**School of Computer Science and Engineering**

**Faculty of Engineering**

**The University of New South Wales**

# Essay on Cryptographic Hashes

# Contents

# List of Figures

# List of Tables

# 1 Introduction

Cryptographic primitives are building blocks with which larger, more complex cryptographic protocols and systems are built. One such cryptographic primitive is the cryptographic hash function, a one-way function that transforms an input of arbitrary size into a fixed length output while maintaining particular properties that will be discussed in this paper. Ultimately, the one-way nature of cryptographic hash functions allows creating systems with a large work ratio, where legitimate users can perform actions efficiently while the amount of work for an attacker to exploit the system is impractical.

In this paper, we will provide an understanding of cryptographic hash functions. We will identify the required properties that define a cryptographic hash function, describe various methods used to construct them and then provide examples of how they are used.

## 1.1 Document structure

The document consists of the following chapters:

- Chapter 1 outlines the content of this paper.
- Chapter 2 introduces the properties of cryptographic hash functions, which make up a subset of general hash functions.
- Chapter 3 describes three common constructions used in cryptographic hash functions: the Merkle-Damgård construction, HAIFA construction and Sponge construction.
- Chapter 4 explores some applications of cryptographic hash functions, namely password hashing, proof of work and digital signatures.
- Chapter 5 summarises this paper.

# 2 The Concept of Cryptographic Hashing

In this chapter, we will first explore the wider concept of hashing, before describing cryptographic hashing and the properties that differentiate a cryptographic hash from a non-cryptographic hash. We will then explore, from a mathematical perspective, the uses of cryptographic hashing in security and the various ways that it can be exploited.

## 2.1 Hashing

A hash function takes in a value from a large or infinitely large set of input values (keys) and maps it to a smaller set of output values [1]. Speaking in terms of length, it takes an input of arbitrary length and produces an output of fixed, shorter length. A hash function must also be deterministic, in that applications of the function on the same input will produce the same output every time. Considering the applications of hash functions, such as in hash maps and file integrity checking using the cyclic redundancy check, we find that it is highly desirable for hash functions to be computationally efficient. The following is a simplistic example of a hash function with an infinitely large set of input values (the real numbers) and the set of output values being the integers $[0, 9]$:

$$H(x) = \lfloor 8x \ (mod\ 10) \rfloor$$

A collision is where a hash function maps two different inputs to the same output, which is inevitable when the size of the set of keys is greater than the set of output values. Uniformity occurs when the inputs are mapped as evenly as possible across the possible outputs [2]. For the $n$ possible output values, each input has an equal chance to be mapped to each output:

$$P(H(x) = i) = \frac{1}{n}, \text{ for all input values } x \text{ and output values } i.$$

So, for $m$ input values, it is expected that each output value has $m \times \frac{1}{n} = \frac{m}{n}$ values mapped to it.

A hash function or hashing technique may be described as perfect if it produces no collisions [3]. A perfect hash function can theoretically be generated when the set of keys is known in advance or does not change. Perfect hashing techniques often involve the use of multiple tables.

## 2.2 Cryptographic hashing

Cryptographic hash functions are a subset of hash functions that are suitable for use in cryptography. Although there are a wide range of uses of hashing in cryptography, they all make use of the following concepts described in this chapter.

In computing security, the 'CIA Triad' consists of confidentiality, integrity and availability [4]. Confidentiality involves protecting information from being disclosed to unauthorised parties. Integrity involves making a piece of information tamper resistant – i.e. making it unable to be modified by unauthorised parties. Availability involves ensuring the authorised parties are able to access this information. Cryptographic hashing is mostly used in situations where integrity is required. The applications of hashing will be explored further in Chapter 0.

### 2.2.1 Hashing vs Encryption

Encryption is a two-way function, meaning that data can be encrypted and decrypted. This is useful in situations requiring confidentiality and/or availability [5]. Hashing on the other hand is a one-way function. A well-designed cryptographic hash makes it practically infeasible to take a hash (the output) and determine what the input was.

### 2.2.2 Terminology

We will quickly clarify the terminology that is going to be used in the following sections and chapters. The input is referred to a message. In real world applications, this may be a string or a stream of bytes representing a file. The output is referred to as the hash or a message digest. The hash function refers to the actual function itself.

## 2.3 Properties of a cryptographic hash function

One of the overall aims of cryptographic hashing is to create a large work ratio. An authorised user can perform his/her actions with minimal effort, while the effort required for an attacker is many orders of magnitude greater.

An example of this is password hashing, which is described in greater detail in section 4.2. A legitimate user can send his/her password to the server, where it is hashed and then compared to the stored hashed copy. Access is granted if the two hashes match. But suppose an attacker acquires the hash and wishes to determine what the original password was. The effort required to work backwards to achieve this makes this infeasible.

The following properties describe this ‘one way’ nature of robust cryptographic hash functions, where it is easy for a legitimate user to hash an input to produce a desired output but nearly impossible for an attacker to take an output and determine what the input was.

### 2.3.1 Avalanche effect

Although it is essential for a cryptographic hash functions to be deterministic, it must also be seemingly random, so that given an output, an adversary will not be able to predict what the input was [6]. With cryptographic hash functions that demonstrate the avalanche effect, a change of a single bit will cause each of the $n$ output bits to change with a probability of 50%. So, on average, $n * \frac{1}{2} = \frac{n}{2}$ or half of all the output bits will change, if one input bit is changed. An example of this is below.

*Table 2.1 A demonstration of the avalanche effect*

| Input | SHA256 Digest |
|---|---|
| wire | 9B2ABFC29CC47494C87171177C2AF369FFF9F067FD768F6F58C5D83E5C658507 |
| fire | DC9F28B12DD1818EE42FFC92ECB940386214598837348D30D3C6C0B7B57E34C9 |

### 2.3.2 Bits of security

Bits of security is often used to express the level of security offered by a cryptographic hash function. A hash function with $n$ bits of security has $2^n$ possible outputs [7]. Taking the SHA-256 cryptographic hash function as an example, this function produces a digest that is 256 bits long. There are $2^{256}$ such binary strings and hence 256 bits of security. Cryptographic hashes are expressed as a string of hexadecimal characters (each character is 4bits long), so a SHA-256 hash is expressed as a string of 64 hexadecimal characters.

There are various exploits that aim to obtain an original value, given a digest from a cryptographic hash function. Strong cryptographic hash functions are resistant to these exploits, and we will describe these properties in the sections to follow.

### 2.3.3 Pre-image resistance

For a cryptographic hash functions to have pre-image resistance, given the digest $d$, it needs to be difficult to find an input $m$ such that $H(m) = d$. In other words, it needs to be difficult to work backwards from the digest to figure out the original input [8].

Unless a hash function is compromised, the only way to obtain a pre-image is to continually test inputs in a brute-force manner until a message is found so that the output of the hash function matches the given digest $d$. Given an $n$ bit hash function, there are $2^n$ different possible digests. Alternatively, we can express this as the time complexity of finding a pre-image to a given digest as $O(2^n)$.

The worst-case scenario is needing to try $2^n - 1$ inputs before finding an output that matches. The best-case scenario would be producing a matching digest on first try, though the probability of this happening is $\frac{1}{2^n}$, which is miniscule. On average, $\frac{2^n}{2} = 2^{n-1}$ attempts need to be made to find a pre-image for a given digest. A hash function where an adversary can find a pre-image with less attempts is deemed to not be pre-image resistant and hence compromised.

### 2.3.4 Second pre-image resistance

For a cryptographic hash functions to have second pre-image resistance, given a message $m_1$, it needs to be difficult to find another input $m_2$ such that $H(m_1) = H(m_2)$. In other words, given a message, it needs to be difficult to find another message with the same hash [9].

Like pre-image resistance, a hash function that is considered second pre-image resistant means that a second pre-image can be found in time complexity $O(2^n)$. This is because the adversary first to compute the digest $d = H(m_1)$ before finding another pre-image, turning this problem into one of finding a pre-image for a given digest.

#### *2.3.4.1 The relationship between pre-image and second pre-image resistance*

A hash function that is second pre-image resistant is also pre-image resistant. As explained above, finding a pre-image of a digest is a sub-problem of finding a second pre-image. However, a hash function that is pre-image resistant may in theory not be second pre-image resistant, demonstrated below.

Suppose a hypothetical hash function $H$ works by taking the first 256 characters of an input message, padding it with '0's if it is shorter than 256 characters and then computing the SHA-512 hash of the padded string of length 256. This hypothetical hash function is pre-image resistant as it is currently infeasible to reverse a SHA-512 hash. However, because of the padding function we defined, it is not second pre-image resistant. Given an input $m_1 =$ 'abc', we can trivially find a second pre-image $m_2 =$ 'abc0' which will produce the same digest, satisfying the requirement of $H(abc) = H(abc0)$.

### 2.3.5 Collision resistance

A hash collision occurs when the following condition is satisfied: $H(m_1) = H(m_2)$ for any inputs $m_1, m_2$ [8]. In other words, a collision occurs where any two messages with the same hash output is found. The difference between finding a collision and finding a second pre-image is neither of the messages are given when finding a collision.

So, in a sense, the requirement is looser. For a hash function with the property of collision resistance, the time complexity for finding a collision between any two inputs is $O(2^{\frac{n}{2}})$ and we justify this by an explanation using the birthday problem below. A hash function that is not collision resistant is able to be exploited so that a collision can be found in polynomial time [10].

#### *2.3.5.1 Birthday Problem*

The birthday problem involves the following thought experiment. The chance of having the same birthday as any random stranger is quite low: $P = \frac{1}{365}$. However, the probability of finding two people out of a group of around 20 to 30 with the same birthday becomes quite large, larger than what one may intuitively expect.

Let us calculate the probability $P(S)$, that two people in a room of $n$ people have the same birthday. To do this, we first calculate the probability that no people in the room have the same birthday. The first person naturally does not have the same birthday as another, since there is nobody yet to compare to. The second person has a $\frac{1}{365}$ chance of having the same birthday as the first person and hence a $1 - \frac{1}{365}$ chance of having a different birthday. The third person has a $\frac{2}{365}$ chance of having a non-unique birthday (comparing to the first two people) and hence a $1 - \frac{2}{365}$ chance of having a unique birthday. This is mathematically described below, where $\overline{P(S)}$ is the probability that everyone has different birthdays.

$$\overline{P(S)} = 1 - P(S) = 1 \times \left(1 - \frac{1}{365}\right) \times \left(1 - \frac{2}{365}\right) \times \ldots \times (1 - \frac{n-1}{365})$$

Manipulating this expression, we get:

$$\overline{P(S)} = \frac{365 \times 364 \times \ldots \times (365 - n + 1)}{365^n} = \frac{365!}{(365-n)!} \times \frac{1}{365^n} = \frac{n!\binom{365}{n}}{365^n} = \frac{{}^{365}P_n}{365^n}$$

So:

$$P(S) = 1 - \frac{{}^{365}P_n}{365^n}$$

We find for a small value of $n = 23$, the probability of two people having the same birthday in a group of 30 is $1 - \frac{{}^{365}P_{23}}{365^{23}} = 50.7\%$.

### *2.3.5.2 Application of the birthday problem to cryptographic hashing*

Note that $\sqrt{365} \approx 23$. So, scaling the birthday problem up to an $n$-bit hash with $2^n$ possible outputs, after $\sqrt{2^n} = 2^{\frac{n}{2}}$ attempts, we have a 50% chance of finding two inputs that map to the same output.

An alternative way to look at it is with $2^{n/2}$ values, there are ${}^{2^{\frac{n}{2}}}C_2 = \frac{2^{\frac{n}{2}}!}{2!\left(2^{\frac{n}{2}}-2\right)!} = \frac{2^{\frac{n}{2}}\times(2^{\frac{n}{2}}-1)}{2} \approx \frac{2^{\frac{n}{2}}\times 2^{\frac{n}{2}}}{2} = \frac{2^n}{2} = 2^{n-1}$ pairs. With each pair, there is a probability of $\frac{1}{2^n}$ for a collision so with $2^{n-1}$ pairs, there is a $\frac{2^{n-1}}{2^n} = \frac{1}{2}$ chance of a collision. So, on average, the time complexity of finding a hash collision is $O\left(2^{\frac{n}{2}}\right)$.

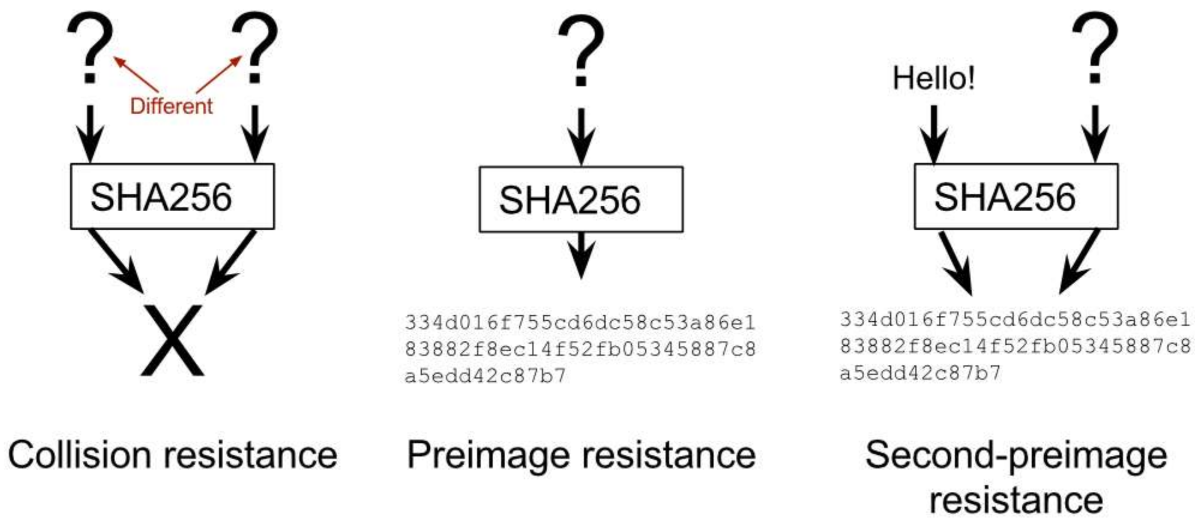
?
?
Different
SHA256
X
Collision resistance
?
SHA256
334d016f755cd6dc58c53a86e1
83882f8ec14f52fb05345887c8
a5edd42c87b7
Preimage resistance
Hello!
?
SHA256
334d016f755cd6dc58c53a86e1
83882f8ec14f52fb05345887c8
a5edd42c87b7
Second-preimage resistance

*Figure 2.1 An illustration of pre-image, second pre-image and collision resistance* [11]

## 2.4 Countermeasures

A hash function is deemed to have pre-image, second pre-image and collision resistance if the time complexity of finding pre-images/collisions align with the definitions above. Hash functions where pre-images or collisions are found with a much lower time complexity are deemed to be broken and are not recommended for use.

### 2.4.1 Cryptanalysis

In practice, it is most common to search for a collision, as the time complexity is significantly lower than finding a pre-image. Cryptanalysis is used to study the hash function and attempt to find weaknesses. An example is MD5 hash function, which produces digests 128 bits long. Using a birthday attack, the time complexity of finding a hash collision is $2^{\frac{128}{2}} = 2^{64}$. However, an exploit exists allowing a collision to be found with a complexity of $2^{24.1}$ [12]. Since $2^{24.1} \ll 2^{64}$, the MD5 hash is deemed to be broken.

Another attack is the length extension attack, which targets an inherent flaw in some hash functions. Hash functions that are built on the Merkle–Damgård construction are vulnerable and we will explore this further in section 3.1. Application-specific methods are also used, such as the use of rainbow tables in cracking passwords. This will be examined further in Chapter 0.

# 3 Hashing in Detail

In this chapter we explore in detail three main constructions of cryptographic hash functions: the Merkle-Damgård construction, HAIFA and sponge construction.

## 3.1 Merkle–Damgård construction

In this section, we will first describe one-way compression functions and iterative hash functions, which are concepts used in the Merkle-Damgård construction. We will then describe the specifics of the Merkle-Damgård construction, examine its key weaknesses and study an example hash function constructed in this manner.

### 3.1.1 One-way compression functions

A one-way compression function takes a larger input of fixed length and compresses it into a smaller output, also of fixed length [13]. Because it is a one-way function, it is difficult to take an output and determine what the original inputs were.

One type of one-way compression function takes in two inputs and produces an output of the same length as one of the inputs. For example, suppose a function takes in two inputs of 256 bits and produces one output of 256 bits. This function essentially compresses 512 bits of information into 256 bits.

We will explore in section 3.1.3 how these compression functions are a component of hash functions made using the Merkle-Damgård construction. However, it is worth noting that since these compression functions are one-way functions, the properties of pre-image resistance, second pre-image resistance and collision resistance, which are desirable for cryptographic hash functions, are also desirable for these compression functions too. One-way compression functions should also have an avalanche effect. In particular, each output bit depends on every input bit. It is this interdependency that causes several output bits to change when a single input bit is flipped.

### 3.1.2 Iterated hash functions

In this section, we explore iterated hash functions in a general sense and how they can be constructed using one-way compression functions. If we denote an m-bit string as $\{0, 1\}^m$, we can express the one-way compression function described in the previous section as: $f: \{0, 1\}^{m+t} \rightarrow \{0, 1\}^m$. So we essentially have a function that condenses $m + t$ bits into $m$ bits. We can now describe a hash function as taking in an arbitrary length input and producing an output of fixed length $m$: $h: \{0, 1\}^* \rightarrow \{0,1\}^m$ [13].

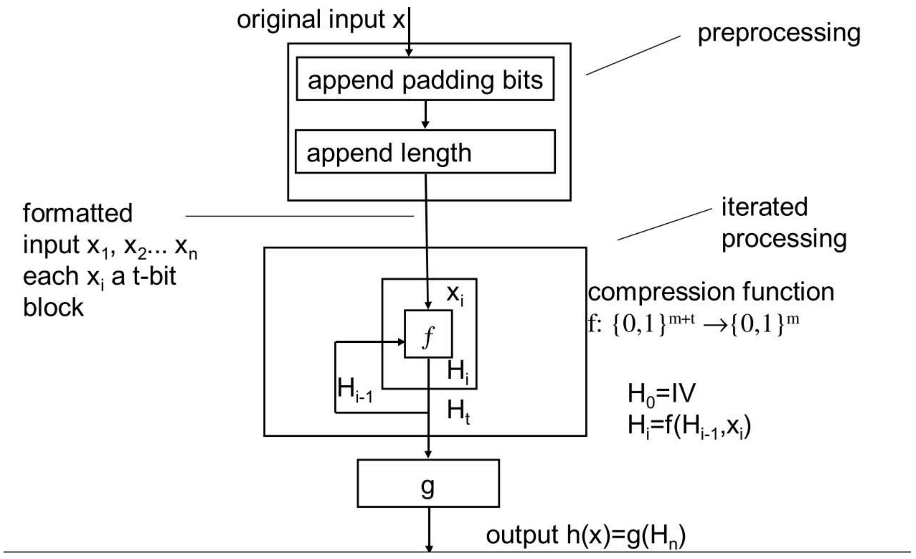

*Figure 3.1 A diagram of the iterated hash function concept* [13]

An iterated hash function works as follows:

1. Pre-process the original input $x$ by appending padding bits until it is a length that is a multiple of $t$. Then break it down into t-bit blocks: $x_1, x_2, \ldots, x_n$.
2. Let $H_0$ be the initialisation vector. Iteratively compute $H_i = f(H_{i-1} \,||\, x_i)$ where || represents string concatenation. In other words, we take each of the t-bit blocks above and compound them with the existing hash of length $m$ and then run it through the compression function.
3. The output of the last iteration, $H_n = f(H_{n-1}||\, x_n)$, is the final output of the overall hash function.

### 3.1.3 How the Merkle-Damgård construction works

The Merkle-Damgård construction is based on the iterated hash function. Firstly, the input is padded so that it is a multiple of a fixed size. Then the padded input is broken into blocks of this fixed size. The hash function continually takes the next block of input and combines it with the output of the previous round, feeding this into the compression function again, in the manner described in section 3.1.2. We now go into depth to describe the method used to pad the input and the use of a finalisation function which modifies the internal state before the final output is produced. The way these two stages are implemented can 'harden' the hash, or in other words, make the hash function less vulnerable to collision attacks.

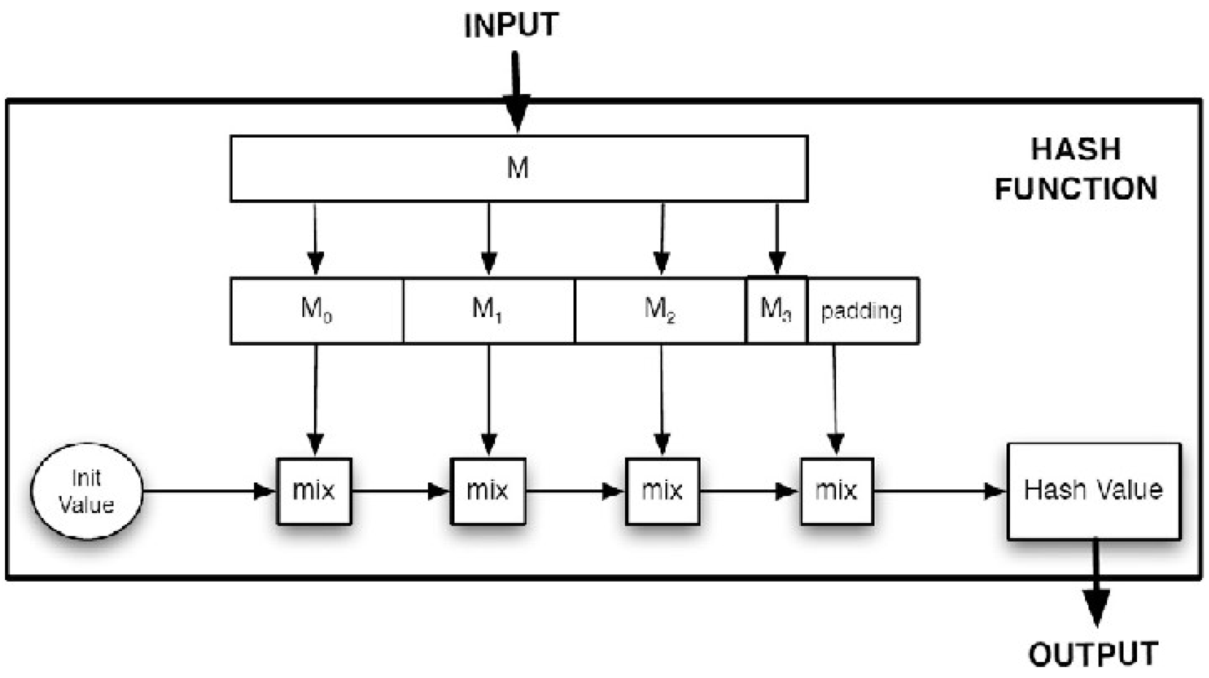

*Figure 3.2 A diagram of the Merkle-Damgård construction* [14]

The method used to pad the input up to the desired length must be chosen carefully. Suppose the hashing algorithm simply pads the inputs with zeros. In this case, `PotatoPie` and `PotatoPie00` both result in the blocks `PotatoPi` and `e0000000`, meaning that the hash output will be the same for both strings, producing a collision. Instead, padding must be 'MD-compliant' by ensuring that inputs of different lengths differ by their last block [15]. i.e. if $|M_1| \neq |M_2|$, then the last block of $Pad(M_1)$ is different to $Pad(M_2)$. One example of an MD-compliant hash is using the final block to represent the length of the input. So `PotatoPie` will be padded to produce `PotatoPi e0000000 00000009` and `PotatoPie00` will be padded to produce `PotatoPi, e0000000, 00000010`. As a result, these two inputs will hash into different outputs.

Hash functions using the Merkle-Damgård construction may also have a finalisation function. This may make the function less susceptible to length extension attacks, which we will discuss in section 3.1.7. Finalisation functions take the output of the final round of iteration. One purpose of the finalisation function may be to compress the internal state into a smaller output hash size. For example, if the compression function used internally produces an output of 256 bits, but the output of the hash function needs to be 128 bits, the finalisation function could truncate the second half of the 256 bits and use the remaining 128 bits as the output of the overall hash function. The finalisation function can also serve to improve the avalanche effect of the hash function.

### 3.1.4 Wide pipe construction

Several variations of the Merkle-Damgård construction have been proposed. The wide pipe construction differs from the traditional construction in that the internal compression function works with a larger input. The compression function in the traditional approach takes in $n$ bits from the previous round (or the initialisation vector) and $m$ bits from the next block of the message to be hashed. A wide pipe approach uses a compression function that takes in $w + m$ bits and compresses this to $w$ bits instead, where $w > n$ [16]. It uses a finalisation function at the end to condense the output from $w$ to $n$ bits.

We can express wide pipe construction in the following manner, with compression functions $f: \{0,1\}^w \times \{0,1\}^m \to \{0,1\}^w$ and finalisation function g: $\{0,1\}^w \to \{0,1\}^n$ and $H_0 \in \{0,1\}^w$ being the initialisation vector used in the first round [16].

$$H_i = f(H_{i-1}, M_i), i = 1 \dots t$$

$$h(M) = g(H_t)$$

### 3.1.5 Double pipe construction

A similar approach to the wide pipe construction is the double pipe construction. Two compression functions run in parallel in the following manner, where the compression can be expressed as $f: \{0,1\}^n \times \{0,1\}^{n+m} \to \{0,1\}^n$ and $H_0', H_0'' \in \{0,1\}^n$ are the initialisation vectors used in the first round of compression [16].

$$H_i' = f(H_{i-1}', H_{i-1}'' || M_i), i = 1 \dots t-1$$

$$H_i'' = f(H_{i-1}'', H_{i-1}' || M_i), i = 1 \dots t-1$$

$$h(M) = f(H_{t-1}', H_{t-1}'' || M_t)$$

In a sense, one can visualise this as cross-pollination between two parallel hashing processes. Both the wide pipe and double pipe construction serve to double the size of the internal state of the hash function and improving its collision resistance [16].

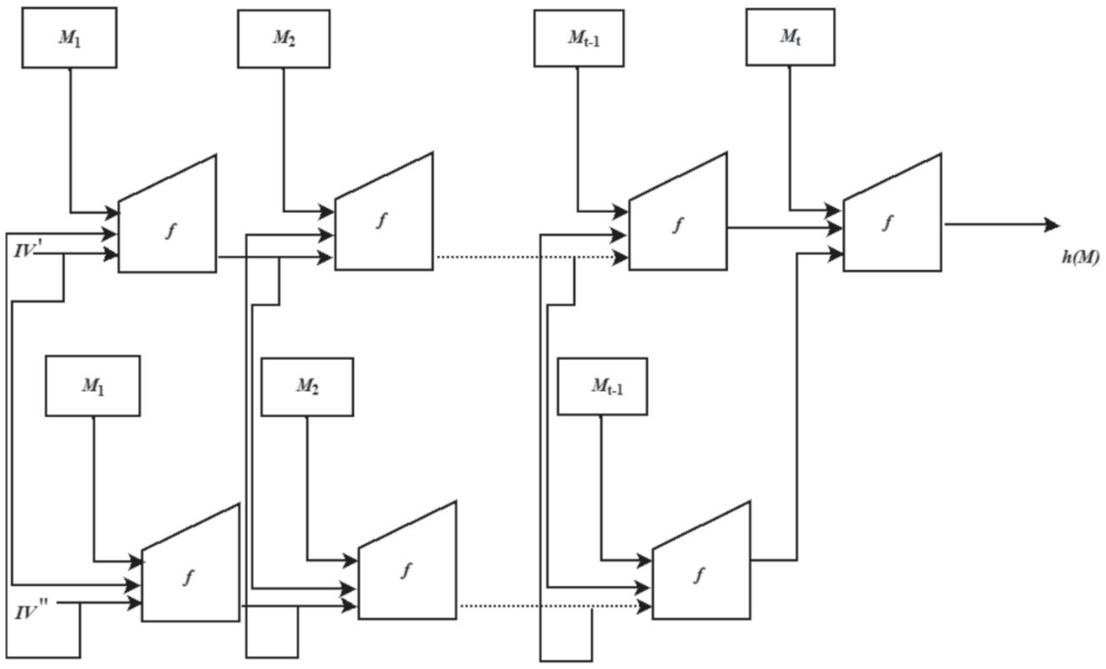

*Figure 3.3 A diagram of the double pipe construction* [16]

### 3.1.6 Tree construction

Another approach is the use of tree construction. Although this makes the hashing more parallelizable, it is not practical as the size of the tree grows exponentially with the size of the message. This approach is suitable for specialised use cases where there is an abundance of parallel computation power [16].

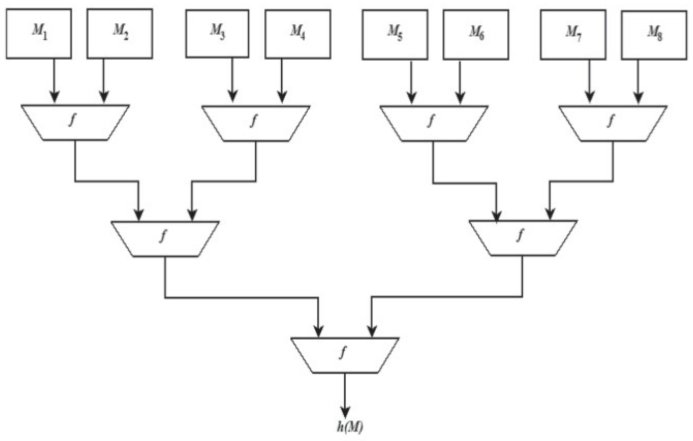

*Figure 3.4 A diagram of the tree construction* [16]

### 3.1.7 Weaknesses

#### *3.1.7.1 Nostradamus attack*

Some hash functions built using the Merkle-Damgård construction are vulnerable to the Nostradamus attack. This attack exploits the cascading nature of the Merkle-Damgård construction and their insufficient Chosen Target Forced Prefix pre-image resistance [17]. This attack allows the user to create different documents with different hashes. Then, in the case of PDFs, a hidden image can be appended to the second document. The contents of the image in second document is changed repeatedly until the hash of the second document matches the first.

The Eindhoven University of Technology demonstrated this by making multiple predictions on the winner of the 2008 US Presential elections. Because they had PDFs of a range of predictions, each with the same hash, they could simply substitute the corresponding PDF after the election was completed, claiming their prediction was correct [18]. Taking a theoretical approach, suppose we have a

128-bit hash function. By appending a hidden image within the PDF with 128 pixels and permuting through all possible black and white images (there are $2^{128}$ such images), using the pigeonhole principle, we are likely to have permuted through all the possible hash values or alternatively found a collision in the hash function.

#### *3.1.7.2 Length extension attack*

Hash functions built using the Merkle-Damgård construction are also vulnerable to length extension attacks. This is because the output once the hash function has completed is simply the internal state of the hash function at that point in time, due to the iterative nature of the hash calculation described in section 3.1.3. We can express this as: given $H(X)$, we need to find $H(Pad(X)||Y)$ where $H$ is the hash function, $Pad$ is the padding function and || represents concatenation. The crucial concern here is that we can append data to $X$ and produce a matching hash, without knowing what $X$ was in the first place [19].

The length extension attack is often used to exploit message authentication codes (MACs). MACs involve sending a message accompanied by a verification code to provide the cryptographic property of integrity. The receiver calculates the verification code using the received message and if it does not match the verification code sent with the message, then it is likely the message was tampered with during transmission.

We now present a simplified example of a length extension attack, which omits the use of a padding function. So, given $H(X)$, we need to find $H(X||Y)$. The hash function operates as follows. Note that it is in the form of an iterated hash function, described in section 3.1.2.

1. For the first character of input, convert it to its ASCII code
2. Then multiply that number by 521, mod it by 1000, add 8538 and mod it by 734
3. Add that value onto the next character and repeat

Suppose a financial institution and an ATM communicated with each other using this protocol. For each communication, the message $M$ is sent along with $H(P||M)$, where $P$ is a shared secret both institutions have. A message is treated as valid if the hash calculated from the received message and the secret $P$, $H(P||M)$ matches the hash that was sent. In theory, an eavesdropper is unable to intercept and modify a message, since it does not know the password.

The eavesdropper intercepts the message `WITHDRAW100` with a hash value of 488. The eavesdropper then modifies the message to `WITHDRAW1000`. Because the previous hash value of 488 was simply the internal state of the hash function, we can take that internal state and continue processing the new data, an extra `0` appended to the end of the message. The ASCII code of `0` is 48 so we perform the following:

$$\big(((488 + 48) \times 521) mod\ 1000 + 8538\big)\ mod\ 734 = 720$$

So, we now have the message `WITHDRAW1000` and the corresponding hash `720` and we were able to calculate this without knowledge of the shared secret. By sending the new message with the valid hash, the altered message will be incorrectly treated by the receiving party as legitimate.

### 3.1.8 An example of a Merkle-Damgård hash: MD5

The MD5 hash produces an output of 128 bits. The compression function works in principle like the iterated hash function described earlier, however its implementation has greater complexity.

The padding function first appends a single 1 bit to the end of the input and then followed by zeros until it is of a length $512k - 64$, or 64 bits less than a multiple of 512. The last 64 bits are then used to represent the length of the original message before padding [20]. This is an example of MD-compliant padding described in section 3.1.3 and serves to make length extension attacks less straightforward, but not impossible.

The compression function operates on a 128-bit state, but unlike the simpler examples described earlier, the 128-bit state is split into four 32-bit words. Let us represent these words as $A, B, C$ and

$D$. Each time a new 512-bit block of the message is consumed, the algorithm completes 4 rounds of 16 operations for a total of 64 operations.

For each of the 16 operations in each round, the function is computed as follows. There are four possible functions that are used, one for each of the 4 rounds of 16 operations.

$$F(B,C,D) = (B \wedge \mathrm{C}) \vee (\neg \mathrm{B} \wedge \mathrm{D})$$

$$G(B,C,D) = (B \wedge \mathrm{D}) \vee (\mathrm{C} \wedge \neg \mathrm{D})$$

$$H(B,C,D) = B \oplus \mathrm{C} \oplus \mathrm{D}$$

$$I(B,C,D) = C \oplus (\mathrm{B} \vee \neg \mathrm{D})$$

The result is then added to $A$. $M_i$, the next 32-bit piece of the current 512-bit message block, is then added to $A$. $K_i$, a 32-bit constant that changes for each of the operations is then added to $A$. The value of $A$ is then left bit-rotated by certain number of places that changes for each of the operations. The value of $B$ is then added to $A$. All these additions to $A$ are done in modulo $2^{32}$. The finally, $A, B, C, D$ is assigned to $B, C, D$ and $A$ respectively.

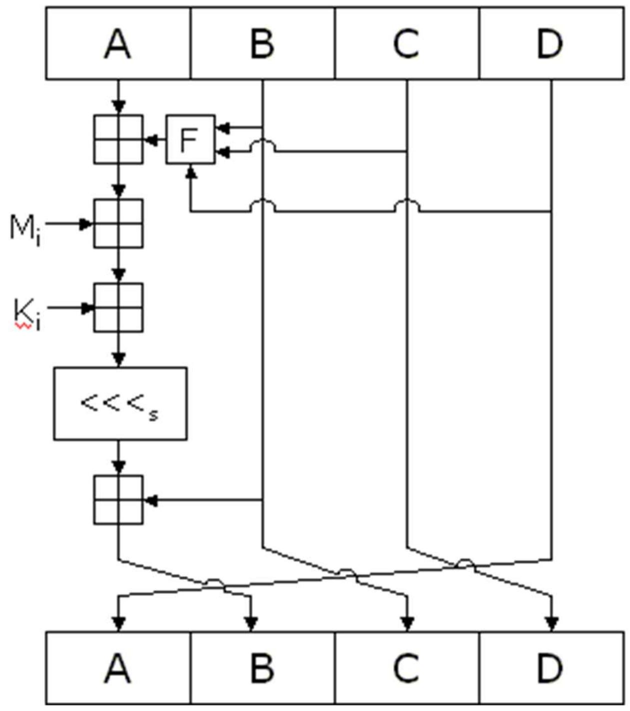

*Figure 3.5 A diagram of the MD5 hash function* [20]

The MD5 hash is now considered broken, as numerous exploits have been found in it. As discussed in section 2.4.1, an exploit exists allowing a collision to be found with a complexity of $2^{24.1}$ which shows MD5 is not collision resistant [12]. One of the main implications of this is that documents and digital certificates verified with MD5 hashes can be compromised, as it is possible to generate an in-authentic copy with details modified while still having the same hash. The role of cryptographic hashing in digital certificates and public key infrastructure (PKI) will be explored in greater depth in section 4.4.

## 3.2 HAIFA construction

In this section we first describe the difference in structure between the Merkle-Damgård construction and the HAIFA construction. We then outline various features of the HAIFA construction: the padding function, resistance to length-extension attacks and use of salts and show how this addresses weaknesses in the Merkle-Damgård construction.

HAIFA construction was developed to address some of the shortcomings of the Merkle-Damgård construction. Internal collisions occur when there are hash collisions in the internal states of the function while the message is being digested. Hashes built using the Merkle-Damgård construction were more susceptible to internal collisions. Like the Merkle-Damgård construction, HAIFA construction is also iterative in nature and involves the use of compression functions. HAIFA enhances the Merkle-Damgård construction by adding a counter storing the number of bits hashed so far and an optional salt (a randomly generated string) to every step of the iteration, making it less likely for a collision to occur in the internal state of the hash [21]. We can express the compression function $C$ as:

$$h_i = C(h_{i-1}, M_i, \#bits, s)$$

where $h_{i-1}$ is the hash value from the previous iteration, $M_i$ is the current message block, $\#bits$ is the counter described above and $s$ represents the salt.

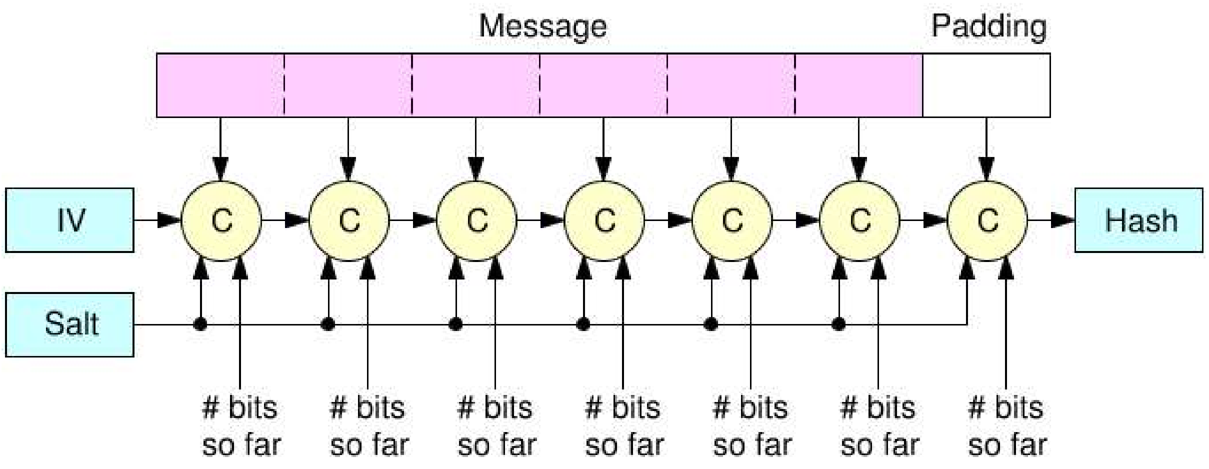

*Figure 3.6 A diagram of the HAIFA construction* [22]

The padding function is similar to the MD-compliant function described in section 3.1.3. Along with including the length of the original message in the padding, the digest size of the hash function is also included at the end. Many hash functions such as SHA-2 have variants with different digest output sizes. Suppose two messages $M_1$ hashed with a digest size of $d_1$, and $M_2$ hashed with a digest size of $d_2$, are found such that there are internal collisions. The last block, which will differ between the hashes due to the different digest sizes of the function will change this behaviour and reduce the chance of a collision in the final hash output [23].

As discussed in section 3.1.7.2, the Merkle-Damgård construction is vulnerable to length extension attacks for message authentication codes (MACs) in the form of $H(k, M)$, which is when the message $M$ is appended to the end of a secret key $k$ and this value is hashed. As shown above, HAIFA construction is different in the sense that each block is also compressed with a counter value holding the number of bits processed so far. As a result, the internal state of the function is further obscured from the point of view of the attacker [23].

The use of salts, which is a randomly generated string chosen by the implementer or user of the hash increases the effort required to find a collision in the hash. Because the salt is not known in advance, the attacker cannot pre-compute hash values offline. Computations must be done with the actual system being attacked, which is often slower. Furthermore, any pre-image attack needs to also take the salt into consideration, creating more work for the attacker.

### 3.2.1 BLAKE2

BLAKE2 is a hash function built using the HAIFA construction. It offers a number of features such as support for parallel computation and personalisation, out of the box. The user can also tailor the hash to their own individual application, by specifying a salt value, a feature described in the previous section. Users can optionally specify a key for use in keyed hashing. The application of this functionality is to simplify the construction of MACs while maintaining immunity to length extension attacks [24].

BLAKE2's authors make speed and low footprint a key emphasis of this hash function. Speed is important in hash functions as applications often require a large throughout of data. For example, file integrity checking requires processing of up to gigabytes of data per file. Password hashing on a server needs to be capable of handling thousands of concurrent login requests.

Faster computation is achieved by using fewer rounds of computation while maintaining security. Bit-wise arithmetic such as rotation is optimised for the latest Intel CPUs, reducing the cost of the compression function from 18 to 16 instructions, which equates to a 12% improvement [24].

## 3.3 Sponge construction

In this section, we will first introduce sponge functions as a mathematical concept. We will then outline its application in building cryptographic has functions. Then we will briefly describe the features of a hash function created using the sponge construction: SHA-3.

A sponge function has three components: a state memory $S$, a transformation function and a padding function. The state memory $S$ of size $b$ bits contains two components: a component of size $r$ (we call this the bitrate) and a component of size $c$ (we call this the capacity) [25]. The sponge function operates in two stages, the ‘absorption’ stage and the ‘output’ stage. This can be imagined with a metaphor of an actual sponge-like material.

In the absorption stage, the input is padded until it is a multiple of $r$. Then each individual r-bit block $B$ is absorbed into the function by repeating the following two steps:

$$R \leftarrow R \oplus \mathrm{B}$$

$$S \leftarrow f(S)$$

where $\oplus$ represents the XOR operation and $f$ is a function that transforms the $b$ bit state memory into another $b$ bit state memory. We can represent this function as $f: \{0,1\}^b \rightarrow \{0,1\}^b$. This is often implemented as pseudorandom permutation of a string sized $2^b$. A simple example of $f$, where $b = 2$, would be $\{0,0\}, \{1,0\}, \{1,1\}, \{0,1\}$ where an input would map from one value into the value that follows. More generally, the function $f$ functions in a similar way to a block cipher, mapping its input into another output of the same length. In a sense, this can be visualised as adding ingredients to a pot and stirring the pot before adding further ingredients [26].

The output stage follows a similar concept but, metaphorically speaking, we take bits out of the pot rather than adding them in. Firstly, the $r$ bit component is outputted and then we run the whole state $S$ through the function $f$. This continues until the required output length is reached.

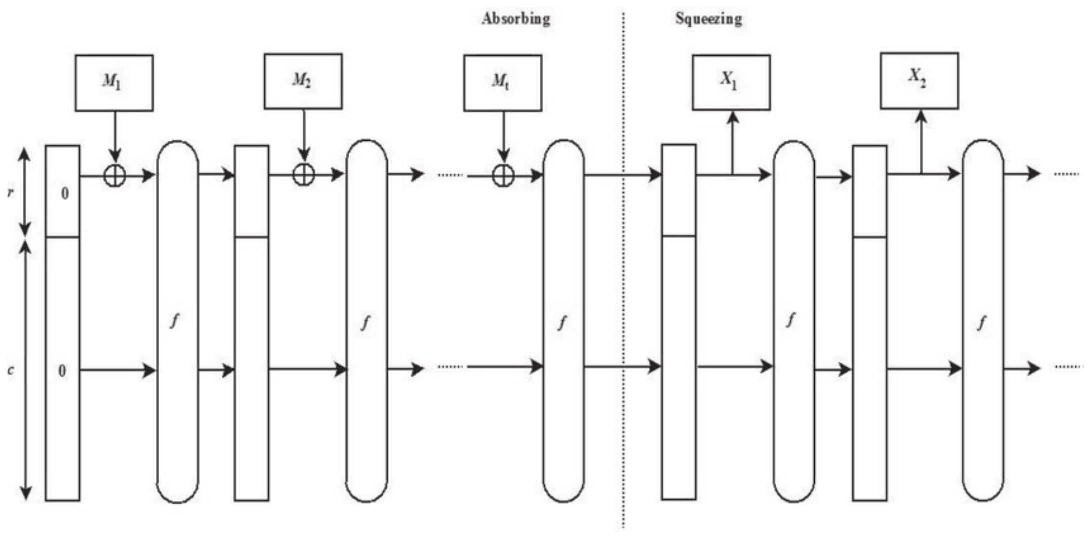

*Figure 3.7 A diagram of the Sponge Construction* [16]

A slight variation of this approach is the duplex construction. Instead of having an absorption stage followed by the output stage, absorption and output are interleaved. The function takes in a $r$ bit block of input and then the function $f$ is applied to $S$. The $r$ bit component of the state is now appended to the output and $f$ is applied to $S$ again [27]. This process then repeats itself until the desired output length is reached.

Sponge functions can be used in building cryptographic hashes. They are different from hash functions built using the Merkle-Damgård construction, where the hash value produced is based on the internal state of the hash function at the end of computation. In contrast, with the sponge construction, the $c$ bit component of the state $S$ is never directly modified by input nor directed taken as output. The $c$ bit component can only be altered by the function $f$. This increased obscurity, from an attacker point of view, giving immunity to attacks such as the length extension attack.

### 3.3.1 SHA-3

The SHA-3 hash function is based on the Keccak family of functions and was the winner of the NIST hash function competition which ended in October 2012. Keccak is built using the sponge construction, one of the newer constructions which, like HAIFA, was designed to address some of the shortcomings of the Merkle-Damgård construction.

Similarly to the HAIFA construction, the padding function also include the digest size, so that different hash variants of a message won't have a common prefix. A demonstration of this is where the SHA3-224 hash of `cat` is

```
447c857980c93d613b8bd6897c05bfd0621245139f021aaa6b57830a
```

while the SHA3-256 hash of `cat` is completely different:

```
d616607d3e4ba96a74f323cffc5f20a3c78e7cab8ecbdbb03b13fa8ffc9bf644
```

The SHA-3 hash function has a state $S$ of size 1600 bits, which is arranged as an array of $5 \times 5$ $64$ bit words. The function $f$ consists of five steps. It uses a combination of XOR, AND and NOT, bitwise rotation and parity computation operations to make the implementation in hardware simple and efficient [28]. The possible output sizes of the SHA-3 hash function are 224, 256, 384 and 512 bits.

In December 2016, NIST released new standards, specifying additional variants of SHA-3. KMAC is a keyed hash function that takes in a key as well as an input string. The main application of this is the construction of MACs, which is set along with a message to confirm its integrity during transmission. TupleHash hashes multiple strings at once. ParallelHash modifies the SHA-3 function to allow parallel computation, reflecting the fact that a wide range of computation devices, even in embedded or mobile uses have multicore processors [29].

# 4 Main Applications of Cryptographic Hashing

In this chapter, we provide an explanation for the overarching purpose of hashing, before delving into its various applications.

## 4.1 Overall purpose of hashing

Hashing is used in cryptographic applications where integrity is required. It often revolves around verifying whether two digital objects are the same at a given point in time or whether a digital object has changed over time. This needs to be done without revealing the content of that digital object.

To illustrate this, imagine a mathematics society running a competition where they release a hard problem. To prove that they know what the correct answer is, they also release the hash of the correct answer along with the problem. Eventually a competitor manages to solve the problem and proves this to the community by sharing the hash of the solution. Because this hash matches the hash published by the mathematics society, we can confirm that this competitor got the right answer without publicly revealing the answer. We can also demonstrate the importance of collision resistance here. Suppose the competitor was able to obtain an incorrect solution that hashes to the same value as the correct solution. In this case, the integrity of this competition would be violated.

## 4.2 Passwords

One of the main applications of hashing is the storage of passwords. In this section, we describe how hashes are applied in authentication as well as various enhancements that may be used.

To authenticate a user, a server needs to know whether the password entered is correct or incorrect. A naïve approach would be to store a list of usernames and their corresponding passwords in the database and do a simple lookup when a user makes a login request. However, it is easy to see that if an attack gained access to this database, the login credentials would be available to them instantly. Some organisations have, in the past, used plaintext to store passwords. An example of this is where Facebook stored user credentials on internal company servers, in a manner so that it would be accessible by employees [30].

A more acceptable approach would be to take the submitted password when the user is registering for the site, run it through a cryptographic hash and store that hash along with the username. As a result, it is infeasible for an attacker or even the site itself to figure out a user's password by looking at what is stored in the database. To login, the user types the password and the authentication system hashes this value. Because of the second pre-image resistance of sound cryptographic hashes, we can safely say that if the two hashes are equal then the original passwords were also equal and hence, we are able to authenticate this user. Note that we are able to make this claim without the knowledge of the password, instead only by examining the hashes.

This application also demonstrates the importance of collision resistance. Suppose that an attacker is able to devise an incorrect password that hashes to the same value as the correct password. In this case, the attacker is able to gain access to the system without knowledge of the correct password.

### 4.2.1 Salting

Let us refer to hash cracking as the act of taking a hash and obtaining a pre-image, which in this case is original password. Lookup tables or rainbow tables may be used in attempt crack hashes. Here, we make use of a space-time tradeoff, reducing the computation required but increasing the storage requirement to hold this lookup table.

To defend against a pre-computed hash attack, salts can be added to passwords. Salts are randomly generated strings that are appended to passwords. This extended string is then hashed, and that hash is stored in the database along with the salt. When a user makes a login attempt, the password input is combined with the salt from the database and hashed. This hash is then compared to the stored hash. There are two primary benefits. Firstly, salts effectively extend the password, increasing the number of possible passwords that need to be tested in a cracking attempt. An 8-character password with a 16-character salt is essentially a 24-character password. Secondly, commonly used passwords would otherwise hash to the same value hash into completely different values, revealing a pattern to an attacker with access to the whole database. Due to the avalanche effect, two identical passwords with different salts will hash to completely different values, as shown with the password `qwerty`, an 8-character salt and their corresponding SHA3-224 hashes below:

$$qwerty4i9sN7AH \rightarrow 10be25c3a8d28f33b7638b9fd22fda679d9108e4087c56c45ae067b1$$

$$qwertyYDj6LSMg \rightarrow 81f3154f88ce1b6fa541e9972605bd30f9f875329e5c6831d6a4195e$$

### 4.2.2 Key stretching

Key stretching is another enhancement to password hashing to improve security. Key stretching involves hashing a password repeatedly to increase the time required for the computation. We can express it as follows, where $x_0$ is an empty string, $p$ is the password, $s$ is the salt, $H$ is the hash function and || is string concatenation:

$$x_i = H(x_{i-1}||p||s)$$

If we take $x_r$, as the hash to be stored, then this computation takes $r$ times longer than a standard hashing operation. The aim is to create a desirable work ratio, by choosing an $r$ that is affordable for a legitimate user but prohibitively time consuming for an attacker [31].

## 4.3 Proof of work

A proof of work is a computation result that is hard to obtain but easy to verify. In this section, we describe how one approach to proof of work manipulates the concept of hashing, retaining the ease of verification of a hash but increasing the complexity of obtaining a valid result that can be considered a proof of work [32]. We examine the idea behind Hashcash, the proof of work system used by bitcoin.

A block in bitcoin consists of multiple transactions. Each block has a header that includes the hash of the previous block, a hash based on all the transactions in the current block, the current block timestamp, the target value and a Nonce value. For the bitcoin network to accept a new block as valid, the block header must be hashed to a value that is lower than the level of difficulty [33]. To express this more simply, the block header must be hashed so there are at least a certain number of leading zeros in the hash. Each time a block is mined, the block and its transactions are added to the chain and the miner is rewarded with bitcoin, to provide an incentive to process these transactions.

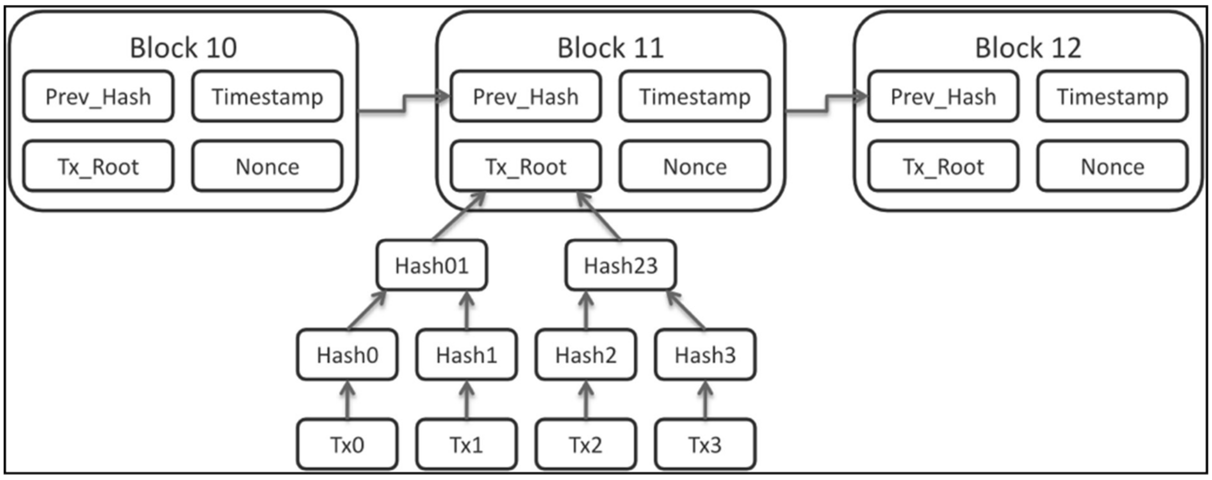

*Figure 4.1 A diagram of the block headers in Bitcoin* [33]

Because the total number of bitcoins in circulation is capped at 21 million, the network aims to have one new block mined at a limited rate of every 10 minutes. To mine a block, the miner hashes the block header described above. If the value of the has is below the target value, this block is accepted by the network. If not, the Nonce value is incremented, and the miner tries to hash the block again. By decreasing the target value and hence increasing the number of leading zeros required, the probability of hashing the header to a value that is below the target value decreases exponentially.

The SHA256 hash function is used, which has a total of $2^{256} = 16^{64}$ values, since it can be expressed as 64 hexadecimal digits, each with 16 possible values. Suppose there is a requirement of 17 leading zeros, then the total number of valid hashes is $16^{47}$ since there are 17 pre-determined and and 47 remaining hexadecimal characters. So, the probability of hashing a valid result is:

$$\frac{16^{47}}{16^{64}} = 3.39 \times 10^{-21}$$

which is miniscule. If an additional leading zero is required, the probability of hashing a valid result decreases further by a factor of 16.

It is also worth noting that if an earlier block is tampered with, its new hash will be different to the hash of the previous block currently stored. If a transaction in the current block is tampered with, the value of the hash that is based on all transactions in the current block will also change. As a result, the hash of the current block changes and this propagates forwards in the chain, causing the proof of work for every block from that point in time onwards to need to be recomputed. The infeasibility of these re-computations required makes the block chain resistant to tampering.

## 4.4 Digital signatures

Cryptographic hashing is also used in digital signatures. Although it may seem appropriate for a whole document to be signed, in practice, the document is hashed and then signed. Most importantly, this allows for integrity. If a document is modified without authorisation, its hash will change and hence the original signature will be rendered invalid. Without the hash, there is no robust way of confirming whether the document has been modified or redacted in some subtle way. Secondly, it allows documents, which will come in various formats, to be converted into an easy to handle hash output which is just a string.

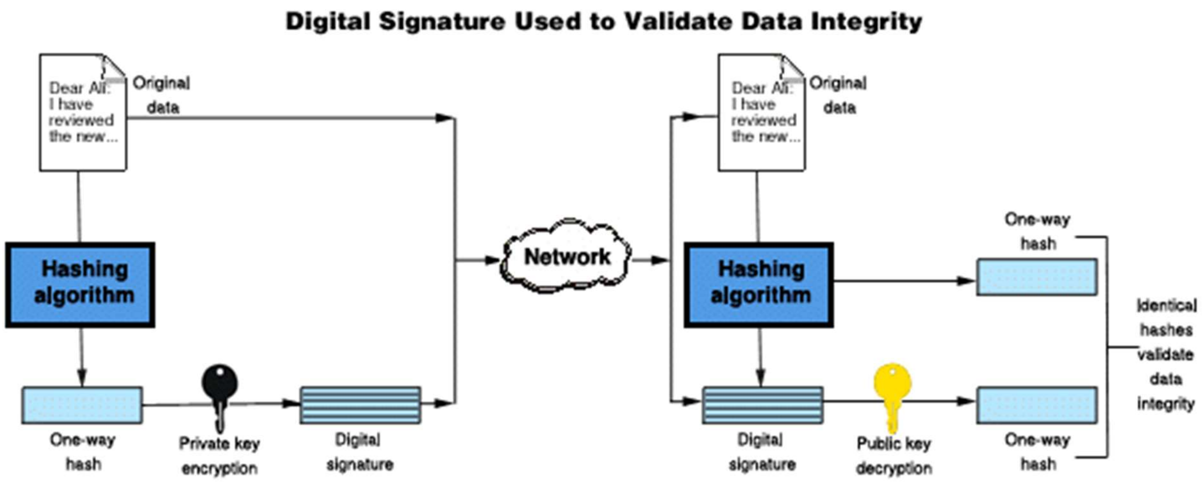

*Figure 4.2 The role of digital signatures in validating integrity* [34]

Cryptographic hash functions are suitable for digital signatures due to their property of second pre-image resistance. Being able to find or generate a second document with the same signature, using the Nostradamus attack described in section 3.1.7.1 or otherwise, would raise security concerns as an attacker can make claims that the signee signed the counterfeit document.

### 4.4.1 Public Key Infrastructure

The use of digital signatures play a role in public key infrastructure, notably in SSL certificates. SSL certificates contain information on who the certificate owner is, the domain of the website, and the certificate authority (CA) that verified this information. This information is then hashed and that hash is signed by the CA [35].

When the web browser navigates to a site over HTTPS, it will take the information that comes with the SSL certificate, hash that information and compare it to the existing hash. If the two hashes match, then the web browser is able to 'trust' the website as it has proven that the information provided is the same information that was verified by the CA. Essentially, the website visited is able to prove "I am who I claim to be" [36].

Suppose an attacker modifies the certificate to include a malicious domain. If such certificate were to be provided to the user, the computed hash would be different and hence the authenticity of the certificate would not be confirmed.

# 5 Summary

In this paper, we have described the concept of one-way hash functions and the additional properties such as pre-image resistance, collision resistance and the avalanche effect that are required for a function to be a cryptographic hash function. We have examined how the Merkle-Damgård, HAIFA and Sponge constructions are built to satisfy these properties.

In summary, a cryptographic hash functions maps an arbitrary input into a unique fixed length output. It is infeasible to use this output to determine what the input was and given a pair of input and output, it is infeasible to generate a second input that hashes to the same output. This behaviour is used in cryptographic applications that require integrity. Some applications such as password hashing involve verifying that two pieces of information are the same without revealing that information. Its use in cryptocurrencies and digital signatures involve verifying that a piece of information has not been modified over time.

# Bibliography

[1] E. W. Weisstein, “Hash Function.”

[2] J. Erickson, “Hash Tables,” no. 1981, 2013.

[3] S. Hanov, “Throw away the keys: Easy, Minimal Perfect Hashing,” 2011. [Online]. Available: http://stevehanov.ca/blog/?id=119. [Accessed: 02-Nov-2019].

[4] T. Chia, “Confidentiality, Integrity, Availability: The three components of the CIA Triad « Stack Exchange Security Blog,” 2012. [Online]. Available: https://security.blogoverflow.com/2012/08/confidentiality-integrity-availability-the-three-components-of-the-cia-triad/. [Accessed: 02-Nov-2019].

[5] W. Jackson, “Differences between encryption and hashing -- GCN,” 2013. [Online]. Available: https://gcn.com/articles/2013/12/02/hashing-vs-encryption.aspx. [Accessed: 02-Nov-2019].

[6] A. Ali, “The Avalanche Effect - By Aman Ali,” 2018. [Online]. Available: https://hackernoon.com/the-avalanche-effect-cfb6642d35dd. [Accessed: 02-Nov-2019].

[7] A. K. Lenstra, “Key Lengths,” pp. 1–32.

[8] “The In’s and Outs of Cryptographic Hash Functions (Blockgeek’s Guide),” 2017. [Online]. Available: https://blockgeeks.com/guides/cryptographic-hash-functions/. [Accessed: 02-Nov-2019].

[9] B. Preneel, “Second preimage resistance,” in *Encyclopedia of Cryptography and Security*, Springer US, 2005, pp. 543–544.

[10] Wellesley College, “Avoiding collisions Cryptographic hash functions Hash functions,” 2016.

[11] K. Rosenbaum, “Cryptographic Hashes and Bitcoin | Manning.” [Online]. Available: https://freecontent.manning.com/cryptographic-hashes-and-bitcoin/. [Accessed: 15-Nov-2019].

[12] M. M. J. Stevens, B. M. M. De Weger, G. Schmitz, and H. C. A. Van Tilborg, “On Collisions for MD5, TU Eindhoven MSc Thesis, June 2007,” no. June, 2007.

[13] M. Kantarcioglu, “Cryptographic Hash Functions Data Integrity and Source Authentication Cryptographic Hash Functions,” pp. 1–15.

[14] Y. Saez, C. Estebanez, D. Quintana, and P. Isasi, “Evolutionary hash functions for specific domains,” *Appl. Soft Comput.*, vol. 78, pp. 58–69, May 2019.

[15] M. Bellare and S. Goldwasser, “Lecture Notes on Cryptography,” 2008.

[16] H. Tiwari, “Merkle-Damgård construction method and alternatives: A review,” *J. Inf. Organ. Sci.*, vol. 41, no. 2, pp. 283–304, 2017.

[17] J. Kelsey and T. Kohno, “Herding Hash Functions and the Nostradamus Attack,” Springer, Berlin, Heidelberg, 2006, pp. 183–200.

[18] Technische Universiteit Eindhoven, “Predicting the winner of the 2008 US Presidential Elections using a Sony PlayStation 3,” 2007. [Online]. Available: https://www.win.tue.nl/hashclash/Nostradamus/. [Accessed: 05-Nov-2019].

[19] D. Clem, “Hash Length Extension Attacks | WhiteHat Security,” 2012. [Online]. Available:

https://www.whitehatsec.com/blog/hash-length-extension-attacks/. [Accessed: 15-Nov-2019].

[20] Ius Mentis, “The MD5 cryptographic hash function (in Technology &gt; hashfunctions @ iusmentis.com).” [Online]. Available: https://www.iusmentis.com/technology/hashfunctions/md5/. [Accessed: 15-Nov-2019].

[21] B. Denton and R. Adhami, “Modern Hash Function Construction,” no. August, pp. 1–5, 2017.

[22] A. Kaminsky, “CSCI 462—Introduction to Cryptography.” [Online]. Available: https://www.cs.rit.edu/~ark/462/module11/notes.shtml. [Accessed: 15-Nov-2019].

[23] E. Biham and O. Dunkelman, “2007.Biham.ePrint——A Framework for Iterative Hash Functions - HAIFA.pdf,” no. June 2005, pp. 1–20, 2006.

[24] J. P. Aumasson, S. Neves, Z. Wilcox-O’Hearn, and C. Winnerlein, “BLAKE2: Simpler, smaller, fast as MD5,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 7954 LNCS, pp. 119–135, 2013.

[25] G. Bertoni, J. Daemen, S. Hoffert, M. Peeters, G. Van Assche, and R. Van Keer, “Keccak Team.” [Online]. Available: https://keccak.team/sponge_duplex.html. [Accessed: 12-Nov-2019].

[26] R. L. Rivest and J. C. N. Schuldt, “Spritz — a spongy RC4-like stream cipher and hash function,” *CRYPTO 2014 Rump Sess.*, 2014.

[27] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, “Duplexing the sponge: Single-pass authenticated encryption and other applications,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 7118 LNCS, pp. 320–337, 2012.

[28] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, and R. Van Keer, “Keyak v1,” pp. 1–22, 2014.

[29] J. Kelsey, S. Chang, and R. Perlner, “SHA-3 Derived Functions : NIST Special Publication 800-185 SHA-3 Derived Functions :,” pp. 1–32, 2016.

[30] B. Krebs, “plaintext passwords — Krebs on Security,” 2019. [Online]. Available: https://krebsonsecurity.com/tag/plaintext-passwords/. [Accessed: 13-Nov-2019].

[31] J. D. Cook, “Salting and stretching a password,” 2019. [Online]. Available: https://www.johndcook.com/blog/2019/01/25/salt-and-stretching/. [Accessed: 13-Nov-2019].

[32] H. Nguyen, “Bitcoin, Chance and Randomness - Hugo Nguyen - Medium,” 2018. [Online]. Available: https://medium.com/@hugonguyen/bitcoin-chance-and-randomness-ba49a6edf933. [Accessed: 13-Nov-2019].

[33] A. Rosic, “What Is Hashing? [Step-by-Step Guide-Under Hood Of Blockchain],” 2017. [Online]. Available: https://blockgeeks.com/guides/what-is-hashing/. [Accessed: 13-Nov-2019].

[34] IBM, “Digital signatures.” [Online]. Available: https://www.ibm.com/support/knowledgecenter/en/SSB23S_1.1.0.12/gtps7/s7dsign.html. [Accessed: 15-Nov-2019].

[35] “Understanding Hash Functions &amp; What Role Do Hashes Llay in TLS/SSL Certificate.” [Online]. Available: https://www.thesslstore.com/knowledgebase/ssl-support/understanding-hash-functions/. [Accessed: 13-Nov-2019].

[36] P. Nohe, “The Difference Between SHA-1, SHA-2 and SHA-256 Hash Algorithms.” [Online].

Available: https://www.thesslstore.com/blog/difference-sha-1-sha-2-sha-256-hash-algorithms/. [Accessed: 13-Nov-2019].