it is.

I would like to thank Daniel Roodt of CROW for his expert advice on both cryptographic hash functions, and on their cryptanalysis. You have provided an amazing amount of support and have helped me understand so much about this topic.

To Dr. Steven Miller and Nick Lim, your statistical expertise has helped me to find the most appropriate ways to test my results, and to ensure that there is no room for misinterpretation of my results.

Finally, I would like to thank all my fellow honours students, Christian Anderson, Daniel St. George, Christopher Chew, and Armajot Parmar, as well as all my fellow CROWs. Your support and suggestions throughout my project has helped make this project a success; it has been a pleasure to work with you all.

Contents

List of Figures

List of Tables

1. Introduction

1.1 Motivation

Cryptographic hash functions are a class of mathematical functions which can be used to assign an arbitrary input an almost-unique number. Because of this near-uniqueness property, cryptographic hash functions enjoy many security-centric applications such as verifying the integrity of data, proof of work systems, and verifying digital identities. Many of these applications depend on the non-existence of collisions, pairs of inputs which are assigned the same number by the hash function. The existence of such a pair would allow malicious actors to potentially subvert entire systems, by replacing valid data with malicious data, both sharing the same hash value. To this extent, it is the job of cryptanalysts to attempt to find such pairs, so that insecure cryptographic hash functions can be deprecated and replaced with more secure algorithms.

The first commonly used hash function was MD4, originally designed by Ronald Rivest in 1990, and made available as RFC1186[1]. Since its initial publication, it was heavily analysed by cryptanalysts, and attacks were quickly formulated against it. In 1992, Bert den Boer and Antoon Bosselaers demonstrated a collision for the last two rounds of MD4[2], and Hans Dobbertin demonstrated a practical attack for the entire function in 1996[3]. MD4 was quickly replaced by Rivest in 1992 by a successor, MD5[4], "… because MD4 was designed to be exceptionally fast, it is 'at the edge' in terms of risking successful cryptanalytic attack.".

While research continued on MD4, cryptanalysts also began looking at MD5, which shared a similar internal structure to MD4. Progress was quickly made, with collisions for the MD5 compression function being found in 1993 by Boer and Bosselaers. However, there was no significant progress on breaking MD5 for almost a decade. In 2005, a team of researchers led by Xiaoyun Wang announced a practical attack on MD5[5], in which an MD5 collision could be found in less than an hour of computation. Later that same year, the full effect of this was shown, when Arjen Lenstra and Benne de Weger successfully demonstrated the creation of two distinct X.509 certificates which shared the same MD5 value[6]. With this, both MD4 and MD5 were considered 'broken', indicating that they could no longer be used practically due to the existence of collisions, and were later recommended for deprecation[7, 8].

This cycle has repeated over the years; new cryptographic hash functions are created, tested, broken, and replaced. This cycle is ultimately beneficial, as the continual changing of functions prevents malicious actors from being able to practically exploit any currently used hash functions. A prime example of this cycle is the Secure Hash Algorithms (SHA) family, a collection of several cryptographic hash function families published by the National Institute of Standards and Technology (NIST). Currently, the SHA family consists of SHA-0, SHA-1, SHA-2 and most recently SHA-3.

SHA-0 was originally designed by the United States National Security Agency in 1993[9] as part of the Capstone Project, but after undisclosed issues in the algorithm were found, it was quickly succeeded by the SHA-1 family in 1995[10]. SHA-1 found a use in many networking standards such as SSL, TLS, and digital signatures, and enjoyed over a decade of use. However, due to its widespread use, it was the focus of many cryptanalysts, with their work culminating in 'The SHAppening'[11]; a breakthrough in late 2015 which demonstrated a collision on the compression function of SHA-1. Although this did not immediately give rise to an actual collision in SHA-1, it was the first practical attack which had the potential to be leveraged to an actual attack. At this time, the authors suggested an immediate deprecation of SHA-1. In early 2017, researchers from Centrum Wiskunde & Informatica (CWI) and Google announced the 'SHAttered attack'[12], a practical collision on the full SHA-1 function. In this attack, two distinct PDF files were created which shared the same SHA-1 hash; a process that required the same processing power as 110 years of single GPU SHA-1 computations.

The SHA-2 family is the successor to the SHA-1 family, first standardized in 2001 under FIPS 180-2[13]. This family had several advantages over SHA-1, including a larger block size and more complex compression function. This family has also been adopted for use in several standards, with one of the most notable uses being in the Bitcoin proof-of-work algorithm. Since its initial release, there has been a large amount of public cryptanalysis on the SHA-2 family, however the current best results are still far from finding a collision. However, as is with the cycle, a new family of hash functions has been designed, SHA-3, which will eventually supersede the SHA-2 family.

1.2 Aim

This project aims to produce a general technique to allow for the automatic cryptanalysis of the SHA-2 family of hash functions, specifically SHA-256. This will use the well-established framework of differential cryptanalysis, which describes how collisions may be found readily by determining how differences propagate through the hash function. Specifically, we will produce a program which will produce potential input differences which can in turn be used to find collisions in SHA-256. However, due to the sheer complexity of SHA-256, we will instead work with a custom reduced variant named MAW32. This hash function shares a very similar internal structure, but has a reduced block and digest size, number of rounds, and internal functions. This allows for testing of techniques in a short amount of time, which can then be used to comment on SHA-256 itself.

Due to the literature on hash function cryptanalysis being very sparse, it is difficult for aspiring cryptanalysts to enter the field. For this project, the majority of material we considered when beginning the literature review for this project consisted of master's and PhD theses, as well as conference proceedings for conferences such as EUROCRYPT. These sources are not well-suited to those with little to no background knowledge of cryptanalysis, and as such, we aim for this report to also serve as a simple introduction to the field of hash function cryptanalysis.

1.3 Importance

Amongst the material we considered in our literature review, the majority of cryptanalysis for SHA-256 used the framework of differential cryptanalysis, originally used by Eli Biham and Adi Shamir in their attack on DES in 1993[14]. Such an attack requires what is known as an input difference, Δ_0, which is pivotal in readily finding collisions for a hash function. However, most of the literature considered did not spend much time at all on how this value was found; rather, the value was simply given, and results derived from it. This is acknowledged within the field by Martin Schläffer, who states "… Wang *et al.* give input differences, differential paths, and the corresponding conditions that allow to find collisions with a high probability. However, Wang *et al.* do not explain how these paths were found. The common assumption is that they were found by hand with a great deal of intuition"[15]. As such, this report provides a well-documented way of deriving Δ_0 itself via a computational method, which in turn may be specialized and used as part of other approaches more efficiently.

1.4 Overview

As part of this project, we completed an in-depth literature review, with a focus on cryptanalytic techniques applied to classic cryptographic hash functions such as MD4, MD5 and SHA-1, as well as SHA-256. This review will be outlined in the next section of this report, and will provide further details typically not covered in most literature. Additionally, we will look in-depth at the framework of differential cryptanalysis, as well as the structure and implementation of hash functions, including SHA-256.

From here, we will define our reduced SHA-256 variant, MAW32, giving specific details as well as a reference implementation, with further explanation as to why it is applicable in our project. We will also describe how discuss how some of the ideas presented could possibly be implemented, along with approximations for computations which are otherwise too complex. In the following chapter, we will formulate a method for testing our possible approaches, giving necessary conditions for our criteria, how results will be compared, and how to determine validity. We will then finish with a discussion of our results, and determine whether any of them provide a suitable solution to our problem. Finally, we will give an evaluation of our work, especially as compared to current results in the field, discuss the impact of our project on the field, and discuss future work relating to this project.

2. Background

2.1 Hash Functions

We define a hash function H to be a pure mathematical function, which takes an arbitrary length input, and produces some fixed length output. We often denote this with the signature $H : \mathbb{Z}_2^* \to \mathbb{Z}_2^n$ for some $n \in \mathbb{N}$, where $\mathbb{Z}_2 = \{0,1\}$, $\mathbb{Z}_2^n$ is all vectors over $\mathbb{Z}_2$ of length n, and $\mathbb{Z}_2^* = \bigcup_{n\in\mathbb{N}} \mathbb{Z}_2^n$ is all arbitrary vectors over $\mathbb{Z}_2$. The input of such a function is referred to as the message, and the output is referred to as the digest, or hash value associated with the message. Typically, the input is viewed as some block of memory, while the hash value is viewed as some n-bit integer. A naïve hash function may involve 'folding' the block by combining each element with each other via the XOR operator. An example, specialized for strings, is given below.

```
uint8_t right_rotate(uint8_t i, size_t n)
{
    return (i << n) | (i >> (8 - n));
};

size_t hash(char *val)
{
    uint8_t *sent = (uint8_t *) val;
    size_t carry = 0;
    for (int i = 0; sent[i] != '\0'; i++)
        carry ^= right_rotate(sent[i], i % 8);
    return carry;
};
```

Figure 1 - Basic Hash Function

Here, we see each element in our block is examined, and is used to affect the overall hash value for the input. This function is also purely deterministic; repeat calls to `hash` is guaranteed to yield the same result. A common application of hash functions is when a 'fingerprint' needs to be associated with some value. A typical example would be a hash table. This is a generalization of an array, in which the array key can be any arbitrary type. To allow for this, the hash value of the key is taken, and used to perform the usual integer-based lookup in an array. A specialized example, utilizing our previous hash function, is given below.

```
class hashtable
{
private:
    size_t len;
    int *data;
public:
    hashtable(size_t n) : len(n)
    {
        data = new int[len];
    }

    int& operator[](char *key)
    {
        return this->data[hash(key) % len];
    }
};
```

Figure 2 - Application of Hash Functions: Hash Table

With this, we can effectively store integers in an array which is indexed by a string. However, there is a potential issue; if $hash$ were to produce the same value for two distinct values of key, then we may inadvertently overwrite some of our data. In this example, although we assigned the value of 2 to the string "$university$", when we go to retrieve the value later, it has been accidentally changed to the value 0.

```
hashtable word_count(256);
word_count["waikato"]    = 1;
word_count["university"] = 2;
word_count["tniversiuy"] = 0;
printf("%d\n", word_count["university"]);
```

Figure 3 - Hash Function Collision

Necessarily, all hash functions must admit collisions; a pair of inputs (x, y) satisfying $H(x) = H(y), x \neq y$. This is a consequence of the domain of the hash function being infinite, while the codomain is finite. We will refer to this pair (x, y) as a colliding pair. In our example, the pair ("$university$", "$tniversiuy$") is a colliding pair for the function $hash$. In applications such as hash tables, a collision is a non-issue. If a collision occurs when inserting a new value, we can easily resize the underlying array, add the colliding values to a linked list, or simply add an offset to the hash value; the 'fingerprint' of our value does not in fact need to be unique. However, this can be a problem in security-critical applications, where the fingerprint is assumed to be almost-unique up to the input.

2.1.1 Properties

General hash functions have very lax properties. By definition, a hash function must be able to take an arbitrary input and produce some fixed length output. Furthermore, as pure functions, they must be entirely deterministic, meaning repeated calls with the same input should always produce the same fingerprint, regardless of any external state. Although not a strict requirement, many hash functions have the following three properties:

- Fast: Computing the hash value of any arbitrary input should have very low overhead
- Uniformly distributed: Each element in the codomain should have an approximately equal chance of occurring, with no bias to a particular output
- Continuity: If two inputs $x, \bar{x}$ differ by only a small value, then $H(x)$, $H(\bar{x})$ should also differ by only a small value

These conditions are prime for applications such as database indexing or redundancy checks, however are not suitable for security-critical applications. For such applications, a more specific class of functions is necessary.

2.2 Cryptographic Hash Functions

A cryptographic hash function is a special type of hash function, with properties specifically tailored for security. However, many of the properties of ordinary hash functions are not desirable for cryptographic hash functions, so a different structure is required.

2.2.1 Properties

A cryptographic hash function is defined to have three inherent properties:

- Fast: Computing the hash value of any arbitrary input should have very low overhead
- Uniformly distributed: Each element in the codomain should have an approximately equal chance of occurring, with no bias to a particular output
- Discontinuity: Also referred to as the avalanche effect, if two inputs $x, \bar{x}$ differ by only a small value, then $H(x)$, $H(\bar{x})$ should differ by a large value. Typically, if $x, \bar{x}$ differ by a single bit, then half of the bits in $H(x)$, $H(\bar{x})$ should be different on average.

Due to the property of discontinuity, the values generated by a cryptographic hash function may appear seemingly random, which in conjunction with uniformity, makes it difficult to relate any given input to a given output. To ensure that the function is non-predictable, we also require that is satisfies three key properties, referred to as resistances:

- Preimage resistance: Given a hash value y, it is infeasible to find a message x such that $H(x) = y$
- Second preimage resistance: Given a message x with hash value $y = H(x)$, it is infeasible to find a second message $\bar{x}$ such that $(x, \bar{x})$ is a colliding pair.
- Collision resistance: It is infeasible to find a colliding pair $(x, \bar{x})$.

By infeasible, we mean that there is no better method than to search the input space of H by brute force. Assuming the codomain of H is perfectly uniformly distributed, such that each element occurs with a probability of 2^{-n}, then we mean that there is no algorithm that runs faster than $O(2^n)$. This applies to both preimage and second preimage resistance, however collision resistance is an interesting exception; to find an arbitrary collision, it suffices to randomly select $2^{n/2}$ inputs and generate their hash values. With this alone, there is a 50% chance that a collision will be found, hence for a hash function to have collision resistance, there must be no algorithm which runs faster than $O(2^{n/2})$. A detailed explanation of this seemingly paradoxical attack is given in the next section.

2.2.2 Generalized Birthday Paradox

Suppose we have a room full of people; we ask, what is the probability that at least two people in that room share the same birthday? Note that we assume that the probability of having a birthday is uniformly distributed, and that there are only 365 different birthdays. If we have 366 people in the room, then by the pigeonhole principle, there is surely at least one pair of people with the same birthday. Let $P(n)$ denote the probability that of n people, at least one pair shares a birthday. This can be thought of the opposite of the probability that of n people, nobody shares a birthday, hence:

$$P(n) = 1 - \prod_{k=0}^{n-1} \frac{365-k}{365} = 1 - \left(\frac{1}{365^n}\right) \prod_{k=0}^{n-1} (365-k) = 1 - \frac{365!}{365^n (365-n)!}$$

Table 1 - Birthday Paradox

$\boldsymbol{n}$	**1**	**2**	**3**	**4**	…	**22**	**23**
$\boldsymbol{P(n)}$	0	0.00273972	0.00820414	0.01635591	…	0.47569530	0.50729723

By iterating through every $n \in \{1, \dots, 365\}$, we can computationally check the probability of at least two people sharing the same birthday, given n randomly selected people. Now, we ask, how many people do we need to have to ensure that it is likely that at least one pair of people will share a birthday? By taking 'likely' to mean $P(n) > 0.5$, we have:

$$P(n) = 1 - \frac{365!}{365^n (365-n)!} > \frac{1}{2} \Rightarrow n \geq 23$$

So, we require at least 23 people in a room to have at least a 50% chance of two people sharing the same birthday as each other. This is readily generalized to any uniform distribution. If each event has a probability of $\frac{1}{d}$, then we must sample approximately $\sqrt{d} = d^{1/2}$ inputs in order to find a matching pair with at least a probability of 50%.

Now, suppose H is a cryptographic hash function with an n-bit hash value. It is then uniformly distributed on its codomain, hence each element in the codomain has a probability of $\frac{1}{2^n}$ of occurring. By the generalized birthday paradox, in order to find two inputs with the same output with at least a probability of 50%, we need at least $\sqrt{2^n} = 2^{n/2}$ samples. This gives an upper bound on the complexity of $O(2^{n/2})$ for breaking collision resistance in a cryptographic hash function.

2.3 Construction of Cryptographic Hash Functions

Due to the strength and complexity required by cryptographic hash functions, many standards for their construction have been developed over the past years. A cryptographic hash function can be considered to be two distinct parts: a compression function, and construction scheme. A compression function is a function $h : \left(\mathbb{Z}_2^n \times \mathbb{Z}_2^b\right) \rightarrow \mathbb{Z}_2^n$ with $b, n \in \mathbb{N}$. We refer to the value b as the block size, and n as the digest size of the compression function. These functions are used as the core of a cryptographic hash function, and as such, must have all the properties required of a cryptographic hash function. A construction

scheme is a technique which allows the extension of a compression function to another function of the form $H : \mathbb{Z}_2^* \rightarrow \mathbb{Z}_2^n$.

2.3.1 Davies-Meyer Construction

To construct a cryptographic hash function, we first need a compression function. This function must also have all of the properties required by a cryptographic hash function. As such, constructing a compression function is non-trivial. To this end, there also exist many constructions for creating a compression function from other cryptographic primitives. One such construction is the Davies-Meyer construction, which creates a compression function from a cryptographically secure block cipher.

Let $E(M, K)$ denote the encryption of a message M with key K with some secure block cipher, and let our message be broken into several fixed-length blocks, denoted M_i. We can then construct a compression function h by setting $h(IV, M) = H_i$, where H_i is the final element in the recurrence relation $H_{n+1} = E(H_n, M_n) \oplus H_n$, where $H_0 = IV$.

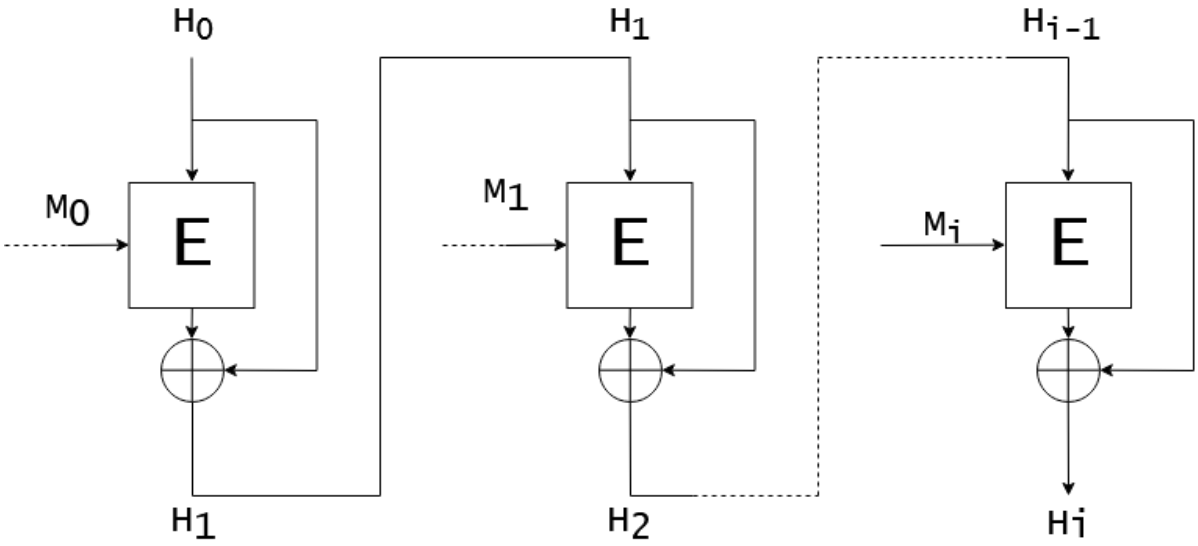

Figure 4 - Davies-Meyer Compression Function

Effectively, our compression function takes some initial value IV, and repeatedly encrypts it with each block in the message M_n, and additionally XORing on each intermediate value of H_n at each step. Due to this additional step, the compression function h is difficult to invert, given that the block cipher E is cryptographically secure. Furthermore, since block ciphers share similar randomness properties with cryptographic hash functions, h will also be collision-resistant.

2.3.2 Merkle-Damgård Construction

Given a secure compression function, how can we extend it to a full hash function? To do this, we need to somehow take an arbitrary length message and divide it into blocks that the underlying compression function can consume. Furthermore, we need to maintain some type of state for the compression function to use. The Merkle-Damgård construction describes how to construct such a function from a compression function.[16]

To begin, we define $Pad(M)$ to be the message padded with its length in bits. Although Pad can be defined arbitrarily, it must satisfy several key properties:

- b must divide $Length(Pad(M))$
- M is a prefix of $Pad(M)$
- If $Length(M) = Length(N)$, then $Length(Pad(M)) = Length(Pad(N))$
- If $Length(M) \neq Length(N)$, then the final b bits of $Pad(M)$ should be different to the final b bits of $Pad(N)$

Let our compression function be given as $h : \left(\mathbb{Z}_2^n \times \mathbb{Z}_2^b\right) \rightarrow \mathbb{Z}_2^n$. Furthermore, let IV be our initial value, a well-known constant which is n-bits in size. Finally, let our message be denoted by M, so $Pad(M)$ can be broken into the b-bit blocks $M'_0, M'_1, \ldots, M'_k$. We define $H(M) = H_k$, where H_k is the final element in the recurrence relation $H_i = h(H_{i-1}, M'_i)$, and $H_0 = IV$ is our initial value.

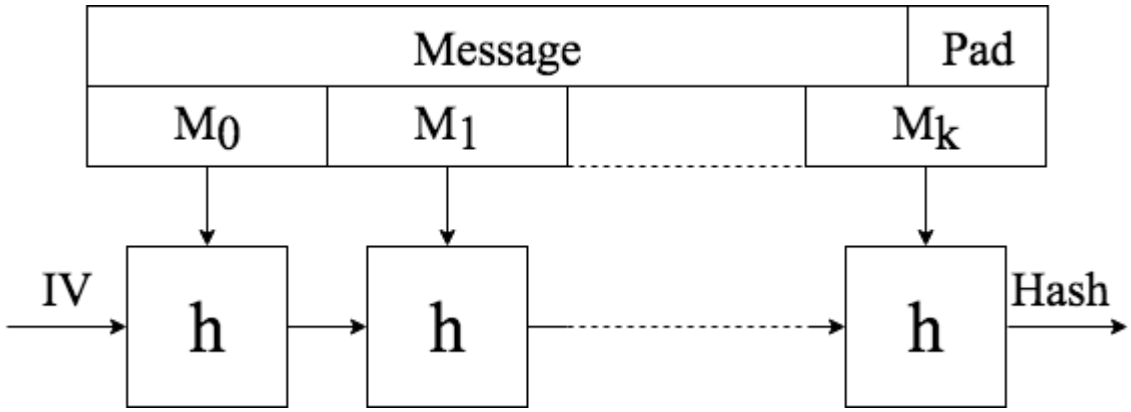

Figure 5 - Merkle-Dåmgard Construction

In essence, we are daisy-chaining the compression function h with itself, k many times, where at each iteration, it uses its last result and the next block in the padded message. Given that h is collision-resistant, then the hash function H derived from h is also collision resistant[17].

2.4 SHA-256

The SHA-2 family was originally standardized under FIPS180-2 by NIST, and was originally designed by the National Security Agency[13]. The most recent revision of the standard is FIPS180-4. This standard defines the SHA-2 family as a collection of hash functions, using a Merkle-Damgård construction, and a compression function derived from an unknown block cipher using the Davies-Meyer construction. As the internal block cipher does not appear to be a well-known cipher, it is commonly assumed that it was specifically designed for this family of functions. All the functions in the SHA-2 family use the same compression function, with the primary difference being in the number of bits used for the internal state, constants, and some minor differences in function definitions.

Below, we will look at how SHA-256 is implemented. We begin with a technical description of the function. SHA-256 has a block size of 512 bits, and a digest size of 256 bits. Internally, all values are stored as 32-bit integers, and as such, addition is performed $\bmod 2^{32}$, and all other operations are defined on 32-bit integers. Finally, SHA-256 uses 64 rounds, meaning the compression function is called 64 times for each block.

2.4.1 Initial Values and Constants

The initial value used in the Merkle-Dåmgard construction, IV, is given by eight 32-bit values, denoted $H_n^{(0)}$, $0 \leq n < 8$. The values used were derived by taking the first 32 bits of the fractional parts of the square roots of the first eight prime numbers:

Table 2 - SHA-256 Initial Value

n	$H_n^{(0)}$	$H_{n+1}^{(0)}$	$H_{n+2}^{(0)}$	$H_{n+3}^{(0)}$
0	6a09e667	bb67ae85	3c6ef372	a54ff53a
4	510e527f	9b05688c	1f83d9ab	5be0cd19

The compression function for SHA-256 utilizes a collection of round constants, denoted by K_n, where $0 \leq n < 64$ is the current round number. Their values are given by:

Table 3 - SHA-256 Round Constants

n	K_n	K_{n+1}	K_{n+2}	K_{n+3}
0	428a2f98	71374491	b5c0fbcf	e9b5dba5
4	3956c25b	59f111f1	923f82a4	ab1c5ed5
8	d807aa98	12835b01	243185be	550c7dc3
12	72be5d74	80deb1fe	9bdc06a7	c19bf174
16	e49b69c1	efbe4786	0fc19dc6	240ca1cc
20	2de92c6f	4a7484aa	5cb0a9dc	76f988da
24	983e5152	a831c66d	b00327c8	bf597fc7
28	c6e00bf3	d5a79147	06ca6351	14292967
32	27b70a85	2e1b2138	4d2c6dfc	53380d13
36	650a7354	766a0abb	81c2c92e	92722c85
40	a2bfe8a1	a81a664b	c24b8b70	c76c51a3
44	d192e819	d6990624	f40e3585	106aa070
48	19a4c116	1e376c08	2748774c	34b0bcb5
52	391c0cb3	682e6ff3	5b9cca4f	4ed8aa4a
56	748f82ee	78a5636f	84c87814	8cc70208
60	90befffa	a4506ceb	bef9a3f7	c67178f2

2.4.2 *Functions*

Given below are all the operations defined within SHA-256. Note that by the ∧ operator, we mean bitwise conjunction, ¬ means bitwise negation, and ⊕ means bitwise exclusive or. Furthermore, by $ROTR^n(x)$, we mean bitwise right-rotation by n bits, and by $SHR^n(x)$, we mean bitwise right-shift by n bits. In more familiar notation:

$$\neg x = !\,x$$
$$x \wedge y = x \,\&\, y$$
$$x \oplus y = x \hat{} y$$
$$SHR^n(x) = x \gg n$$
$$ROTR^n(x) = (x \gg n) | (x \ll (32 - n))$$

With these, we can now define the internal functions used by SHA-256. Note that the Ch function may be read as 'choose', while the Maj function may be read as 'majority'. Each argument to the below functions are 32-bit unsigned integers.

$$Ch(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z)$$
$$Maj(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$$
$$\Sigma_0^{256}(x) = ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x)$$
$$\Sigma_1^{256}(x) = ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x)$$
$$\sigma_0^{256}(x) = ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x)$$
$$\sigma_1^{256}(x) = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x)$$

2.4.3 *Message Padding*

As per the Merkle-Damgård specification, the message must be padded before it can be digested. The Pad function for SHA-256 is defined as follows:

1) Append a single 1 bit to the end of the message

2) Append k many 0 bits

3) Append the length of the original message in bits as a 64-bit integer

Since the block size of SHA-256 must divide the length of a padded message, one of the necessary conditions to use the Merkle-Damgård construction, we must have k being the smallest positive integer satisfying $l + 1 + k \equiv 448 \pmod{512}$, where l is the length of the original message in bits. This naturally gives rise to two cases for how $Pad(M)$ works:

- The length of the message is $l \leq 447$, in which case the block can be mutated in-place, with $k = 447 - l$ zeroes being appended to the original message.
- The length of the message is $l \geq 448$, in which case another block must be appended. In this case, we firstly append 512 – $l - 1$ many zeroes to our first block, then construct another block with 448 zeroes, and finally the length of the original message in bits. In total, this requires we append $k = (512 - l) + 447$ zeroes.

After padding has been performed, we can then break the message into 512-bit blocks, each labelled $M^{(i)}$, $0 \leq i < k$. These will be used in conjunction with the initial value $IV = H^{(0)}$ in the compression function. Furthermore, each $M^{(i)}$ can be broken into 32-bit integers, $M_0^{(i)}, M_1^{(i)}, \dots, M_{15}^{(i)}$.

2.4.4 Compression Function

Let our message be given by $M^{(i)}$, which can be further divided into 32-bit words, denoted $M_k^{(i)}$, $0 \leq k < 16$. We define the message expansion, W_t to be:

$$W_t = \begin{cases} M_t^{(i)} & : 0 \leq t < 16 \\ \sigma_1^{256}(W_{t-2}) + W_{t-7} + \sigma_0^{256}(W_{t-15}) + W_{t-16} & : 16 \leq t < 64 \end{cases}$$

Define the working registers, $a, \dots, h$, to be the value of $H_0^{(i)}, \dots, H_7^{(i)}$, i.e. the result from the previous compression function call, or in the case of the first message, the value of IV. We then iterate through every $0 \leq t < 64$:

$$T1 \coloneqq h + \Sigma_1^{256}(e) + Ch(e,f,g) + K_t + W_t$$
$$T_2 \coloneqq \Sigma_0^{256}(a) + Maj(a,b,c)$$
$$h = g$$
$$g = f$$
$$f = e$$
$$e = d + T_1$$
$$d = c$$
$$c = b$$
$$b = a$$
$$a = T_1 + T_2$$

Finally, we update $H_0^{(i+1)} = a + H_0^{(i)}$,..., $H_7^{(i+1)} = h + H_7^{(i)}$. We repeat the above process until we have obtained $H^{(k)}$, which is the SHA-256 hash of the given message.

2.5 Cryptanalysis

Cryptanalysis is a general term for a collection of techniques used to subvert cryptographic primitives. Originally, this term was used in the context of breaking ciphers, and has been used since classic times. In order to determine whether or not a cipher was secure, researchers would attempt to devise ways to, typically using only a set of ciphertext values, or limited number of plaintext-ciphertext pairs, learn information about the key used in the encryption process.

Cryptanalysis has since been generalized to work with a variety of different cryptographic primitives, and is no longer restricted to only block ciphers. More so, since block ciphers are tied to the construction of cryptographic hash functions, the cryptanalysis of hash functions is also possible. Cryptanalytic techniques give a framework with which we can try to break the key properties of hash functions, specifically preimage, second preimage, and collision resistances. Current hash function cryptanalysis typically utilizes one of several well-established frameworks, such as linear cryptanalysis, differential cryptanalysis, and biclique attacks.

2.5.1 *Linear Cryptanalysis*

Many cryptographic primitives are very complex in terms of their structure, which in turn make them very difficult to analyse in their current form. Instead, it may be easier to work with an approximation of the primitive, which may in turn help find non-random behaviours. Specifically, linear cryptanalysis refers to the process of approximating a component as an affine equation, meaning approximating the component as a map $x \to Mx + c$, where M is a linear transformation on some vector space. Linear cryptanalysis has been very successful in the field of block cipher cryptanalysis, with its first use being in the breaking of the FEAL cipher by Mitsuru Matsui and Atsuhiro Yamagishi in 1993[18].

2.5.2 *Differential Cryptanalysis*

More complex systems may be difficult to linearize. This is especially the case if the system is exceptionally random, such as with hash functions. This calls for a more complex framework. Rather than seeking to linearize each component, differential cryptanalysis instead seeks to determine how differences in inputs are propagated through the components of the function. This process was first introduced by Biham and Shamir in their 1993 paper on the cryptanalysis of the DES cipher[14], and known by IBM prior to this time as the "T-attack"[19].

There are several key parts to differential cryptanalysis, with the most primitive being the notion of a difference. As cryptographic hash functions work on the bit-level, the most natural definition for a difference of two values is the XOR operation, $\oplus$. However, in more recent years, more precise definitions for a difference have been introduced. In Wang et al.'s 2005 cryptanalysis of MD5, they introduced the notion of a modular difference[5]. Firstly, this measure of difference did not use the XOR operator; rather, it used subtraction mod 2^{32}. The primary benefit of this choice was that the resulting difference was signed, which allowed information about the inputs to be recovered from only the difference. For example, if two 4-bit integers x, y, gave a modular difference of $2^3 - 2^1 \pmod{2^4}$, then we know:

$$x_3 = 1, y_3 = 0$$
$$x_2 = y_2$$
$$x_1 = y_1$$
$$x_0 = 0, y_0 = 1$$

In comparison, with the same choices for x, y, their XOR difference would be $x \oplus y = 7$, which indicates only that either one of x_3, y_3 are 1, and either one of x_0, y_0 are 0.

Further improvements were made to the notion of difference in Christophe De Cannière and Christian Rechberger's 2006 cryptanalysis of SHA-1[20]. This paper introduced the notion of a generalized difference, which is summarized by the table below:

Table 4 - Generalized Conditions on a Pair of Bits

(x, y)	#	**0**	u	**3**	n	**5**	x	**7**	**1**	−	A	B	C	D	E	?
$(0, 0)$		X		X		X		X		X		X		X		X
$(1, 0)$			X	X			X	X			X	X			X	X
$(0, 1)$					X	X	X	X					X	X	X	X
$(1, 1)$									X	X	X	X	X	X	X	X

To describe the same x, y as given before, we can simply give $\Delta = [\mathrm{u}--\mathrm{n}]$. The set of solutions in Δ are precisely those which were given before. However, this method of describing differences is even more flexible, as it allows for multiple conditions on the same bit to be expressed. However, the price to be paid for this generality is an increase in the amount of space necessary to store this information. To store the XOR difference of two n-bit integers, we need only a single n-bit integer, while with generalized conditions we require a $4n$-bit integer.

Now, consider a function $f : (\mathbb{Z}_2^b)^r \to \mathbb{Z}_2^b$, i.e. a mapping from r many b-bit integers to a single b-bit integer. We say $\Delta_{out} \in (\mathbb{Z}_2^b)^r$ is an output difference of f, corresponding to an input difference $\Delta_{in} \in (\mathbb{Z}_2^b)^r$, if there is some $x, x' \in (\mathbb{Z}_2^b)^r$ such that the difference between x, x' is Δ_{in}, and the difference between $f(x), f(x')$ is Δ_{out}. Using XOR differences, this is the same as saying there is an $x \in (\mathbb{Z}_2^b)^r$ with $f(x \oplus \Delta_{in}) \oplus f(x) = \Delta_{out}$. We say that the propagation of Δ_{in} through f is the set of all Δ_{out} satisfying the above relation. Using XOR differences, this is the same as saying the propagation of Δ_{in} through f is the set $\{ f(x \oplus \Delta_{in}) \oplus f(x) : x \in (\mathbb{Z}_2^b)^r \}$. Furthermore, we can count the frequency of each Δ_{out} to determine the probability that it holds, ρ. In general, the propagation of an input difference through a function is all such output difference and probability pairs, where the output difference satisfies our above relation, and the probability is the frequency at which the output difference occurred.

Since a hash function can be viewed as the composition of a set of components, i.e. $H = u_n \circ u_{n-1} \circ \ldots \circ u_1 \circ u_0$, it is enough to explain how Δ_{in} can be propagated through $u_1 \circ u_0$.

Recall that $(u_1 \circ u_0)(x) = u_1(u_0(x))$, so to begin, we propagate Δ_{in} through u_0, which yields a set X_0, where $(\Delta', \rho_0) \in X_0$ is an output difference corresponding to Δ_{in} through u_0, and has a probability of ρ_0. For each of these pairs, we may then propagate Δ' through u_1 to attain another set, $X_{1,\Delta'}$, where $(\Delta_{out}, \rho_{1,\Delta'}) \in X_{1,\Delta'}$ is an output difference corresponding to Δ' through u_1, and has a probability of $\rho_{1,\Delta'}$. By taking all possible $X_{1,\Delta'}$, we have the propagation of Δ_{in} through $u_1 \circ u_0$ as $X_1 = \{ (\Delta_{out}, \rho) : \exists \Delta'. \Delta_{out} \in X_{1,\Delta'} \}$, where ρ is taken to be the product of all $\rho_{1,\Delta'}$ corresponding to Δ_{out} in $X_{1,\Delta'}$.

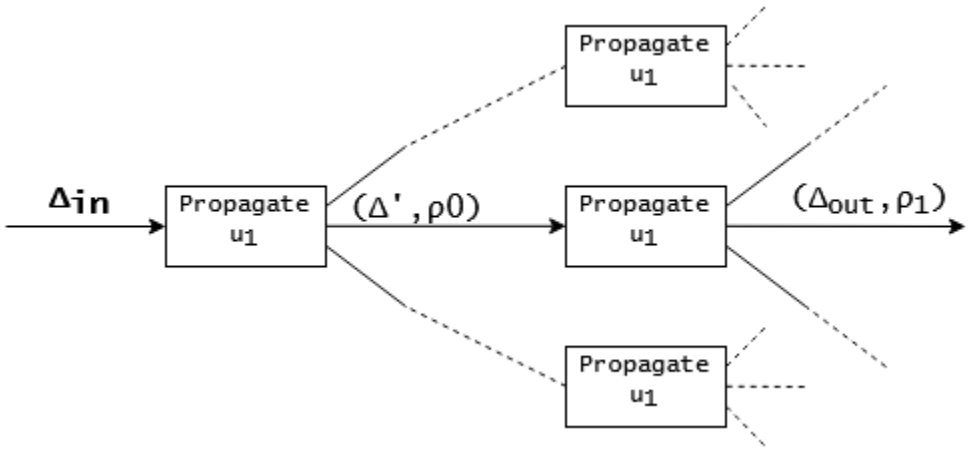

Figure 6 - Propagation of Differences

So, propagation of a difference through a hash function produces a tree, which we will refer to as the propagation tree, in which each branch represents one possible outcome of our difference. The Δ_{in} for a hash function H is typically referred to as the input difference, Δ_0. A path from the root to any leaf of this tree is referred to as a differential characteristic, or more commonly a differential trail.

To perform a differential attack on a hash function H, we first need to select a choice for Δ_0. From here, we can construct the propagation tree, and collect the possible output differences, $(\Delta_{out}, \rho) \in X$. The specific output difference we seek is $\Delta_{out} = 0$; if this is the case, then our two inputs must share the same hash value, as the difference in their hashes is zero. To find a collision, it is then enough to simply randomly sample inputs x, x' such that their difference is Δ_0, and compute their hashes until a collision is detected. However, the likelihood of $\Delta_{out} = 0$ holding, ρ, is likely to be very small, making it unlikely for a randomly chosen input to result in a collision. This is due to the fact that the propagation tree will be very large, and the probability for each branch will be close to uniformly distributed. However, if we place conditions on our choice for x, we are able to manipulate the probability

of each branch, and overall increase the value of ρ. Such conditions are referred to as sufficient conditions.

However, some of these conditions may instead lead to contradictions. For example, by imposing $x_0 = 1$, it may be the case that for one of the branches to hold, we have $x_0 = 0$, which clearly cannot happen. To compensate for this, whenever a condition is imposed, we may update the tree, and at each branch, determine if adding the given condition would lead to a contradiction later on in the tree. If so, the condition may be modified, and its imposition attempted again.

So, the general structure of a differential attack is:

- Select a value Δ_0
- Compute the propagation tree
- Find sufficient conditions which increase the probability of ρ holding, and ensure no contradictions are formed
- Randomly sample inputs x, x' such that they have a difference of Δ_0 and obey the conditions imposed on them

2.5.3 Applications to MD4 and MD5

MD4 is a cryptographic hash function originally designed in Ron Rivest in 1990[1]. Like all other members of the MD family, MD5 uses a Merkle-Damgård structure, with a Davies-Meyer compression function. The block size for this function is 512-bits, with a digest size of 128-bits, and uses only 3 rounds in the compression function.

Shortly after release, public cryptanalysis began. In 1993, Boer and Bosselaers were able to find a semi-free start collision[2], meaning a collision in which the initial value may be chosen freely. Later in 1996, Dobbertin was able to demonstrate a collision on the compression function itself[3]. With these breakthroughs, almost a decade after release, Wang et al was able to produce the first true collision for MD4[21] and MD5[5].

The work of Wang et al. used differential cryptanalysis heavily. Their work furthermore helped to evolve the field even further, with the introduction of the idea of modular differences, a stronger way to measure the difference between any two values. The attack works on a 1024-bit (2-block) message, and has the following input differences:

$$\Delta M_0 = M_0' - M_0 = (0,0,0,0,2^{31},0,0,0,0,0,0,2^{15},0,0,2^{31},0)$$

$$\Delta M_1 = M_1' - M_1 = (0,0,0,0,2^{31},0,0,0,0,0,0,-2^{15},0,0,2^{31},0)$$

With these input differences in mind, Wang et al. were able to derive a set of sufficient conditions for the input difference to hold, and furthermore derived a message modification scheme which allowed for an increase in the probability of a collision occurring. With this, they were able to discover the following MD5 collision for a 1024-bit message:

Table 5 - MD5 Collision by Wang et al.

M_0	2dd31d1c 4eee6c56 9a3d695c f9af9887 b5ca2fab 7e46123e 58044089 7ffbb863 4ad552b3 f4098388 e4835a41 7125e825 51089fc9 cdf7f2bd 1dd95b3c 3780
M_1	d11d0b96 9c7b41dc f497d8e4 d555655a c79a7335 cfdebf06 6f129308 fb109d17 97f2775e b5cd530b aade8225 c15cc79d dcb74ed6 dd3c55fd 80a9bb1e 3a7cc35
M_0'	2dd31d1c 4eee6c56 9a3d695c f9af987b 5ca2fab7 e46123e5 80440897 ffbb8634 ad552b3f 4098388e 4835a41f 125e8255 1089fc9c df772bd1 dd95b3c3 3780
M_1'	d11d0b96 9c7b41dc f497d8e4 d555655a 479a7335 cfdebf0 66f12930 8fb109d1 797f2775 eb5cd530 baade822 5c154c79 ddcb74ed 6dd3c55f 580a9bb1 e3a7cc35
H	9603161f a30f9dbf 9f65ffbc f41fc7ef
H^*	a4c0d35c 95a63a80 5915367d cfe6b751

In this table, we are given the first block, M_0 and M_0' such that $\Delta M_0 = M_0' - M_0$ as provided before, and similarly for M_1 and M_1'. The value H is the hash value found without any padding applied; however, due to the Merkle-Damgård structure, this readily extends to an actual collision, with hash value given by H^*.

2.5.4 Applications to SHA-1

SHA-1 is a family of cryptographic hash functions originally designed by the United States National Security Agency in 1995[10], and were intended to be successors to the SHA-0 hash function. This was supposedly to fix issues in the compression function, which otherwise would reduce the security of the function. Like the MD family, the SHA-1 family uses a Merkle-Damgård structure with a Davies-Meyer compression function. The block size for this function is 512-bits, with a digest size of 160-bits, with the compression function running for 80 rounds.

In 2017, through a joint effort by Centrum Wiskunde & Informatica (CWI) and Google, the first collision for the full SHA-1 was found[12]. The result of this attack was two distinct PDF documents which shared the same SHA-1 hash. According to their initial release on https://shattered.io/, the complexity of this attack "required over 9,223,372,036,854,775,808 SHA1 computations" and "… took the equivalent processing power as 6,500 years of single-

CPU computations and 110 years of single-GPU computations". This success followed from CWI's earlier success in the cryptanalysis of SHA-1 in 2015 in which they demonstrated an algorithm to find a free-start collision for SHA-1 in practical time[11]. Both results used a combination of differential cryptanalysis and other techniques to produce collisions in practical time.

2.5.5 Applications to SHA-2

Research on SHA-2 is still ongoing, with more progress being made every year. The techniques employed by most cryptanalysts for this family is primarily differential cryptanalysis, however much like with CWIs cryptanalysis of SHA-1, other techniques are being incorporated to help make attacks more practical. One such example is the results from Florian Mendel and Mario Lamberger in 2011[22], in which a more exotic technique referred to as second-order differential cryptanalysis[23] was successfully used to break 33 out of 64 steps of SHA-2.

2.5.6 Shortcomings of Differential Cryptanalysis

Unfortunately, the field of hash function cryptanalysis is dominated by experts; there is very little material aimed at those who are new to the field. This is especially true for differential cryptanalysis. This is highlighted by Wang *et al.*'s 2005 paper on the first demonstrated MD5 collision[5]. In this paper, the choice of Δ_0, as well as how all the sufficient conditions were derived, are left unexplained. This is typically the case in all of the literature reviewed for this project. As such, much of what we have discussed has been learnt from trial and error, and slowly piecing together information from across conference papers and postgraduate theses. Given that the security of cryptographic hash functions is of the upmost importance to everyone working in the field of cyber security, and even to those simply using security-related applications, there should be more information on these topics to allow more people to enter the field and contribute their ideas.

2.6 Optimization Algorithms

Many problems in computer science can be phrased as follows: Given a function f in several variables, what is the maximum value attained by f? For clarity, let $f : \Omega \to \mathbb{R}^+$, where Ω is our search space. The maximum value attained would then be $M = \max_{x \in \Omega} f(x)$. However, if Ω is large, or f is difficult to compute, this may not be feasible. Instead, we are interested in an approximation of the maxima, a value $x \in \Omega$ such that it is a local maximum,

meaning if $N \subset \Omega$ is some neighbourhood of x, then $f(x)$ is the maximum value attained in N.

2.6.1 Fitness Functions

Generally, most problems are not of the type $f : \Omega \to \mathbb{R}^+$, rather $f : \Omega \to \Gamma$. As such, the notion of $\max_{x \in \Omega} f(x)$ may not make sense if Γ does not have a total ordering on it. To deal with this, we may use a fitness function, which is a function $h : \Gamma \to \mathbb{R}^+$. In essence, a fitness function h allows us to give every $y \in \Gamma$ a type of ranking, which can then be compared to other values in Γ. This in turn allows us to determine the maximum as $\max_{x \in \Omega} h(f(x))$.

2.6.2 Random Search

The simplest possible approach to searching for the maximum of f is by simply guessing-and-checking solutions. To perform this, we must firstly choose n many values from Ω, denoted x_i, from a uniform distribution. We may then compute $\max h(f(x_i))$. Clearly this algorithm is unlikely to find the global maximum of f, but given a sufficiently large n, may get close. Due to this, we may opt to use random search as a baseline approach when testing other optimization algorithms, or as a way to bootstrap more advanced algorithms.

2.6.3 Hill Climbing

Suppose that our fitness function h is smooth, meaning it does not contain any sudden jumps or sharp points. Rather than randomly sampling values from Ω, we may instead choose a single $x \in \Omega$, and sample small perturbations of x, which we denote x_i. We can then find the maximal perturbation, i.e. the value x_d which maximizes $h(x_d)$, and $h(x_d) > h(x)$. Since f is smooth, if we move our value x in the direction of x_d, we will obtain a larger value. We may repeat this until there are no such $h(x_i) > h(x)$.This technique may be visualized as a point climbing up a hill, to the largest point on that hill. The downside of this technique however is if we are unlucky in our choice of the initial x, we may end up on one of the smaller hills in f, and will then be unable to reach the global maximum.

2.6.4 Genetic Algorithms

Genetic algorithms belong to the class of evolutionary algorithms, processes in which a pool of values are iteratively improved with respect to some fitness function. In this

algorithm, we view our values as genes, and as such, have several possible operations that can be performed:

- On a single gene, we may mutate it in one or more places to generate a new gene
- On a pair of genes, we may choose one or more cross-over points, in which the material of the first gene prior to the cross-over point is concatenated with the material of the second gene after the cross-over point, generating a new gene

As this runs, each gene pool at each iteration slowly improves, with good characteristics from one gene being slowly shared with the others. To prevent gene pool stagnation, mutations occur randomly when generating the next pool, helping prevent the pool from converging to a single value.

2.7 Summary

Due to their security-critical applications, it is necessary to ensure the security of commonly used cryptographic hash functions such as SHA-2. To this end, it is up to cryptanalysts to analyse said functions to determine if they have any exploitable weaknesses, and if so, suggest their deprecation to allow standards and applications to move to new, more secure algorithms. However, the process by which the analysis is done is not entirely clear; this is primarily due to the lack of public knowledge around hash function cryptanalysis, and the lack of any explanatory texts. Furthermore, for people wanting to come into the field, there is no material that they can consult to learn the background material, rather they must read highly technical conference papers and postgraduate theses, and learn from their mistakes. A specific issue we have highlighted is in the framework of differential cryptanalysis; the entire framework is pinned on a singular value, denoted Δ_0, with which all the analysis is possible. However, no recent text has properly explained a process with which to obtain such a value. Some authors even mention this issues in their own papers, such as with Schläffer and Oswald in their 2006 paper: "… Wang *et al.* give input differences, differential paths, and the corresponding conditions that allow to find collisions with a high probability. However, Wang *et al.* do not explain how these paths were found. The common assumption is that they were found by hand with a great deal of intuition"[15].

To this end, this project aims to fix a gap in public knowledge in the field of hash function cryptanalysis by finding a novel way of discovering a good value of Δ_0. This method will apply several optimization algorithms to deal with this non-linear problem, specifically random search, genetic algorithms, and a hybrid approach, and determine whether or not

any of them are suited to the task. To ensure that this is possible in a tractable amount of time, we will use a custom reduced variant of SHA-2, named MAW32. This hash function will share a similar internal structure, but with a drastically reduced complexity, allowing for cryptanalysis in a shorter amount of time.

To determine the success of this project, we will consider several metrics. Firstly, we will determine an appropriate fitness function for a value of Δ_0, and compare the performances between our optimization algorithms by comparing the mean value of this fitness function across all generated values. Furthermore, we will use the random search as a baseline; if either of our two other algorithms perform consistently better than random search, we will consider them successful. To ensure that our fitness function also models the problem of finding a value of Δ_0 such that collisions are likely, we will compare our results to a practical test in finding collisions in MAW32.

3. Design

3.1 MAW32

When beginning this project, we quickly noticed that the majority of the literature was highly technical. Because of this, it was difficult to fully appreciate the content by reading alone. As such, we felt it was appropriate to implement our own hash function alongside implementing SHA-256, to get a better understanding of how they work

The MAW32 hash function can be considered to be a reduced SHA-2 variant. It shares a similar internal structure to the SHA-2 family, however due to size constraints lacks the Ch component present in all other SHA-2 hash functions. It uses the Merkle-Damgård structure, with a Davies-Meyer compression function. The block size of this function is 64-bits, with a hash size of 32-bits. Due to this small hash size, it is immediately vulnerable on modern computers to the birthday attack, which can readily find collisions in 2^{16} calls to the hash function. Unfortunately, MAW32 was not initially designed to specifically be a reduced variant of SHA-256, and as such contains some minor differences which will be explained later on.

This reduced size comes at a benefit however; since the search space is so small, it is feasible to find collisions in a tractable amount of time, allowing new ideas and conjectures to be easily tested. Furthermore, due to the shared internal structure, we conjecture that attacks that can be employed on MAW32 may also be extensible up to the SHA-2 family. We take advantage of this when exploring different techniques for finding differential characteristics in MAW32.

3.1.1 Initial Values and Constants

The initial value used in the Merkle-Damgård structure is derived from the fractional expansion of π (OEIS A062964), with

$$H_0^{(0)} = 24, H_1^{(0)} = 3f, H_2^{(0)} = 6a, H_3^{(0)} = 88$$

This was originally chosen for simplicity, and is one of the divergences from SHA-2, which uses the square roots of the first 8 prime numbers as its initial value. However, in the framework of differential cryptanalysis, this has no major effect on our results. The round constants are denoted by K_n, where $0 \leq n < 16$ is the current round number. Their values are derived from the fractional expansion of e (OEIS A170873):

Table 6 - Round Constants for MAW32

n	K_n	K_{n+1}	K_{n+2}	K_{n+3}
0	$b7$	$e1$	51	62
4	$8a$	ed	$2a$	$6a$
8	bf	71	58	80
12	$9c$	$f4$	$f3$	$c7$

As before, this was chosen for simplicity, and is another one of the divergences from SHA-2, which uses the cube roots of the first 64 prime numbers as its round constants. These values also have no major effect on the framework of differential cryptanalysis.

3.1.2 Functions

As in SHA-2, MAW32 defines several functions:

$$SHR^n(x) = x \gg n$$
$$ROTR^n(x) = (x \gg n) | (x \ll (8 - n))$$
$$Maj(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$$
$$\sigma_0(x) = ROTR^2(x) \oplus ROTR^3(x) \oplus ROTR^5(x)$$
$$\sigma_1(x) = ROTR^1(x) \oplus ROTR^4(x) \oplus SHR^3(x)$$

It should be noted that each of these functions take 8-bit integers as input, and produce 8-bit integers as output.

3.1.3 Message Padding

Since we are designing a Merkle-Damgård hash function, the message must be padded before it can be digested. In MAW32, we define the Pad function as follows:

1) Append a single 1 bit to the end of the message

2) Append k many 0 bits

3) Append the length of the original message in bits as a 32-bit integer.

Since the block size of MAW32 must divide the length of a padded message, one of the necessary conditions to use the Merkle-Damgård construction, we must have k being the smallest positive integer satisfying $l + 1 + k \equiv 32 (\mathrm{mod}\ 64)$, where l is the length of the original message in bits. This naturally gives rise to two cases for how $Pad(M)$ works:

- The length of the message is $l \leq 31$, in which case the block can be mutated in-place, with $k = 31 - l$ zeroes being appended to the original message.
- The length of the message is $l \geq 32$, in which case another block must be appended. In this case, we firstly append $64 - l - 1$ many zeroes to our first block, then construct another block with 32 zeroes, and finally the length of the original message in bits. In total, this requires we append $k = (64 - l) + 33$ zero bits.

After padding has been performed, we can then break the message into 64-bit blocks, each labelled $M^{(i)}$, $0 \leq i < k$. These will be used in conjunction with the initial value $IV = H^{(0)}$ in the compression function. Furthermore, each $M^{(i)}$ can be broken into 8-bit integers, $M_0^{(i)}, \dots, M_7^{(i)}$.

3.1.4 Compression Function

Let our message be given by $M^{(i)}$, which can be further divided into 8-bit words, denoted $M_k^{(i)}$, $0 \leq k < 8$. We define the message expansion, W_t to be:

$$W_t = \begin{cases} M_t^{(i)} & : 0 \leq t < 8 \\ \sigma_0(W_{t-3}) + W_{t-4} + \sigma_1(W_{t-8}) & : 8 \leq t < 16 \end{cases}$$

Define the working registers, a, b, c, d, to be the value of $H_0^{(i)}, H_1^{(i)}, H_2^{(i)}, H_3^{(i)}$, i.e. the result from the previous compression function call, or in the case of the first message, the value of IV. We then iterate through every $0 \leq t < 16$:

$$T1 := d + \sigma_1(b) + K_t + W_t$$
$$T_2 := \sigma_0(a) + Maj(a, b, c)$$
$$d = c$$
$$c = b + T_1$$
$$b = a$$
$$a = T_1 + T_2$$

Finally, we update $H_0^{(i+1)} = H_0^{(i)} + a, \dots, H_3^{(i+1)} = H_3^{(i)} + d$. A diagram of MAW32 is supplied below.

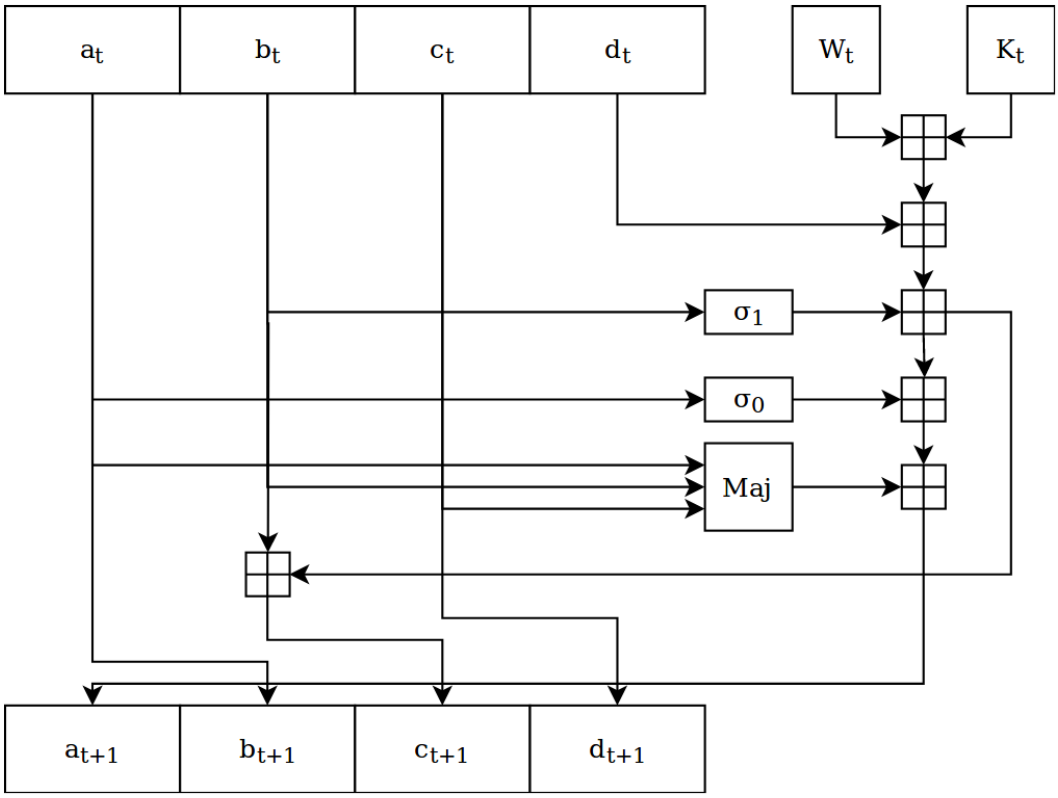

Figure 7 - Diagram of MAW32

3.2 Trail Searching

Now that we know about the internal structure of both SHA-2 and MAW32, how can we search for differential trails? There are three key parts to this: propagating an input difference through a component, propagating an input difference through the hash function itself, and finally selecting an input difference.

3.2.1 Propagating through a Component

As discussed in the background of differential cryptanalysis, the result of a propagation through a component is all possible output differences corresponding to an input difference through the component, along with their respective probabilities. First, we must discuss our choice of difference. In this project, we have opted to use XOR difference. This has the downside of not being very descriptive, but is much simpler to use, has a smaller memory footprint, and is overall easier to understand. In comparison, modular differences and generalized conditions on bits are very descriptive methods, but the way to propagate them efficiently through a component is less clear.

In MAW32 and SHA-2, there are two classes of components; linear and non-linear. We say a component is linear if it is a semigroup homomorphism on $(\mathbb{Z},\oplus)$; specifically, if f is our component, then $\forall x, y \in \mathbb{Z},\ f(x \oplus y) = f(x) \oplus f(y)$. Now, for XOR, to propagate an input difference Δ through f, we compute $\forall x \in \mathbb{Z},\ f(x \oplus \Delta) \oplus f(x)$, but $f(x \oplus \Delta) \oplus f(x) = f(x) \oplus f(\Delta) \oplus f(x) = f(\Delta)$ is constant. Hence if a component is linear, propagation can be done in $O(1)$ time. An example of such functions are the σ and Σ functions in SHA-2 and the σ functions in MAW32.

A non-linear component does not have this property however. A simple example is modular addition; below, we have shown the results of addition $\bmod 2^3$ with a fixed input difference $\Delta = (3,4)$. In two variables, our propagation would be $f(x \oplus \Delta) \oplus f(x) = \big((x \oplus \Delta_x) + (y \oplus \Delta_y)\big) \oplus (x + y) = \big((x \oplus 3) + (y \oplus 4)\big) \oplus (x + y)$

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	7	1	3	1	7	1	3	1
1	7	5	3	5	7	5	3	5
2	7	5	3	5	7	5	3	5
3	7	1	3	1	7	1	3	1
4	7	1	3	1	7	1	3	1
5	7	5	3	5	7	5	3	5
6	7	5	3	5	7	5	3	5
7	7	1	3	1	7	1	3	1

So, the result of propagating $\Delta = (3,4)$ through addition $\bmod 2^3$ is $\left\{\left(1,\frac{1}{4}\right),\left(3,\frac{1}{4}\right),\left(5,\frac{1}{4}\right),\left(7,\frac{1}{4}\right)\right\}$.

If the search space is large however, a brute-force approach will not work. In this case, we may opt for random sampling. In MAW32, there is only one function which is too large to brute-force, which is $Maj(x,y,z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$. The search space for this function has 2^{24} elements, which takes roughly a second to compute on a single processor. Instead, we opt to randomly sample 2^{16} values for (x,y,z), and propagate with only these. This brings the time taken for propagation down to roughly 2 milliseconds. Uniform random sampling is the best choice for this, as any other process may reveal inherent patterns in the propagation. For example, if we sampled the above by taking only the values

of x that were a multiple of 4, we would end up with only 7 as the output difference, which is clearly different compared to the true propagation. However, there are still 2^{24} possible input differences for Maj, and this is still unfortunately a bottleneck in the long run. To deal with this, we opt to pre-compute the propagations for all possible Δ for both key mixing, modular addition, and Maj. This data can be produced on demand by the utility $trail\_gen$, and takes roughly 8 hours to run. The source code for this utility can be found through the link provided in appendix A.

After performing a propagation, we have all possible output differences, along with their respective probabilities. However, in many cases, there are a large number of output differences, each with a very low probability of occurring. Due to this low probability, we do not need to visit them while traversing the propagation tree, and opt to discard them prematurely. To this end, we define the threshold probability, 2^{-n}, such that if $\rho < 2^{-n}$, the output difference is discarded.

3.2.2 Propagating through a Hash Function

As covered in the background discussion, we can view the entire hash function as the composition of several components, and as such, propagation forms what we refer to as the propagation tree. When performing a propagation, we may dictate how many rounds to propagate to, for instance round 8 of 16. This allows us to limit the size of the tree generated by only propagating the hash function up to the end of that specific round. Furthermore, we have that every branch in the propagation tree has a probability larger than that of the probability threshold. To efficiently generate this tree, we use a backtracking algorithm, described below:

1) Initialize the state stack, which contains information about the next propagation to perform, with the state of the root of the tree
2) While the state stack is nonempty, pop the top element from the stack and attempt to continue its propagation
3) If the result of the next propagation is empty, we abort this propagation and go back to 2
4) If the result of the next propagation is a single value, we update our propagation and continue propagation
5) If the result of the next propagation is more than a single value, we put a copy of the current state, along with the propagations still to be considered on the stack, and continue propagating.

This algorithm allows us to efficiently generate and traverse the tree while also minimizing memory usage. While it is possible to implement this as a recursive algorithm, this is not extensible to more complex functions due to limited stack sizes. Whenever a propagation reaches the final round, we observe the value of the registers it has propagated so far; if they are all zero, we have found a trail through the propagation tree which leads to a collision, and we may update our statistics accordingly.

3.2.3 Selecting and Input Difference

With a framework for performing a propagation in mind, we must now consider how to produce a suitable value for Δ_0. The first and most immediate choice is random search. By randomly picking a $\Delta_0 \in \mathbb{Z}_2^b$ and attempting to propagate it, we can measure the probability of the value giving an output difference of zero. With this, we can easily determine how suitable a given Δ_0 is for performing a differential attack. However, since propagation is an EXPSPACE problem, this alone is not feasible. Depending on the choice of Δ_0, it may take up to one minute on a single processor to complete a propagation. Due to this, we opt to restrict our choices for Δ_0.

Our first restriction is, Δ_0 should consist of a dense and sparse section. In the sparse section, it should be nothing but 0, while in the dense section, it may be anything. In MAW32, we opt for the sparse section to be the first 4 bytes, with the dense section being the last 4 bytes. This is commonly used in hash function cryptanalysis, such as in Mendel[24]: “As in the attack of Nikolić and Biryukov we are interested in differential characteristics with differences in only a few message words. Then, large parts of the expanded message have no

difference which in turn, results in an attack on more than 24 steps". We also place a restriction on the message expansion itself. Specifically, we require at least two out of eight of ΔW_t for $8 \leq t < 16$ to be zero, i.e. at least two non-trivial message expansions must have a difference of zero. This helps to reduce the overall number of branches in the tree, helping to improve the efficiency of propagation.

3.3 Trail Optimization

Before we can begin optimizing a choice of Δ_0, we must first define a fitness function for it. This fitness function should represent how likely a choice of Δ_0 is to produce a collision in MAW32. There are two choices, which we will refer to as the naïve fitness function and probability fitness function.

3.3.1 Naïve Fitness Function

After propagation has occurred, we will have n many differential trails in the propagation tree. Of these n many trails, $0 \leq k \leq n$ will correspond to trails of the form $\Delta_0 \to 0$, i.e. trails which correspond to an output difference of 0. By assuming that every branch has roughly the same probability of being visited, we can define the fitness of an input difference Δ_0 to be $\frac{n}{k}$, where n, k are the values described before applied to the propagation tree of Δ_0. To interpret this result, we say that if Δ_0 has a fitness of $\frac{n}{k}$, and Δ_0' has a fitness of $\frac{n'}{k'}$, then Δ_0 is a better input difference if $\frac{n}{k} > \frac{n'}{k'}$. The benefit of using this fitness function is that it can be applied to any tree of any size without loss of precision; rather than storing a floating point number, we may store the values n, k as unsigned integers, and use the fact that $\frac{n}{k} > \frac{n'}{k'} \iff nk' > n'k$ to avoid floating point imprecision. The downside of using this fitness function however is our assumption that each branch has roughly the same probability of occurring; this is unlikely to be the case in general, and as such, may bias results.

3.3.2 Probabilistic Fitness Function

When propagating a choice of Δ_0, we construct the propagation tree; in this tree, every branch is assigned a probability in [0,1], which represent the likelihood of a randomly chosen pair of inputs differing by that amount. Assuming independence, the probability of specific path in the tree occurring is simply the product of the probability of each branch. So, each zero trail $\Delta_0 \to 0$ holds with some probability ρ. To determine how likely a collision is to

occur for our input difference, we may simply sum these values of ρ for all possible zero trails in the propagation tree. Since these events are typically not independent, this gives a lower bound on the probability. The downside to using this function is floating point imprecision; the probability of an arbitrary trail holding for 8 rounds of MAW32 is typically around 2^{-50}; however, the smallest non-zero number that can be stored in a double-precision floating point number is $2.22045 \times 10^{-16} \approx 2^{-52}$. As such, if any of our choices of ρ are less than 2^{-52}, they will be rounded to zero, and hence discarded. If many of these trails exist, then we may introduce imprecision, which in turn causes a bias towards lower fitness', influencing our result negatively.

Equipped with these functions, we may now describe our methods for finding our choices of Δ_0. As described earlier, we may randomly select a value of Δ_0 by assigning the first 4 bytes as 0, and randomly assigning the last 4 bytes. To increase the likelihood of collision, we also ensure that the message expansion is 0 for at least 2/8 of the message expansions.

To decide which optimization algorithms are applicable to this problem, we first must ask if either of our fitness functions are continuous. If so, then a potential solution would be hill climbing or gradient descent, which require a smooth fitness function to work. However, since this problem is discrete, it is not feasible to apply gradient descent, and we conjecture that both of our fitness functions are not well behaved, hence hill climbing is not a possible candidate. However, genetic algorithms are applicable to non-linear problems such as this, so we opt to use this method.

3.3.3 True Fitness Function

The purpose of a fitness function is to model some desirable behaviour when choosing values. However, what is the behaviour we are trying to select for when choosing values for Δ_0? We say that one input difference is better than another if it is able to, over a large period of time, effectively compute more collisions than the other. More aptly, we could say that the input difference with the most collisions per second is optimal. Using this, we will define the true fitness function for selecting input differences, as well as a suitable approximation.

First, note that the zero difference will always result in a trivial collision when we try to perform a differential attack; however, most input differences will not have such a quality. Let $C(\Delta, t)$ denote the number of collisions found using the input difference Δ after t seconds. We then define the true fitness of Δ to be $F(\Delta) = \lim_{t\to\infty} \frac{C(\Delta,t)}{C(0,t)}$, informally, as the ratio of

collisions found between Δ and the zero difference, if we were to run the test forever. Clearly $F(\Delta) \in [0,1]$, and in fact represents the probability that Δ will successfully create a collision for an arbitrary input. By fixing various values of t, we can approximate F to a high degree of certainty.

To approximate $F(\Delta)$, we perform the following:

1) Fix a value for $t > 0$, which we will refer to as our timeout

2) Choose an input difference Δ_0

3) Run the program $hasher\ diff\ maw32\ 8\ \Delta_0$ for t seconds, and record the results

4) Count the number of collisions found by $hasher$ before the timeout has elapsed

5) Repeat from 2

Given a sufficiently large value of t, these results will follow the same distribution as $F(\Delta)$, which is sufficient for our testing. For the purposes of our testing, we use the approximation where t = 10, which is due to the time limitations imposed by an honours project.

3.3.4 Genetic Algorithms for MAW32

To begin, we must first formalize what a gene is for our problem. Since our values for Δ_0 can be viewed as n-bit integers, this is already a suitable candidate for a gene. More generally, we can view a gene as an array of n bits. Although there may be other ways to model this problem, we have opted for this as it is the simplest to work with. To perform a mutation on a gene, we may simply flip a random bit in the gene. To ensure the new gene is viable, we must also try to propagate it, and if it has a non-zero chance of finding a collision, we keep it, otherwise it is discarded, and we try again. To cross-over two genes, we fix a single midpoint. We copy the first gene up to the midpoint, copy the second gene from the midpoint to the end, and use these snippets to create a new gene. To ensure non-trivial results, we ensure that the midpoint is in the dense section of both genes. As before we also try to propagate the new gene to ensure it is viable, and discard it if it is not.

Our genetic algorithm can be described as follows:

1) Initialize the gene pool with randomly generated solutions
2) Randomly delete half of the genes from the pool, with a bias towards deleting those with a low fitness with respect to our fitness function of choice
3) When repopulating the pool, we have a $1/16$ chance to mutate a random gene, otherwise we attempt to cross over two randomly selected distinct genes, with a midpoint chosen to be somewhere in the dense section.
4) When the pool is full again, compute the maximum, and repeat from 2

One issue we immediately notice is that mutation alone is unlikely to generate a valid gene; as such, we also consider what we refer to as the hybrid algorithm. In this algorithm, we effectively run the same algorithm, except every time we delete half of the pool, we require a number of genes to be added the pool. We refer to this technique as immigration. This helps to prevent stagnation in genetic material, which is the task that mutation usually performs. The number of genes introduced every generation can be configured by altering the size of the gene pool and immigration rate.

However, creating random solutions every generation is a computational bottleneck. To rectify this, at the program start, we spawn a number of threads configurable by the user. These worker threads will continually attempt to generate new random solutions, and when found, add them to a global queue which can later be accessed by the main thread. This helps to reduce bottlenecking from the immigration process, and also helps to speed up the initial pool creation.

3.3.5 Random Number Generators

A crucial component of this entire process is the ability to generate random numbers. To this end, there are several tools we can use. The C language provides a simple random number generator, which we will refer to as the C Random Number Generator (CRAND). The default implementation is a linear congruential generator, which obeys the following recurrence relation:

$$x_{n+1} = ax_n + c \pmod{m}$$

In this relation, a, c, m are all taken to be fixed constants, typically large 32-bit integers, and x_0 is provided as a seed value. However, due to this simple relation, linear congruential generators, hence CRAND as well, exhibit a very low periodicity. We say that a random number generator has periodicity p if, for all possible n, $x_{n+p} = x_n$, i.e. the random number

generator repeats after at most p calls. In the case of CRAND, the periodicity is $p = 2^{32}$. Repeating random numbers is typically undesirable, and due to the amount of randomness required in simulations such as this, is something that should be avoided.

To this end, for this simulation we opt to use a Mersenne Twister. This is provided by the C++ standard library as $std::mt19937$ (MT19937), due to the fact that it has a periodicity of $p = 2^{19937} - 1$. This makes it much more suitable for simulations, however is still not optimal. For the purposes of our testing however, we deem it sufficient. As with all random number generators, MT19937 requires an initial seed to produce random numbers. On a Linux system, this can be provided by polling the device $/dev/urandom$, which provides cryptographically secure random numbers. However, for the purposes of portability and a reduction in complexity, we opt instead to use $std::random\_device$, again provided by the C++ standard library.

3.4 Performing an Attack

Trail optimization is implemented by a utility named maw_trail, which implements both fitness functions and all three techniques described earlier. The result of this utility is a collection of input differences, Δ_0. Given an input difference Δ_0, how can we attempt to find a collision? Recall that an input difference Δ_0 is a value such that $H(x) \oplus H(x \oplus \Delta_0) = 0$ with high probability. Rewriting, this means $H(x) = H(x \oplus \Delta_0)$ with high probability. So, to perform a differential cryptanalytic attack, we apply the following algorithm:

1) Randomly sample the input space for blocks, x

2) Compute $x' = x \oplus \Delta_0$

3) Compute $H(x)$, $H(x')$

4) If $H(x) = H(x')$, then we have detected a collision (x, x')

5) Repeat from 1

If Δ_0 holds with probability 2^{-n}, then we anticipate having to sample 2^n values to find a collision. This basic algorithm has been implemented in the *hasher* function. This is not efficient however; to increase the likelihood of finding a collision, it is necessary to determine a set of sufficient conditions for the input difference. With these conditions, we may then modify our two inputs to ensure they satisfy the given conditions, which will in turn increase the chance of finding a collision.

4. Results

To test the efficacy of our techniques, we have run several tests to determine how they perform in relation to each other, and in relation to finding actual collisions. Specifically, we have tested the efficacy of both the naïve fitness function and the probabilistic fitness function across the three different generation techniques, specifically random search, genetic algorithms, and our hybrid algorithm. We will give a full description of how these tests were carried out, including conditions, assumptions, and parameters for all programs used.

Firstly, all of our tests are performed on MAW32 to ensure they can be run in a tractable amount of time. Another benefit of this is we will also produce a very large quantity of data, which increases the certainty of our answers. The results of this analysis may then be used to determine the efficacy of our techniques on the SHA-2 family of hash functions. To ensure a fair and unbiased test, we impose the following conditions:

- All tests are to be run on the same computer, provided by the University
- At no point should any other test be running simultaneously
- The test will run uninterrupted for exactly 24 hours
- None of the memoization files will be changed at any point
- The configuration for maw_trail is as follows:
    - Rounds: 8/16
    - Probability threshold: -3.000000
    - Immigration rate: 5%
    - Gene pool size: 64

Overall, 6 tests will be run, testing all combinations of generation techniques and fitness functions. To analyse these results, we opt to format the output as a CSV. This is not immediately supplied by maw_trail, which also includes important diagnostic information in its output, so it is necessary to perform a simple search-and-replace. The data from maw_trail that we are interested in is of the form “[Timestamp] (Fingerprint: %1, Fitness: %2) - %3”, where “%n” denotes some variable. We reformat these lines to be of the form “%1,%2,%3”, meaning the columns are the fingerprint, fitness, and technique respectively. The fingerprint column should contain a 64-bit hexadecimal integer, the fitness column should contain some integer, and the technique column should contain one of the following values:

Immigration, Survivor, Generated, Best. However, the "Survivor" and "Best" techniques are themselves diagnostic outputs, so should be discarded.

To properly analyse our results, we derive two such CSV files from any given output. The first file contains all of the data of the form described above. However, this file may contain a large number of duplicates, and as such, we derive a second file which consists of only the unique input differences that were generated. When performing statistical analysis, this will be performed on the second, unique collection of data, to avoid massive amounts of bias towards the genetic algorithm and hybrid algorithm.

However, we must also gauge how effective each of our techniques are at producing viable input differences, i.e. input differences which are capable of finding collisions readily. To do this, we will use the unique results from each test, and compute an approximation of the true fitness function, with a timeout of $t = 10$. This is facilitated by a simple Python script, $test\_diffs.py$. This script takes a CSV file of the form described above, and uses the $hasher$ utility to attempt to find actual collisions in MAW32. The result of this script is another CSV file in which the fitness column now contains the number of actual collisions found by the hasher utility.

To analyse our data, we will use some elements from statistical analysis. A brief description of each technique is given below. Firstly, for each data set, we will consider the mean, μ. This is a measure of central tendency, i.e. a value which represents what a typical value looks like in the data set. This can be used to determine what the average fitness of an input difference generated by that technique may be. The standard deviation, σ, gives a notion of volatility, specifically it measures how close data lies to the mean μ. A large value for σ indicates that values are spread very loosely around the mean, indicating that the technique may not be able to repeatedly generate such values. Although we may readily compare the means of two different data sets under the same fitness function, the same is not true for the standard deviation. However, we can compare the coefficient of variation, $c_v = \frac{\sigma}{\mu}$, which is effectively the normalized standard deviation. If the coefficient of variation for one data set is significantly larger than another, we can infer that the first set is much more loosely spread out than the latter. This indicates that the first set is less likely to repeatedly produce input differences with a high fitness, in comparison to the latter.

To compare a fitness function to the true fitness function, we will use a visual comparison. Although there exist techniques to determine if two data sets are 'similar looking' to another, they are not sufficient for our purposes. One such test is the

Kolmogorov-Smirnov test, which determines if two data sets are drawn from the same distribution as each other. However, in the case that a data set is sufficiently large, almost all tests will fail, as even minor discrepancies will be amplified by the sheer amount of data. As such, we will compare the two data sets visually. This is done by first normalizing both data sets by dividing by their maximal values, and then producing a simple histogram with both data sets superimposed on each other. We may then inspect this visualization to determine if there are any major discrepancies between the two data sets.

4.1 Naïve Fitness Function

4.1.1 Statistical Summary

Given in the tables below are a summary of the tests performed on the naïve fitness function. The second table gives the results from the true fitness function on the unique data from each test. Values are given to 6 decimal places where appropriate.

Table 7 - Statistical Summary for Naive Fitness Function

	Results	**Unique Results**	**Mean**	**Standard Deviation**	**Coefficient of Variation**
Random	5725	5435	0.006337	0.008501	1.341328
Generated	454186	1567	0.021079	0.015624	0.741186
Hybrid	94241	6766	0.010691	0.013681	1.279637

Table 8 - Statistical Summary for True Fitness of Naive Fitness Function

	Unique Results	**Mean**	**Standard Deviation**	**Coefficient of Variation**
Random	5435	16.485094	68.912978	4.180321
Generated	1567	14.275862	71.764724	5.026998
Hybrid	6766	20.303917	67.523848	3.325656

Overall, random search produced the least number of results over the 24-hour period. However, compared to the other techniques, it had the lowest proportion of duplicated results, with a duplication rate of only 5.1%. The average fitness of an input difference generated using this technique was very low, at only 0.00634 on average. Furthermore, the spread of the data was the highest out of all techniques, with a coefficient of variation of 1.3413, indicating a high level of spread around the mean. We see that the random search technique, while it does produce possible input differences, does so in a highly unpredictable manner, and with a very low fitness overall.

Almost the opposite is true for the genetic algorithm however. With over 450,000 total results, only 1567 are unique, giving a duplication rate of over 99.6%. This is likely due to the gene pool quickly converging to a local maximum, causing the pool to become stale. However, the values it generated were on average much better than those from the random search technique, with an average fitness of over 0.0210. Furthermore, with a coefficient of variation of only 0.74119, the data is far less spread out than the random search technique, indicating that the generated data is situated much more closely around the mean. As such, we see that the genetic algorithm is able to produce input differences of a high fitness, but is prone to getting stuck on a particular solution.

Finally, as the hybrid algorithm is a mix of the two, it inherits both positive and negative traits from both techniques. Although it produced less solutions than the genetic algorithm overall, a far higher proportion of them were unique, with a duplication rate of only 92.8%. However, this also caused a decrease in the average fitness at only 0.0107, and additionally increased the amount of spread, with a coefficient of variation of over 1.2796. As such, we see that the hybrid algorithm is able to take some of the positive features of the genetic algorithm such as higher average fitness, but also inherits some of the negative features of random search such as high spread. Overall, we see that the hybrid algorithm is inferior to the genetic algorithm.

Comparing the results from the naïve fitness to the approximation of the true fitness, we notice some disparities. Firstly, we see that the results are almost reversed. While the technique with the highest naïve fitness was the genetic algorithm, it had the lowest true fitness, with the highest spread of all. Conversely, while the lowest naïve fitness was random search, it fared well with respect to the true fitness, and surprisingly the best technique was the hybrid algorithm, which had a true fitness of approximately 20.30. As such, we see that the naïve fitness function may not appropriately represent the true fitness of an input difference, however we will investigate this further with our visualizations of data.

4.1.2 *Visualization of Data*

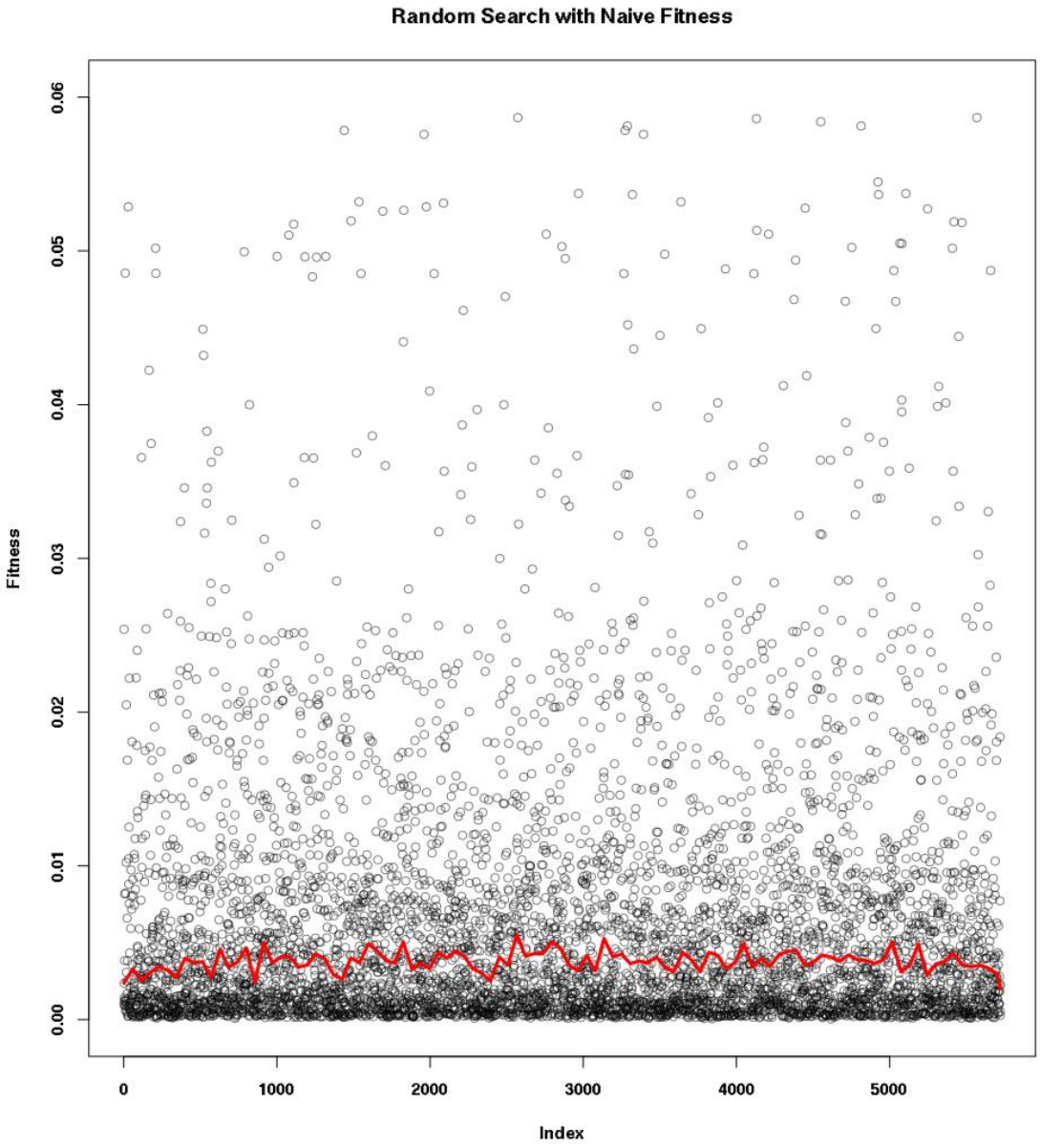

Figure 8 - Scatterplot of Random Search Results with Naïve Fitness

In our above diagram, we see the naïve fitness of all input differences generated by the random search technique, with a non-linear trendline given in red. The majority of points are clustered fairly close to the mean of 0.0063, however there are a number of generated values with a far higher fitness. The trendline for this set of data lies close to this mean as well, with a fair amount of variance. Overall, this indicates that while the random search technique is able to produce a large amount of unique data, the typical fitness with respect to the naïve fitness function is very low, as noted in our statistical summary.

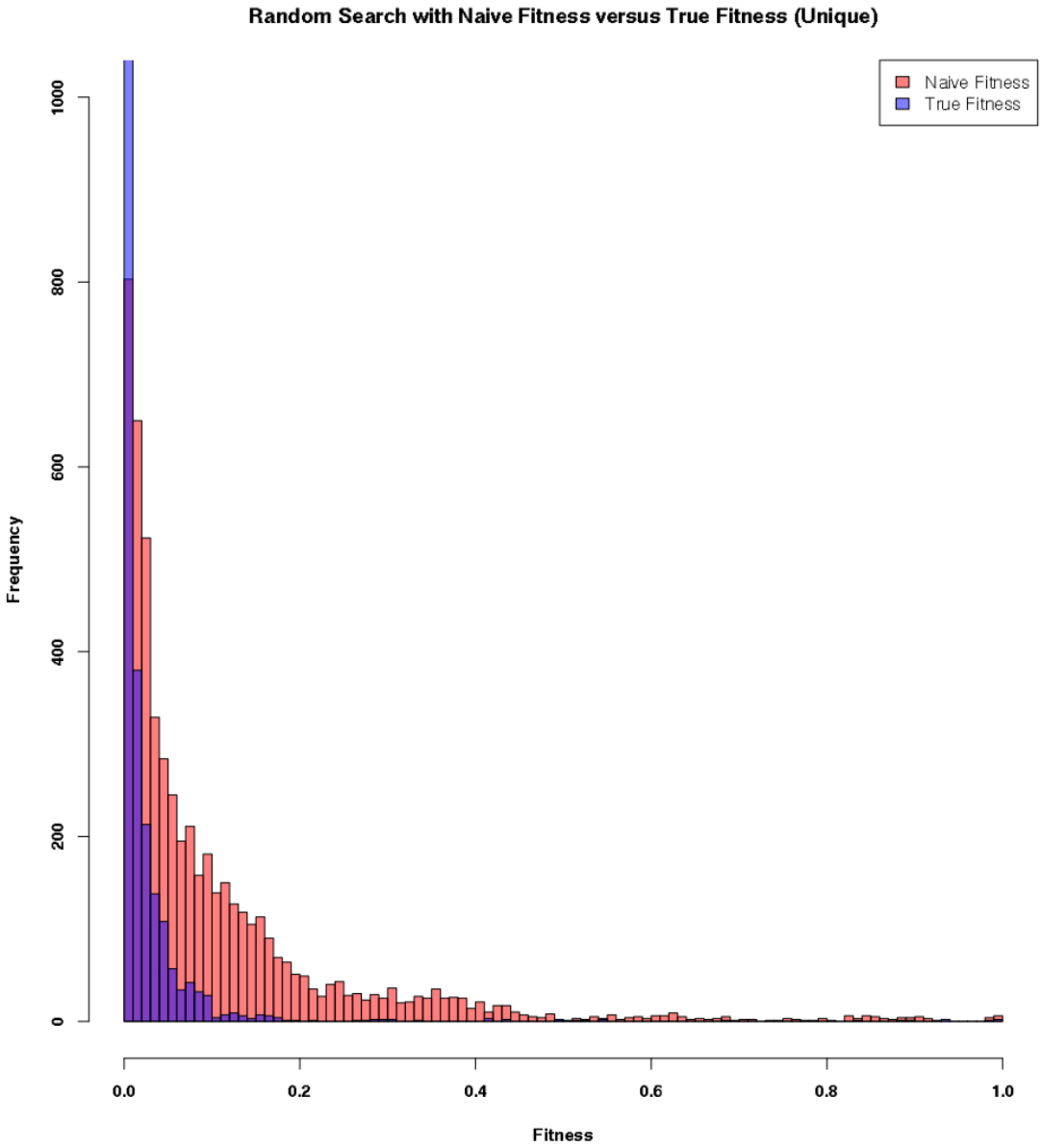

Figure 9 - Histogram of Unique Random Search Results with Naïve Fitness

In this diagram, we take the naïve fitness of the unique data, and compare it to that of the true fitness, given in red and blue respectively. We can immediately see that the naïve fitness function grossly overestimates the true fitness of the generated input differences, causing a large amount of skew in comparison to the true fitness. Furthermore, the naïve fitness function has a considerable amount of data to the right of our diagram, whereas the true fitness function is not even visible at this point. As such, we see that the naïve fitness function may not be a good representation of the true fitness of an input difference, due to it overrepresenting the true fitness of an input difference. This in turn may cause our other techniques to select for the wrong qualities.

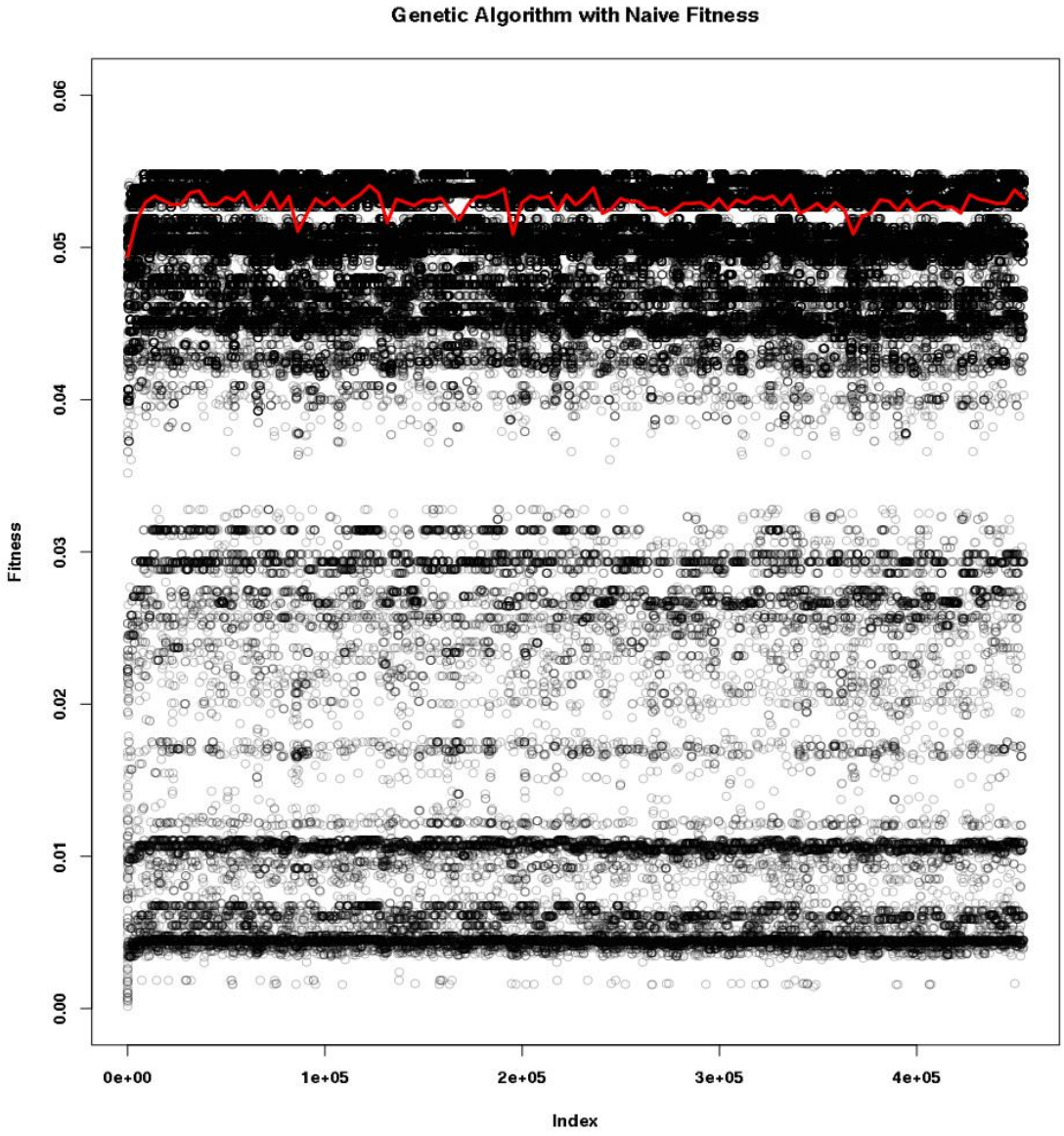

Figure 10 - Scatterplot of Genetic Algorithm Results with Naive Fitness

The genetic algorithm was able to produce a massive quantity of data over the 24-hour period it ran. We can see that for the first thousand values, the fitness of the gene pool was at its lowest, with the trend line starting at approximately 0.05. However, this quickly improved, and settled at approximately 0.052, with fairly little variance. One distinguishing feature of this technique is the banding; there are very thick lines of points present at the marks 0.005, 0.01, 0.03, and from 0.04 to 0.055, with the majority of points being present in the final band. This is likely due to the genetic algorithm repeatedly constructing the same input differences over and over as part of crossing-over two genes. Overall, we see that the genetic algorithm is able to produce a large quantity of data of very high fitness, with respect to the naïve fitness function, however appears to converge to particular values very quickly.

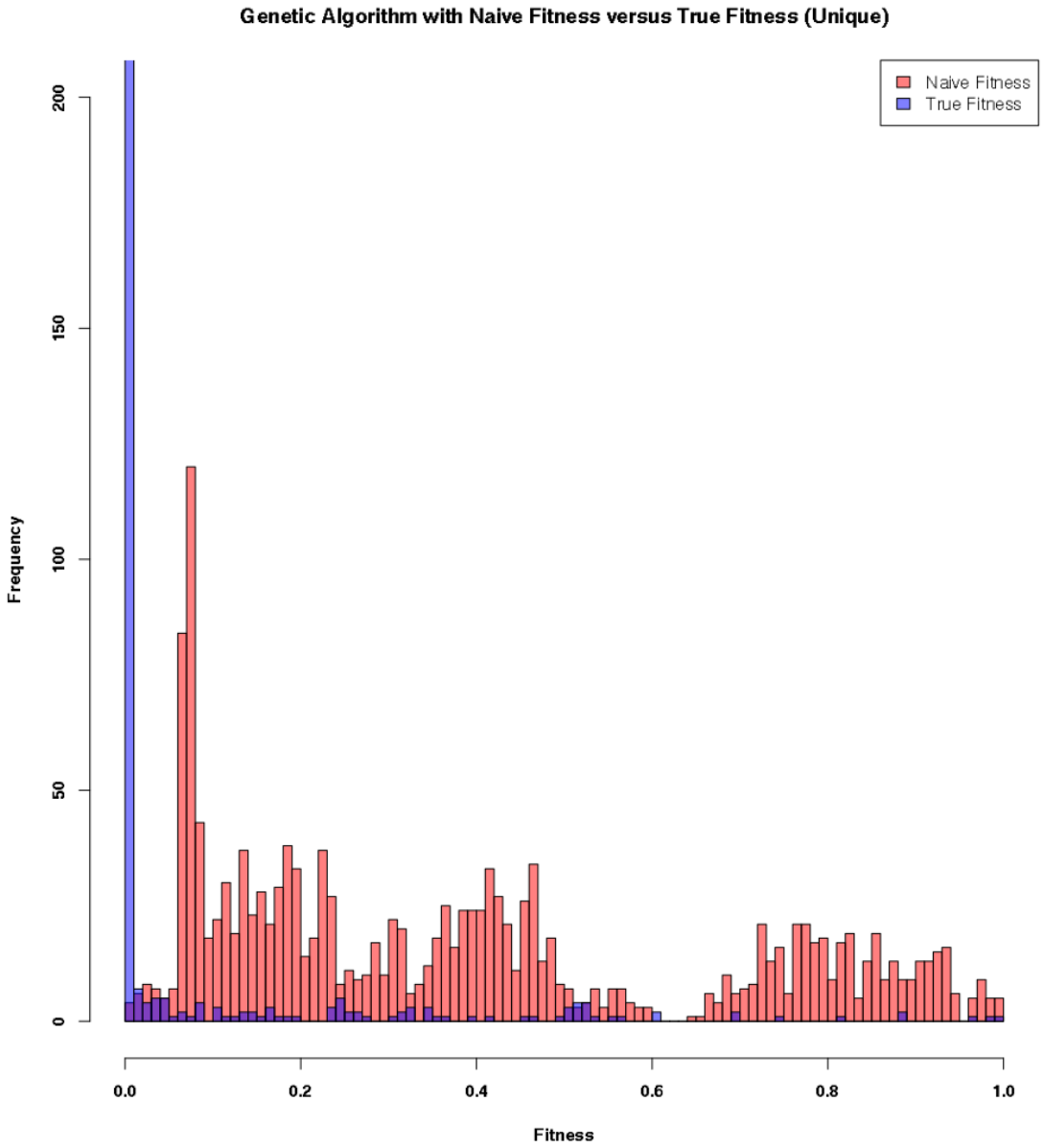

Figure 11 - Histogram of Unique Genetic Algorithm Results with Naïve Fitness

Comparing the performance against the true fitness function, we see a problem similar to that in the random search technique; the naïve fitness function is overrepresenting the true fitness, as shown by the large swathes of red in our diagram. The majority of input differences generated by the genetic algorithm were almost unusable with respect to the true fitness function, as shown by the blue spike at 0. This further cements the idea that the naïve fitness function is not a viable way of representing the true fitness, which in turn indicates that our genetic algorithm was selecting for the wrong properties.

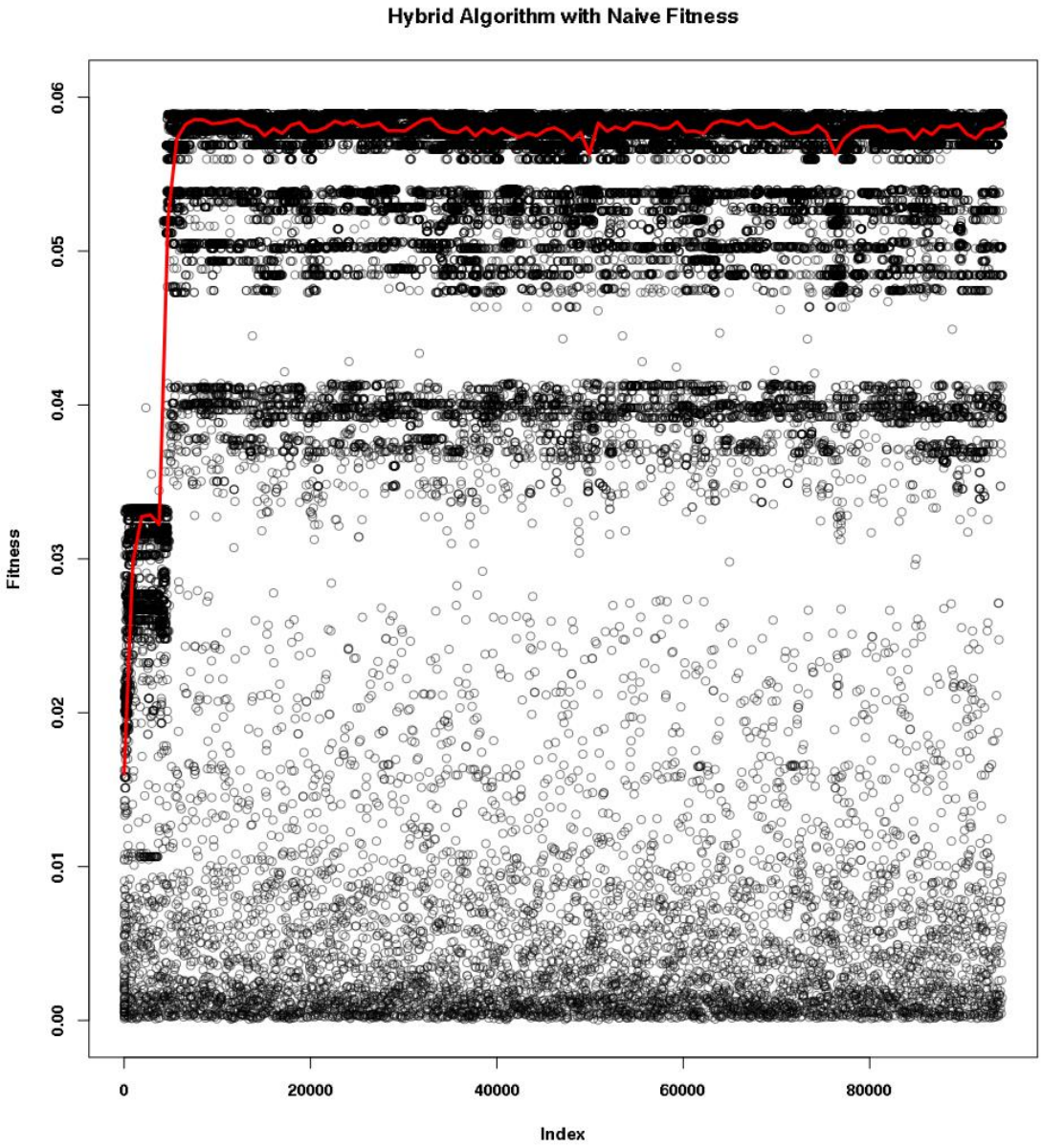

Figure 12 - Scatterplot of Hybrid Algorithm with Naïve Fitness

Unlike the genetic algorithm, the hybrid algorithm has far less banding, with only minor banding at the 0.06, 0.05, and 0.04 mark. It is visually quite similar to the random search, with a large clustering of points near the bottom of the plot. Due to the lesser amount of data, the trend is slightly easier to see. The initial pool is far less fit than that in the genetic algorithm, but quickly increases in fitness, and settles at an overall higher fitness than the genetic algorithm. Furthermore, the volatility of the trendline is much lower at this point, indicating it is much more stable than either the genetic algorithm or the random search. Overall, we see that the hybrid algorithm is able to produce a large amount of data, all with a very high fitness with respect to the naïve fitness function.

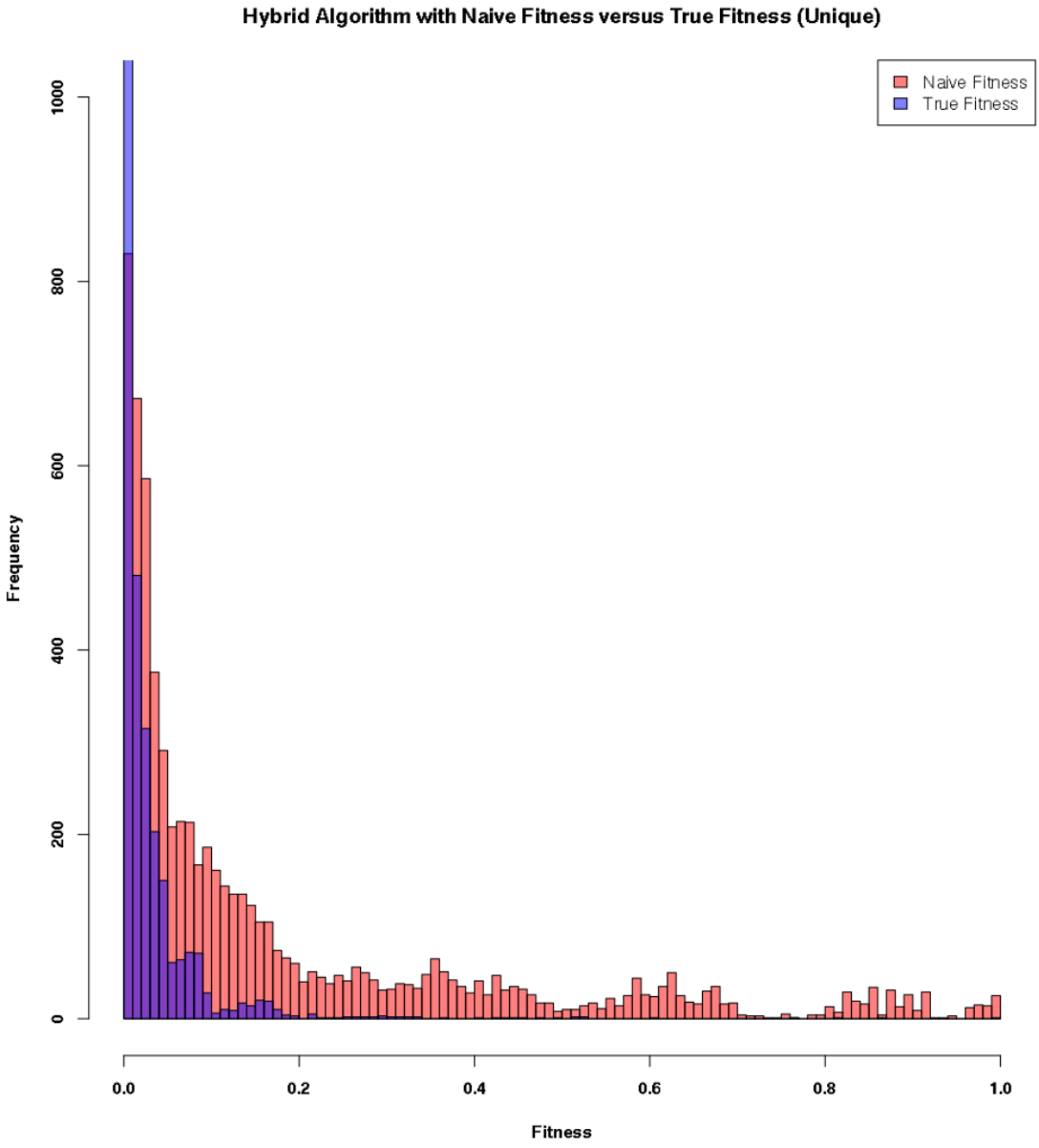

Figure 13 - Histogram of Unique Hybrid Algorithm Results with Naïve Fitness

Compared to the previous techniques, the fitness of input differences produced by the hybrid algorithm are much closer to the fitness' derived from the true fitness function. However, it is still not perfect, with a large spike close to zero indicating that a large proportion of generated values are not viable for producing collisions. Furthermore, although there are a large number of data points to the right of the diagram, there is no visible blue in this portion, indicating that most of the values generated were not able to produce actual collisions at any rate, further indicating that the naïve fitness function does not accurately represent the true fitness of an input difference.

4.1.3 Evaluation

Overall, we believe that the naïve fitness function, although computationally simple and space-efficient, is not a good representation of the true fitness for an input difference. Many of the input differences produced through our tests were of very low quality with respect to the true fitness, and were simply overrepresented by the naïve fitness function. As such, the genetic algorithm and hybrid algorithm selected for the wrong properties, and as such amplified these issues, causing the notable spikes at zero.

4.2 Probabilistic Fitness Function

4.2.1 Statistical Summary

Given in the tables below are a summary of the tests performed on the probabilistic fitness function. The second table gives the results from the true fitness function on the unique data from each test. Values are given to 6 decimal places where appropriate.

Table 9 - Statistical Summary for Probabilistic Fitness Function

	Results	**Unique Results**	**Mean**	**Standard Deviation**	**Coefficient of Variation**
Random	5322	5078	0.162716×10^{-8}	2.199505×10^{-8}	13.517420
Generated	77,846	378	2.885907×10^{-8}	4.233442×10^{-8}	1.466936
Hybrid	172,641	5435	1.167244×10^{-8}	7.471202×10^{-8}	6.400720

Table 10 - Statistical Summary for True Fitness of Probabilistic Fitness Function

	Unique Results	**Mean**	**Standard Deviation**	**Coefficient of Variation**
Random	5078	19.145160	72.116260	3.766813
Generated	378	42.485410	69.317800	1.631567
Hybrid	5435	14.720830	64.894980	4.408378

We notice a very similar pattern in this data that was also present with the naïve fitness function. Overall, the random search did worst, followed by the hybrid algorithm, and finally the genetic algorithm faring the best. However, there are some minor differences that should be highlighted.

Firstly, the average fitness of the input differences generated by the random search are almost an order smaller than those of the hybrid and genetic algorithm at 0.1627×10^{-8}, whereas in the naïve fitness function, the difference was fairly minor. Furthermore, the coefficient of variation is far higher at 13.5174, indicating a very large amount of spread in the data, in contrast to the fairly minor spread in the naïve fitness function. Similar to our previous results though, 5322 total results were generated by random search, with 5078 being unique, a duplication rate of only 4.6%. As with the naïve fitness function, the random search technique is highly volatile, and although is able to produce a fair number of input differences, produces those with very low fitness overall.

Interestingly, the genetic algorithm fared very poorly in this test with respect to the number of data points generated, with only 77,846 total input differences, and only 378 being unique, a duplication rate of 99.5%. Regardless, it attained the highest average fitness of 2.8860×10^{-8}, and the lowest spread of all at only 1.4669. We speculate that the reason this test attained so few unique values was due to it quickly converging to a local maximum, causing the gene pool to rapidly become stale.

Finally, the hybrid algorithm generated the most results of all in this test, at 172,641 total results, with 5435 being unique, a duplication rate of 96.8%. It had a moderately high average fitness at 1.1672×10^{-8} and spread of 1.6401. Much like the results from the naïve fitness function test, the hybrid algorithm appears to have taken on many of the positive qualities from the genetic algorithm, and negative qualities from random search, which overall impacted its performance.

The true fitness follows a similar trend to that of the probabilistic fitness function. One key difference however, is that the hybrid algorithm actually performs the worst out of all three, in contrast to all previous results. The genetic algorithm produced solutions of a very high fitness, which were on average able to produce 42 collisions over 10 seconds, and the lowest spread at 1.6315. This indicates that the probabilistic fitness function may indeed be a good representation of the true fitness function, however this is yet to be shown definitively.

4.2.2 *Visualization of Data*

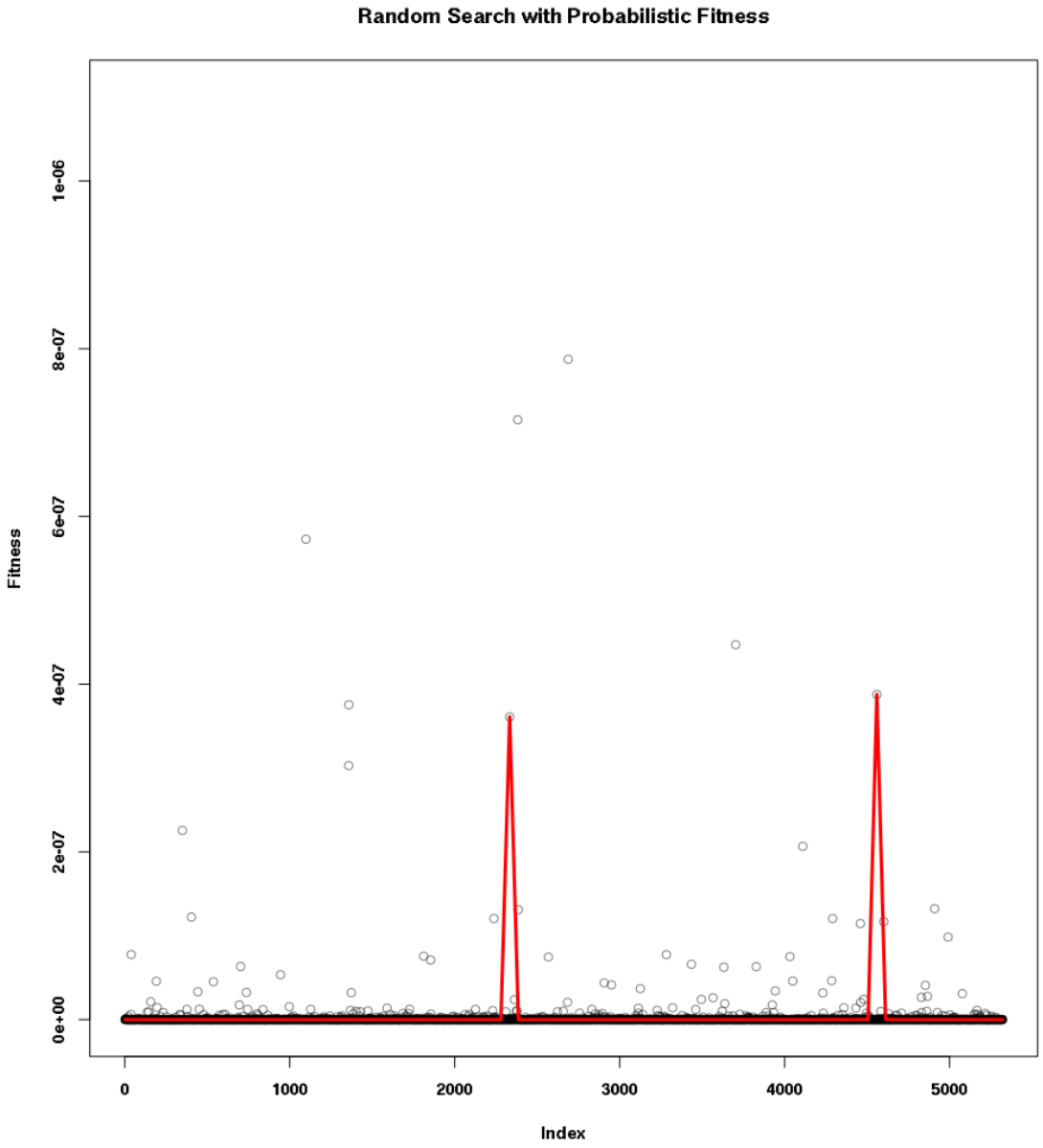

Figure 14 - Scatterplot of Random Search Results with Probabilistic Fitness

Unlike the previous iteration of this test, we see an extreme level of clustering of points close to the zero mark, with only few points straying outside of this band. We see that the trendline spikes twice at 2200 and 4500, which is likely due to a small cluster of points around these areas. Overall, we see that the random search technique fares very poorly at producing input differences with a high probabilistic fitness.

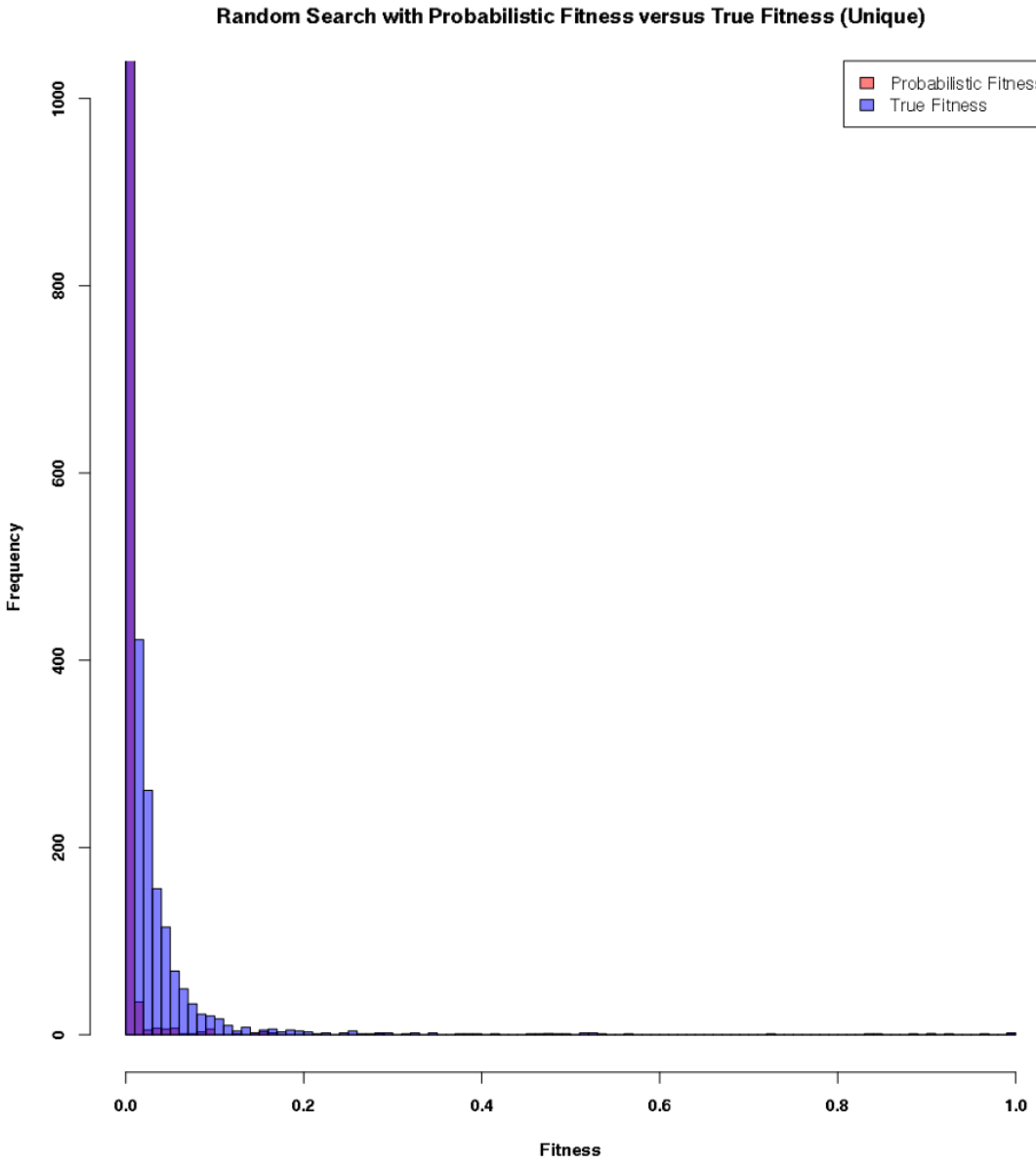

Figure 15 - Histogram of Unique Random Search Results with Probabilistic Fitness

In this graph, we see that the probabilistic fitness function is highly underrepresenting the true fitness, as demonstrated by the large swaths of blue. The majority of data points from the probabilistic fitness function lie around the zero mark, which is possibly due to the issues with precision discussed in the earlier chapter. However, the general shape of the two distributions is very similar.

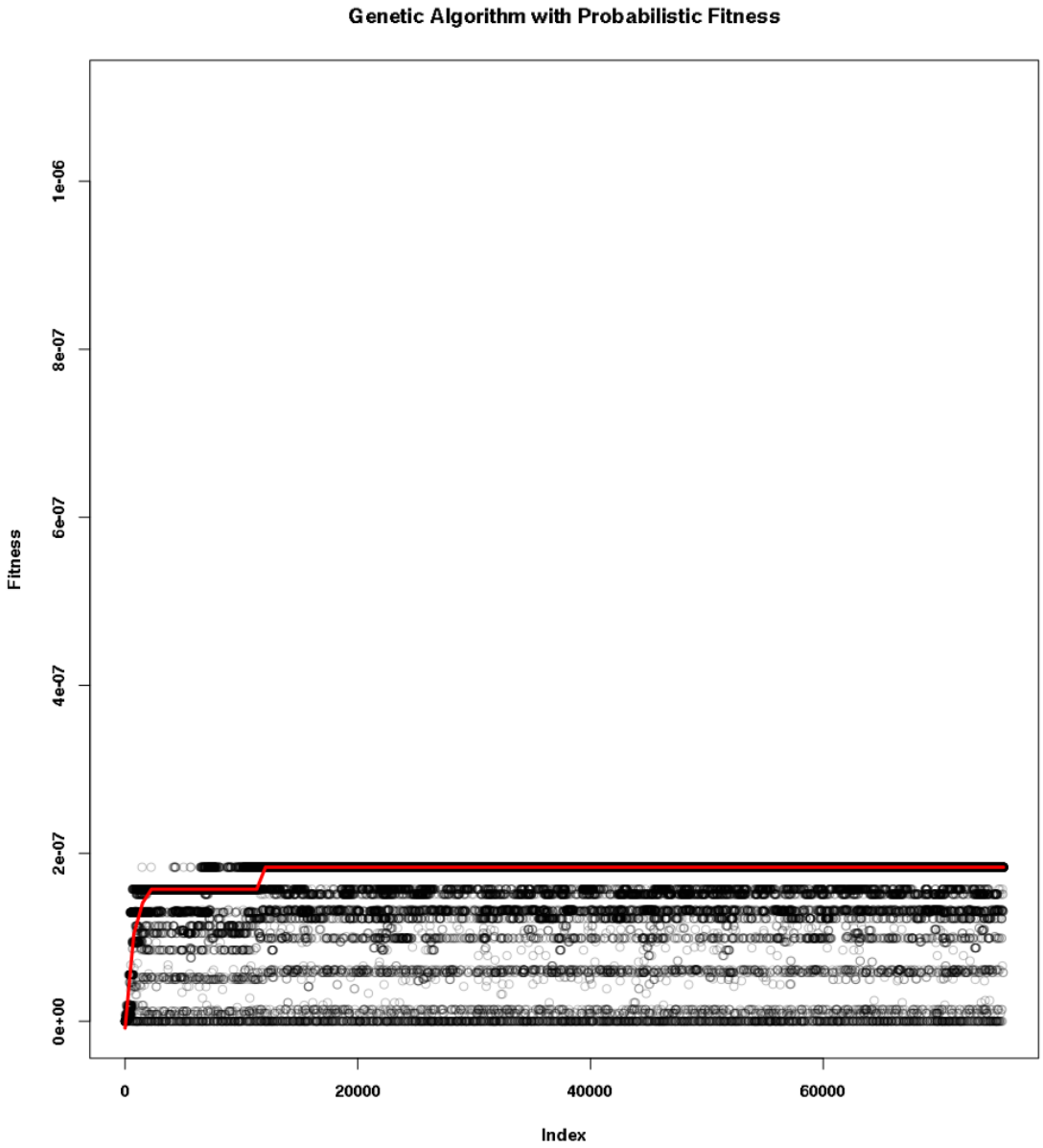

Figure 16 - Scatterplot of Genetic Algorithm Results with Probabilistic Fitness

Similar to the results from the naïve fitness function, we see a very large amount of banding, with a large quantity of data points present at the higher bands such as 2×10^{-7}. Again, we attribute this to the gene pool converging to a local maximum. The increase of the trendline is much more obvious in this test, where the gene pool starts at almost zero, and rapidly increases to close to the population mean. However, the maximum fitness generated by this technique is very low in comparison to the other techniques.

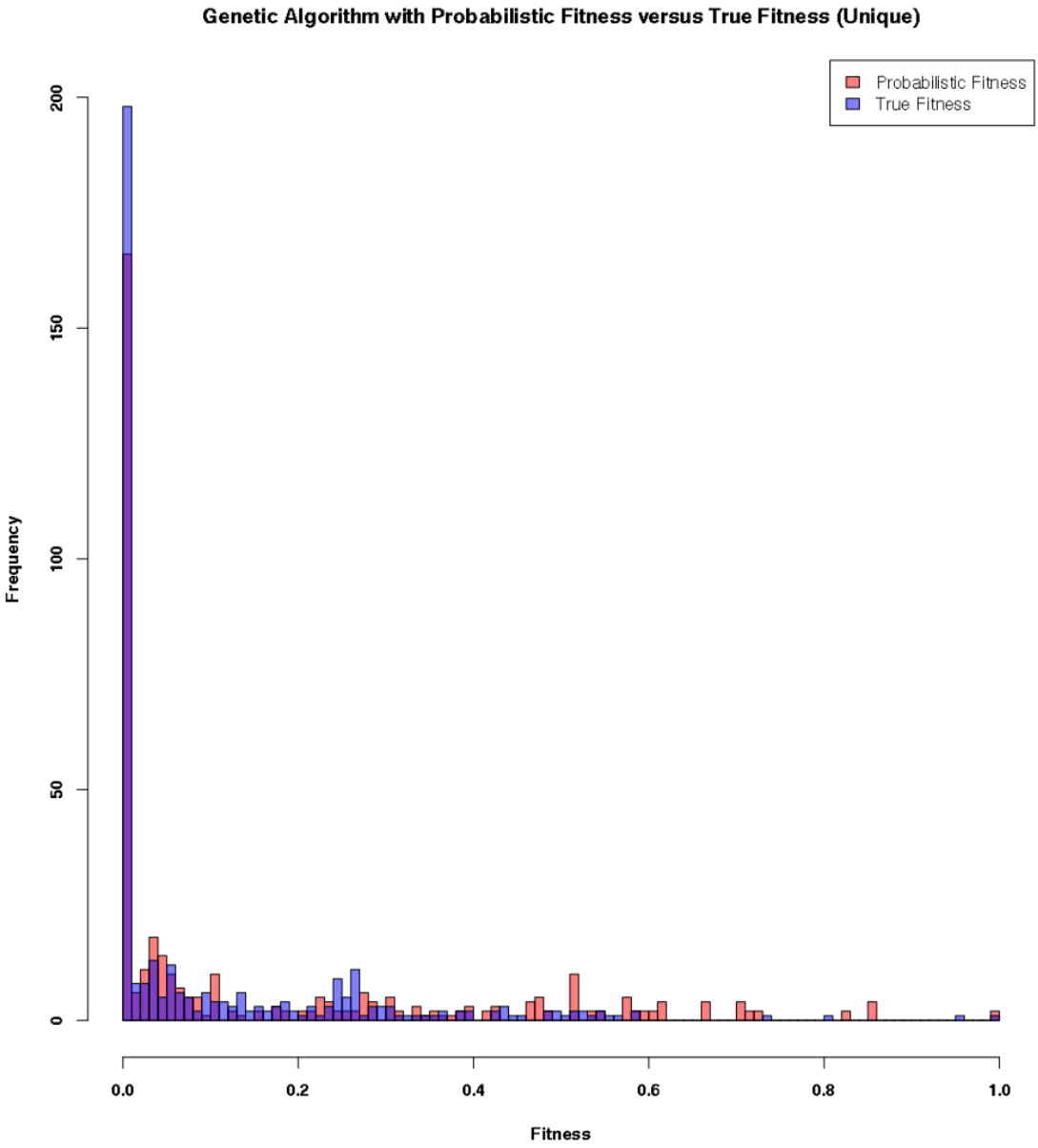

Figure 17 - Histogram of Unique Genetic Algorithm Results with Probabilistic Fitness

Interestingly, we see that the probabilistic fitness function represents the true fitness fairly well in this histogram. There is very little difference between the two plots as indicated by the pure red and blue blocks, in contrast to all other comparisons we have seen thus far. However, this may simply be due to the fact that there are only slightly over 350 distinct data points in this graph, which could cause some bias.

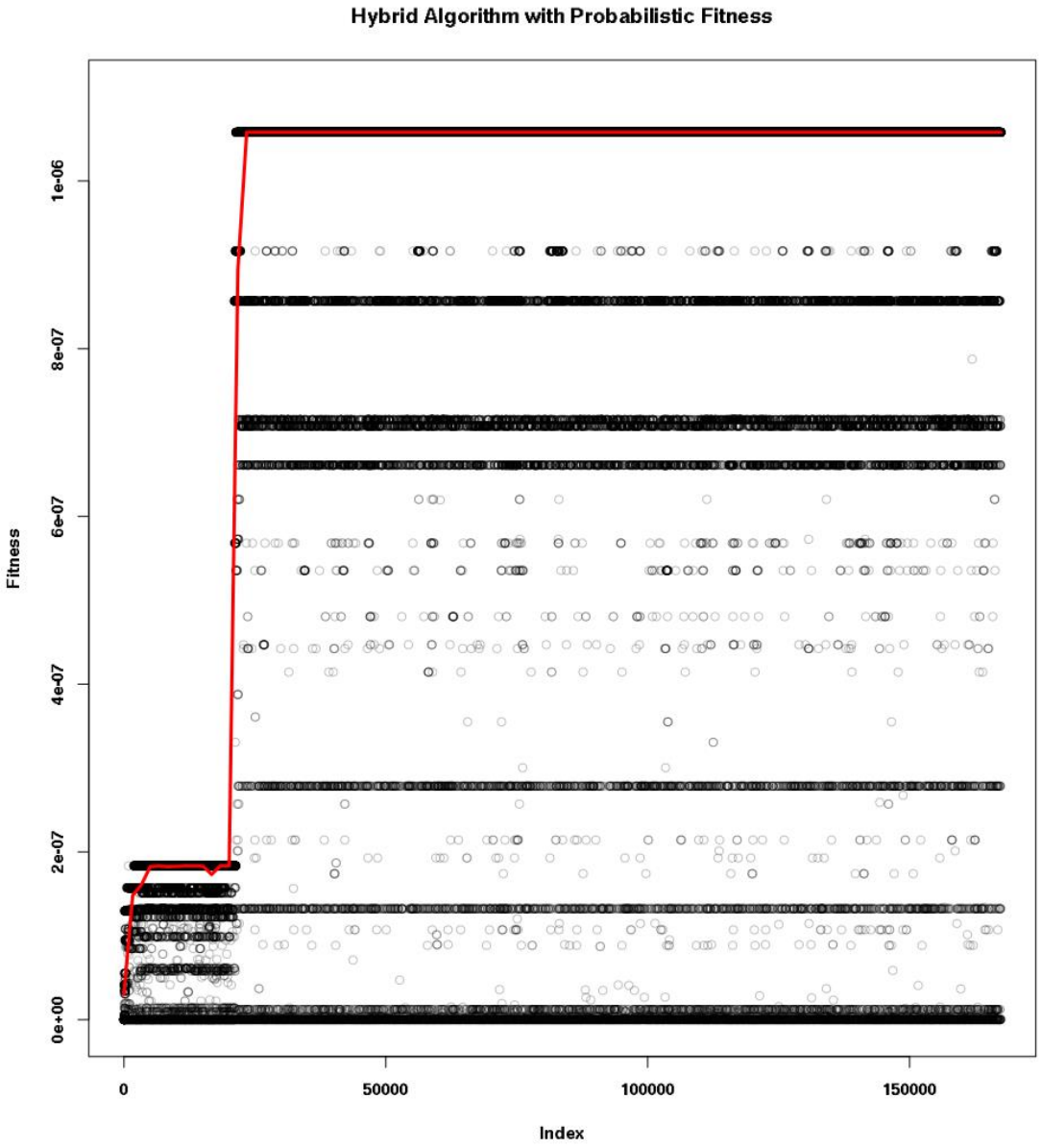

Figure 18 - Scatterplot of Hybrid Algorithm Results with Probabilistic Fitness

Like our previous tests, the hybrid algorithm begins with a fairly poor gene pool, and is able to quickly rise at the 25,000 mark, plateauing at a fitness of over 1×10^{-6}, the highest of all we have seen in this round of testing. Much like the genetic algorithm, we see a large amount of banding spread across the graph, indicating stability, however there is still a large amount of randomly generated points occurring.

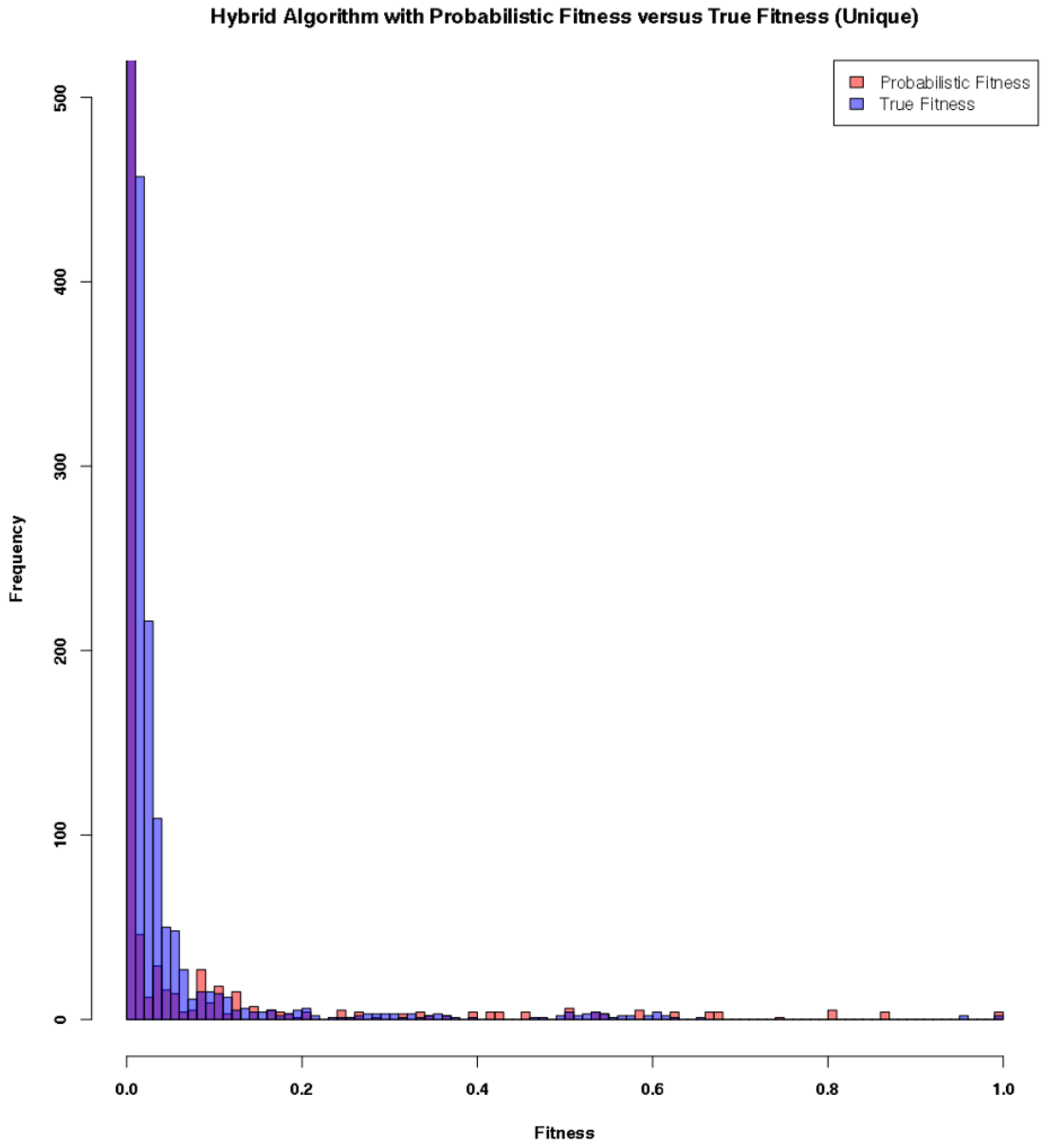

Figure 19 - Histogram of Hybrid Algorithm Results with Probabilistic Fitness

Similar to the random search technique, we see that the probabilistic fitness is again underrepresenting the true fitness of the generated input differences, specifically with a bias towards lower fitness', as evidenced by the large swatch of blue at the lower end of the graph. However, the general shape of the distributions is again quite similar, indicating that the probabilistic fitness function may indeed represent the true fitness fairly well.

4.2.3 Evaluation

We believe that the probabilistic fitness function does represent the true fitness fairly well, although there are some issues with underrepresentation present. We believe that this issue is caused by floating point imprecision, as many trails may hold with probabilities as small as 2^{-50}. If the probability of a trail is sufficiently small, then it will simply be rounded to zero by the hardware and will not actually contribute anything to the overall fitness, which is not desirable behaviour. However, the since the general shape of the distributions is very similar, we believe that it does indeed represent the true fitness well, even with this discrepancy.

4.3 Summary

From our testing, we have derived several important results. Firstly, we have shown that the naïve fitness function does not adequately represent the true fitness function, due to it grossly overrepresenting the true fitness of an input difference. The only benefit to using this fitness function is the low overhead it imposes, as it is simply the ratio of zero trails to all trails in the propagation tree. Furthermore, since it selects for the wrong property, it causes the genetic and hybrid algorithms to select for the wrong input differences, which in turn decreases their ability to produce collisions.

The probabilistic fitness function represents the true fitness fairly well, but unfortunately underrepresents it in most cases. We believe that this may be caused by floating point errors, since many of the values used will be of the order 2^{-50}. Since the smallest non-zero double-precision value is 2^{-52}, and number smaller than this will simply be rounded to zero. As such, if we have many trails in our propagation tree with probabilities smaller than this minimum value, they will not contribute anything to the overall fitness, which is not desirable behaviour. This in turn leads to the underrepresentation problem. Overall, the probabilistic fitness function shares the same type of distribution as the true fitness function, even with its underrepresentation.

With respect to the probabilistic fitness function, the random search technique was very poor at producing good input differences, but was able to consistently produce several thousand unique values over a 24-hour period. The spread of these values was very high, with most of the values tending to very low fitness'. However, with respect to the true fitness, many performed very well, being able to produce nearly 20 collisions over a 10 second period. The genetic algorithm was decidedly better however, with the highest average true fitness of

all techniques, and the lowest variation. Unfortunately, the hybrid algorithm underperformed severely, being beaten by pure random search, indicating that it is not particularly suited to this problem.

Overall, we believe that these results indicate that optimization algorithms can indeed be applied in the search for input differences for differential cryptanalysis, with the most appropriate technique being a pure genetic algorithm, and fitness function being the probabilistic fitness function. However, there are issues present in both of these which impact their overall performance, and mitigations will be addressed in our discussion.

5. Discussion

As demonstrated in our testing, we have given a possible fitness function which approximates the true fitness of an input difference. With this, it is possible to apply optimization algorithms to help generate and improve such values, which in turn helps answer the question of, how does one choose a good input difference? In this section, we discuss our results, their relation to current research, and their overall impact on the field.

5.1 Current State of Differential Cryptanalysis

The field of differential cryptanalysis is quite mature; it has been used by experts in the field publicly for almost 30 years, and has been known about privately for almost 45 years. It has been used successfully in the cryptanalysis of many cryptographic components, most notably the DES and FEAL cipher, as well as MD5 and SHA-1. It has been shown time and time again that it is a very powerful framework, and when used efficiently, can readily break many different primitives. However, the public literature on this field is skewed heavily towards those already in the field. Of all the literature considered for this project, none of it was suitable to a beginner cryptanalyst, a problem we intend to solve with this project. Furthermore, in most literature considered, no explanation to how the input difference Δ_0 was found, which impacts negatively on the field, as it makes research more difficult to replicate, and means that other cryptanalysts cannot learn from the processes of others.

5.2 Applications of our Work

We believe that our work has made a significant contribution to the field of hash function cryptanalysis. As outlined in our background, the literature on this field is very sparse, with the only public source of literature effectively being master's and PhD theses and conference proceedings. Since this report covers several key components of differential cryptanalysis in great detail, including their purpose, example implementations, and conditions, we believe that this report may serve as a suitable introductory text to the field of differential cryptanalysis. This in turn helps to bridge the gap in literature discussed earlier.

Furthermore, we believe that this project has highlighted a potential area of research within the field of differential cryptanalysis, which is computational methods for searching for choices of Δ_0. As discussed earlier, due to the lack of appropriate literature, it is not

apparent what techniques should be applied to find suitable choices for Δ_0, and in many cases is assumed to be found by hand, meticulously, by teams of researchers. As such, this project offers a novel method to derive this value, and furthermore opens several new areas of research within the field, such as effective conditions, memoization techniques, and different optimization techniques. We also believe that the techniques highlighted in this report may be used in tandem with existing techniques to help make searching for such a value easier, and help in breaking cryptographic hash functions like SHA-2.

5.3 Limitations

While the techniques discussed in this report may be readily extended to most cryptographic hash function, there are several important limitations to our project which impact the application to more complex functions.

5.3.1 Propagation Tree Size

Many of the choices we made in our design were with the intention of decreasing the complexity of the propagation tree at each step. For example, we selected only those input differences which had at least 2/8 message differences being precisely zero, and furthermore left an entire section of the input difference as zero. Without such optimizations, the size of the propagation tree would increase, causing an exponential increase in the amount of time spent trying to propagate a value. In the worst possible case, given a threshold probability of 2^{-n}, we would have at most 2^n branches to check. In MAW32, there are 7 non-linear operations in each round, meaning that for each round there will be 2^{7n} branches to be considered in total.

There are many possibilities for solving this issue however. One such solution is the notion of sufficient conditions and contradiction searching. This involves computing a set of necessary conditions which can increase the probability of a particular branch, typically by placing some amount of restriction on what the input may look like. Furthermore, such conditions may eliminate entire branches, providing a substantial reduction to the overall size of the propagation tree. This was not considered for this project however, due to the lack of public knowledge and difficulty of implementation. Another potential solution is to use conditional probabilities for each branch. This notion is similar to that of searching for necessary conditions. In this report, we have assumed that the probability of a particular branch being taken is independent of another branch before it, however this almost never the case. As such, being in one part of the propagation tree may drastically change the

probability of another branch being taken. As before, this was not considered for this project, due to the overall complexity and memory requirements of such a system.

5.3.2 *Effective Memoization*

Due to the amount of time it takes to propagate an input difference through a non-linear component, an early optimization we made was to memorize the results of propagations, which effectively allowed us to reduce the problem of propagation to a table lookup. In MAW32, this was quite feasible. For the key mix and addition operations, there are only 2^8 and 2^{16} possible inputs respectively. To propagate all possible input differences would require 2^{16} and 2^{32} operations in total, which is entirely feasible. For the Maj component however, this was not possible. In this component, there are 2^{24} possible inputs, which on a single thread takes between 0.1s and 1s to evaluate. However, since there are 2^{24} possible input differences, to exhaustively compute all propagations would take approximately two months to complete. To combat this, a trade-off was made; rather than precompute the propagations exhaustively, we would compute all possible input differences, but sample the input space with 2^{16} possible inputs. Due to the size of the sample space, the probability we derive from this should closely match the true probability.

The technique of aggressive memoization has several drawbacks however. As discussed before, since the probabilities computed are independent of the context of the propagation, the results we get are not the true probability of each branch. Furthermore, this technique is not extensible. In SHA-256, the simplest component has an input space of 2^{64}, which is itself intractable to exhaustively compute. Furthermore, with 2^{64} possible input differences, even by sampling the input space, we would not be able to memorize the propagation for the simplest SHA-256 component ahead of time. Finally, the memory footprint for such a table is immense; it would not be storable in RAM for most computers. As such, memoization cannot be done exhaustively for most hash functions; we recommend it only for this project.

5.3.3 *Hardware Limitations*

Due to the limitations of an Honours project, all testing was done on a single machine, assigned by the Cyber Security Department for the purposes of this project. The CPU used was an Intel i7-4770, which had 4 physical cores and hyperthreading, and had a base frequency of 3.40GHz. For memory, it had 8GB of available RAM, and 1GB of swap. As such, it was not possible to perform as much testing as we would have preferred. In our practical tests, we limited ourselves to using four worker threads and one controller thread, which

ensured the machine itself remained responsive, allowing it to still be used while tests were running. In more in-depth research, we believe that parallelization and distribution is very important. Rather than using four worker threads, it is much more effective to use several thousand CUDA cores on a graphics card, or instead perform the work across several computers. To this end, an important limitation was the way the propagation code was written; it is not readily parallelizable. As such, it is not currently possible to write a small program to distribute the work over several computers or graphics devices, which is itself a bottleneck.

5.3.4 Floating Point Precision

As discussed in our results, while the probabilistic fitness function did represent the true fitness of an input difference fairly well, it unfortunately underrepresented the true fitness by a fair amount. This is likely due to issues with floating point precision. As any values smaller than 2^{-52} are rounded to zero, any differential trail that holds with a probability smaller than this will effectively be rounded to zero by the underlying hardware. This is a major problem, as it can cause our algorithms to discard potentially good input differences due to them having a large number of differential trails which all hold with very small probabilities. To solve this, there are two possible approaches.

The first approach is to switch from using floating point numbers to fixed point numbers. A fixed-point number is one in which the amount of precision is set to be a specific value, for instance 12 decimal places. Although the memory footprint of these is much higher than that of a traditional floating-point number, it does not sacrifice precision like a floating-point number does. Furthermore, as the amount of precision can be specified by the programmer, it allows for arbitrary precision at the cost of memory. In turn, this means that our lower probability trails will not be disregarded like they would with a traditional floating-point number, which helps to solve the problem of underrepresentation.

The second approach is integer approximation. Rather than storing the actual fitness as a floating-point number, we store its value $\log_2$. For instance, the fitness of a trail of probability 2^{-53} would be stored precisely as -53, whereas it would be discarded by a double-precision floating point number. The issue with this however is adding together two fitness'. With a floating-point number, this is trivial. However, to add two fitness' stored as $\log_2$ we effectively must compute $\log_2(2^a + 2^b)$. However, we cannot directly compute this, as 2^a or 2^b may be smaller than 2^{-52}, in which case they would immediately be rounded to zero. As such, we must find appropriate ways to approximate this value without loss of

precision. However, if such an approximation can be found, then this is a viable method of storing the fitness without causing underrepresentation.

6. Conclusion

The initial aim of this project was to cryptanalyze the SHA-2 family of cryptographic hash functions, in order to determine whether or not any exploitable weaknesses were present. However, due to the lack of appropriate literature, specifically with regards to calculating the value Δ_0, performing this task was not clear. As such, this project aimed ot find appropriate ways of computing Δ_0 in order to improve the public knowledge in the field of hash function cryptanalysis. To this end, we created our very own SHA-like hash function to cryptanalyze. This function shared an almost identical internal structure to the SHA-2 family, meaning that experiments performed on it could be readily extrapolated to SHA-2.

6.1 Results

As demonstrated in our results, we were able to effectively generate potential values for Δ_0 which could in turn be used to find collisions in MAW32 with a high degree of certainty. We also investigated potential fitness functions, and determined which was the most appropriate in solving our problem. With this, we were able to test several algorithms for computationally searching for choices for Δ_0, which we have shown to be better than pure random search. These choices for Δ_0 were even able to find collisions in MAW32 consistently, indicating that our techniques were indeed applicable and not simply random chance.

Our results are not limited to only MAW32 however. This hash function shares a similar internal structure to the SHA-2 family, and as such, we have conjectured that the techniques explored in this project may be applied to SHA-2 as well. However, practically demonstrating this requires a very large amount of time and processing power, and as such cannot be done within the time frame of an Honours project. However, we do not believe that this is limited to only the SHA-2 family. Rather, we conjecture that this may be applied at the very least to any Davies-Meyer and Merkle-Damgård construction hash function, which are still in common use. As such, we do not believe that this technique will readily extend to more complicated hash function families such as SHA-3.

6.2 Contribution

As described in the motivation for this project, a common issue in the field of hash function cryptanalysis is the lack of appropriate literature; the majority of the available literature are theses and conference proceedings, which are highly technical and not a good

source for beginners to learn from. Furthermore, a common trend seen in these words is a lack of in-depth explanation for several key topics, one of which is the selection of the value Δ_0. In this project, we have bridged this gap by explicitly describing how such a value may be chosen, what conditions to impose, and several possible algorithms for finding it. Furthermore, we have also described a process in which any existing values for Δ_0 may be improved with respect to a fitness function.

We believe that this project will impact the field in two important ways. Firstly, this report contributes to the pool of public knowledge, primarily as an introductory text to the field of hash function cryptanalysis, which as discussed earlier is severely lacking in such literature. This report provides an in-depth analysis and explanation of several key terms used within the field, provides more rigorous definitions, and provides possible solutions and implementations. As such, we believe that this report may be useful to aspiring cryptanalysis as a hands-on introduction to the field.

Furthermore, we believe this project will impact the field itself by providing a new technique to attack cryptographic hash functions. We have demonstrated that optimization algorithms may be applied in the search for good choices of Δ_0, which are key to finding practical collisions in cryptographic hash functions. As such, we believe that the techniques explored in this report may be used to supplement other techniques used in the field by current researchers in an effort to find more optimal values of Δ_0, allowing for more substantial leaps to be made in breaking cryptographic hash functions.

6.3 Future Work

This project has opened several avenues for future research. The optimization algorithms used in this project are very few, and are in no way optimal in solving this problem. As such, other optimization algorithms should be tested to determine their efficiency and ability to generate viable input differences, as well as other possible fitness functions and propagation techniques. To facilitate this, it is necessary to improve the performance of the propagation function. One technique mentioned in most literature suggests searching for sufficient conditions for a differential trail to hold, which in turn allows us to eliminate particular branches from the propagation tree, as their conditional probability shrinks to zero. One avenue for future research is computationally searching for these conditions, either analytically or approximately. These in turn will allow propagation to occur at a substantially higher rate.

Another issue faced is the structure of the propagation function itself; by its definition, it cannot be readily parallelized. However, being able to do so would immediately be beneficial, as it would allow us to both parallelize and distribute the workload, allowing us to utilize a plethora of devices, rather than using a single processor. Projects of this type have been used in the past, for example the MD5CRK project, a distributed project for finding MD5 collisions, and GIMPS, a distributed project for calculating Mersenne primes. Such restructuring would also allow for the parallelization over graphics devices via CUDA. Since operations such as propagation are very simple in nature, they may be readily pushed onto a massively parallel device such as a graphics card or even an ASIC, an Application Specific Integrated Circuit. This in turn would facilitate other work, including better methods for propagation, finding sufficient conditions for propagations, and calculating conditional probabilities for branches. All of these in turn decrease the overall time spent on propagations, and furthermore improve the performance of the fitness function, leading to better results.

Finally, the most important issue is the extension to SHA-2. As it stands, the code used for the MAW32 propagations may be slightly modified to cryptanalyze the SHA-2 family, but due to time and computational constraints, could not be completed in the duration of this project. As such, we leave as future work in this area, extending the existing framework to SHA-2 and running the necessary tests in order to try to find potential candidates for Δ_0 for the cryptanalysis of the SHA-2 family.

References

1. Rivest, R. *The MD4 Message Digest Algorithm*. 1990; Available from: https://tools.ietf.org/html/rfc1186.
2. den Boer, B. and A. Bosselaers. *An Attack on the Last Two Rounds of MD4*. 1992. Berlin, Heidelberg: Springer Berlin Heidelberg.
3. Dobbertin, H. *Cryptanalysis of MD4*. 1996. Berlin, Heidelberg: Springer Berlin Heidelberg.
4. Rivest, R., *The MD5 Message-Digest Algorithm*. 1992.
5. Wang, X. and H. Yu. *How to Break MD5 and Other Hash Functions*. 2005. Berlin, Heidelberg: Springer Berlin Heidelberg.
6. Lenstra, A. and B. de Weger. *On the Possibility of Constructing Meaningful Hash Collisions for Public Keys*. 2005. Berlin, Heidelberg: Springer Berlin Heidelberg.
7. Turner, S. and L. Chen. *MD4 to Historic Status*. 2011; Available from: https://tools.ietf.org/html/rfc6150.
8. Turner, S. and L. Chen. *Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms*. 2011; Available from: https://tools.ietf.org/html/rfc6151.
9. Technology, N.I.o.S.a., *Secure Hash Standard (SHS)*, in *FIPS 180*. 1993.
10. Technology, N.I.o.S.a., *Secure Hash Standard (SHS)*, in *FIPS 180-1*. 1995.
11. Stevens, M., P. Karpman, and T. Peyrin. *Freestart collision for full SHA-1*. 2015.
12. Stevens, M., et al. *The First Collision for Full SHA-1*. 2017. Cham: Springer International Publishing.
13. Technology, N.I.o.S.a., *Secure Hash Standard (SHS)*, in *FIPS 180-2*. 2002.
14. Biham, E. and A. Shamir, *Differential Cryptanalysis of the Data Encryption Standard*. 1993: Springer-Verlag New York.
15. Schläffer, M. and E. Oswald. *Searching for Differential Paths in MD4*. 2006. Berlin, Heidelberg: Springer Berlin Heidelberg.
16. Merkle, R., *Secrecy, Authentication, and Public Key Systems*, in *Department of Electrical Engineering*. 1979, Stanford University.
17. Damgård, I.B. *A Design Principle for Hash Functions*. 1990. New York, NY: Springer New York.
18. Matsui, M. and A. Yamagishi. *A New Method for Known Plaintext Attack of FEAL Cipher*. 1993. Berlin, Heidelberg: Springer Berlin Heidelberg.
19. Coppersmith, D. *The Data Encryption Standard (DES) and its strength against attacks*. 1994.
20. De Cannière, C. and C. Rechberger. *Finding SHA-1 Characteristics: General Results and Applications*. 2006. Berlin, Heidelberg: Springer Berlin Heidelberg.
21. Wang, X., et al. *Cryptanalysis of the Hash Functions MD4 and RIPEMD*. 2005. Berlin, Heidelberg: Springer Berlin Heidelberg.
22. Lamberger, M. and F. Mendel, *Higher-Order Differential Attack on Reduced SHA-256*. 2011.
23. Lai, X., *Higher Order Derivatives and Differential Cryptanalysis*, in *Communications and Cryptography: Two Sides of One Tapestry*, R.E. Blahut, et al., Editors. 1994, Springer US: Boston, MA. p. 227-233.
24. Mendel, F., T. Nad, and M. Schläffer. *Finding SHA-2 Characteristics: Searching through a Minefield of Contradictions*. 2011. Berlin, Heidelberg: Springer Berlin Heidelberg.

Appendices

A: Source Code

All source code is available freely on GitHub at the following link: https://github.com/mitchgrout/COMP520