Cryptographic Hash Functions

Çetin Kaya Koç
koc@ece.orst.edu
Electrical & Computer Engineering
Oregon State University
Corvallis, Oregon 97331

Technical Report
December 9, 2002
Version 1.5

1 Introduction

Cryptographic hash functions play a fundamental role in modern cryptography. They are related to conventional hash functions commonly used in non-cryptographic computer applications, however, they differ in several aspects, for example, in terms of input and output data sizes and certain desired properties. Cryptographic hash functions are used in data integrity and message authentication.

A hash function takes a message as input and produces an output referred to as a hash-code, hash-result, hash-value, or hash. More precisely, a hash function H maps bitstrings of arbitrary finite length to strings of fixed length, say n bits. Therefore, they are many-to-one type of functions (many input values map to one output value).

We will use S^*, S^n, and S^m to denote bit strings of arbitrary length, length n, and length m, respectively. A hash function maps an arbitrary length bit string to a bit string of length n. Since in general input size is much larger than n, the function is many-to-one, implying that the existence of collisions (pairs of inputs with identical output) is unavoidable. Indeed, restricting H to a domain of t-bit inputs ($t > n$), we can see that if H were random in the sense that all outputs were essentially equiprobable, then about 2^{t-n} inputs would map to each single output, and two randomly chosen inputs would yield the same output with probability 2^{-n} (which is independent of t).

The basic idea of cryptographic hash functions is that a hash-value serves as a compact representative image (sometimes called a *digital fingerprint* or *message digest*) of an input string, and can be used as if it were uniquely identifiable with that string.

Hash functions are used for data integrity in conjunction with digital signature schemes, where for several reasons a message is typically hashed first, and then the hash-value, as a representative of the message, is signed in place of the original message. A distinct class of hash functions, called *message authentication codes* (MAC), allows message authentication using secret-key cryptographic techniques. MAC algorithms may be viewed as hash functions which take two functionally distinct inputs, a message and a secret key, and produce a fixed-size (say n-bit) output, with the design intent that it be infeasible in practice to produce the same output without knowledge of the key. A MAC algorithm can be used to provide data integrity and symmetric data origin authentication, as well as identification in secret-key schemes.

2 Usage of Hash Functions

A typical usage of hash functions for data integrity is as follows. The hashvalue corresponding to a particular message x is computed at time T1. The integrity of this hash-value (but not the message itself) is protected in some manner. At a subsequent time T2, the following test is carried out to determine whether the message has been altered, i.e., whether a message $x1$ is the same as the original message. The hash-value of $x1$ is computed and compared to the protected hash-value; if they are equal, one accepts that the inputs are also equal, and thus that the message has not been altered. The problem of preserving the integrity of a potentially large message is thus reduced to that of a small fixed-size hashvalue. Since the

existence of collisions is guaranteed in many-to-one mappings, the unique association between inputs and hash-values can, at best, be in the computational sense. A hash-value should be uniquely identifiable with a single input in practice, and collisions should be computationally difficult to find (essentially never occurring *in practice*).

3 Hash Functions and Compression Functions

A hash function H maps an element of S^* to S^n as follows

$$H \ : \ S^* \ \rightarrow \ S^n$$

For example, the xor function

$$H(b_1 b_2 b_3 \ldots) = b_1 \oplus b_2 \oplus b_3 \oplus \cdots$$

is a hash function with output 0 or 1, while the input size is arbitrary. A hash function is usually built upon a compression function G which maps an element of S^m to an element of S^n, where m is a fixed positive integer. For example,

$$G(b_1 b_2 b_3 \ldots b_m) = b_1 \oplus b_2 \oplus b_3 \oplus \cdots \oplus b_m$$

is a compression function with output 0 or 1, while the input can be any one of 2^m possible binary strings. In general, it is expected that $m > n$.

4 Properties of Hash and Compression Functions

In order to be cryptographically useful, a hash function H and its compression function G needs to have certain properties. These properties are described below.

Easy Computation: It should be easy to compute $H(x)$ given x. This property is important only from the point of time and storage area needed to compute the hash value

Preimage Resistance: For essentially all pre-specified outputs, it should be computationally infeasible to find any input which hashes to that output, i.e., to find any preimage x such that $H(x) = y$ when given y for which a corresponding input x is not known. Another terminology used for this property is that the hash function is *one-way hash function*.

Second-Preimage Resistance: It should be computationally infeasible to find any second input which has the same output as any specified input, i.e., given x, to find a second-preimage y such that $H(x) = H(y)$ however, $x \neq y$. Another terminology to describe this property is that the hash function is *weak collision resistant*.

Collision Resistance: It should be computationally infeasible to find any two distinct inputs x and y which hash to the same output, i.e., such that $H(x) = H(y)$. Note that here there is free choice of both inputs. Another terminology to describe this property is that the hash function is *strong collision resistant*.

Here, the terms easy and computationally infeasible (or hard) are understood according to the context. Easy might mean polynomial time and space; or more practically, within a certain number of machine operations or time units, perhaps seconds or milliseconds. A more specific definition of computationally infeasible might involve super-polynomial effort; require effort far exceeding understood resources; specify a lower bound on the number of operations or memory required in terms of a specified security parameter; or specify the probability that a property is violated be exponentially small.

There are certain relationships between the between the stated properties of hash functions. We will briefly describe them below,

- *Collision resistance implies second-preimage resistance.* If H has collision resistance, then it is computationally hard two find any two inputs x and y which hash to the same value $H(x) = H(y)$, therefore, it is at least as difficult (or more difficult) when x is fixed and we are trying to find a y.

- *Collision resistance does not guarantee preimage resistance.* There are some pathological examples of hash functions which are collision resistant, however, they are not one-way (they are not preimage resistant). However, in practice, we can safely say that *collision resistance indeed implies preimage-resistance.*

5 Example Hash Functions and Their Properties

5.1 Modulo-32 Checksum

A simple modulo-32 checksum (32-bit sum of all 32-bit words of a data string) is an easily computed function which offers compression, however, this hash function is not collision resistant. Let x_i be the ith word of the input data, assuming $i = 1, 2, \ldots, k$ (i.e., there k words, each of which is 32 bits). Then the final hash value z is computed using

$$z = x_1 \oplus x_2 \oplus \cdots \oplus x_k$$

Given z, we can first randomly choose a sequence of y_is for $i = 1, 2, \ldots, k-1$ such as

$$y_1 y_2 \cdots y_{k-1}$$

and then choose y_k as

$$y_k = z \oplus (y_1 \oplus y_2 \oplus \cdots \oplus y_{k-1})$$

It is easy to show the sequence $y_1 y_2 \cdots y_k$ has the same hashvalue as the sequence $x_1 x_2 \cdots x_k$ since

$$\begin{aligned} H(y_1 y_2 \cdots y_k) &= y_1 \oplus y_2 \oplus \cdots \oplus y_{k-1} \oplus y_k \\ &= (y_1 \oplus y_2 \oplus \cdots \oplus y_{k-1}) \oplus z \oplus (y_1 \oplus y_2 \oplus \cdots \oplus y_{k-1}) \\ &= z \end{aligned}$$

5.2 Modular Squaring

Another hash function example is the squaring of integers modulo a prime p, e.g.,

$$H(x) = x^2 \pmod{p}$$

which behaves in many ways like a random mapping. However, $H(x)$ is not one-way function because finding square roots modulo primes is easy (there is a polynomial algorithm for it). On the other hand,

$$H(x) = x^2 \pmod{n}$$

is a one-way function for appropriate randomly chosen primes p and q where $n = pq$ and the factorization of n is unknown, as finding a preimage (i.e., computing a square root mod n) is computationally equivalent to factoring and thus intractable. However, finding a second-preimage, and, therefore, collisions, is trivial: given x, $n - x$ yields a collision since

$$(n - x)^2 = x^2 \pmod{n}$$

5.3 Block Cipher

A one-way function can be constructed from DES or any block cipher E which behaves essentially as a random function, as follows:

$$H(x) = E_k(x) \oplus x$$

for any fixed known key k. The one-way nature of this construction can be proven under the assumption that E is a random permutation. An intuitive argument follows. For any choice of y, finding any x (and key k) such that $y = E_k(x) \oplus x$ is difficult because for any chosen x, $E_k(x)$ will be essentially random (for any key k) and thus $E_k(x) \oplus x$ will be random too. Therefore, for a given $y = H(x)$, where x is unknown, our chances of finding x is as good as guessing x at random

By similar reasoning, if one attempts to use decryption and chooses an x, the probability that $x = E_k^{-1}(x \oplus y)$ is not better than random chance. Thus $H(x)$ appears to be a one-way function. As it is, $H(x)$ can handle only fixed-length inputs, therefore, it is not a hash function (it is a compression function), but it can be extended to yield a hash function, i.e., by building a hash function on top of the compression function.

6 Attacks on Hash Functions

Given a specific hash function, it is desirable to prove a *lower bound* on the complexity of attacking it under specific scenarios. However, such lower bound proofs are very scarce and they depend on the mathematical properties of the hash function. Typically the best we can do is to obtain the complexity of an applicable known attack, which gives on *upper bound* on security.

An attack of complexity 2^t means that the attack requires 2^t operations or time units in order to be successful. Here, typically, an operation means the execution of the compression function of the hash function. The storage complexity of an attack should also be considered for realistic scenarios.

In the following we describe the known attacks on generic hash functions and their compression functions. We assume that the input size of the compression function is m, while the outputsize of the compression function and the hash function is n with $n < m$. These attacks are algorithm independent attacks which can be applied to any hash function, treating it as a block-box whose only significant property is its output size in bits, which is n.

Naive Attack: Given a fixed message x of length t bits, and a hash function H, a naive method of finding an input colliding with x is to exhaustively search all t-bit binary numbers y until one is found with $H(x) = H(y)$. The storage requirement is negligible since after an inequality, we discard $H(y)$ and start a new computation. Assuming the hashcode is a random value, the probability of a match for a single test is 2^{-n} (which is independent of t). This implies that a collision will be found much earlier than after exhaustively trying all 2^t values since $t > n$.

Birthday Attack: For an n-bit hash function (or compression function) $H(x)$ with input x, we may expect a guessing attack to find a preimage x or second-preimage y (with $H(x) = H(y)$) within 2^n hashing operations. This problem is related to the following observation, which is known as *birthday paradox*.

> When drawing randomly with replacements from a set of N elements, a repeated element will be encountered after approximately $\sqrt{N}$ selections with probability more than $1/2$.
>
> More accurately: If $r = \sqrt{2\lambda N}$ tries can be made on a hash function of N possible values, then a collision can be found with probability $(1 - e^{-\lambda})$.

For example, if we attack an n-bit hash or compression function, after approximately

$$\sqrt{2 \times 8 \times 2^n} = 4 \times 2^{\frac{n}{2}}$$

tries, we will obtain a collision with probability $1 - e^{-8} = .9997$, which is nearly 1.

Birthday Attacks with Storage: More sophisticated versions of the birthday attacks can devised by using a storage to save some of the hash values computed during the attack. In the following, we describe such an attack.

Input:	A legitimate x_1 and a fraudulent x_2 and n-bit hash function H
Output:	x_1' and x_2' which are obtained from x_1 and x_2 with minor modifications with $H(x_1') = H(x_2')$
Step 1.	Generate $t = 2^{n/2}$ minor modifications x_1' of x_1 This can be accomplished by selecting $n/2$ bit locations in x_1 and then complementing each bit for all $n/2$ bits, and obtaining $2^{n/2}$ distinct messages from x_1
Step 2.	Hash each such modified message, and store them in a table so that they can be searched according to the hashvalue This can be done in t operations and requires t entries in the table
Step 3.	Generate minor modifications from x_2' of x_2 and compute each $H(x_2')$ and check in the table for $H(x_2') = H(x_1')$ Continue until a match is found. Each table lookup requires constant time, and a match is expected within $t = 2^{m/2}$ tries

The above attack is a real, practical attack, and applicable to real-world situations. There are several other attacks, most of which are theoretical attacks based on random collisions in the compression function or the hash function, which may not mean much in terms of meaningful messages. In this attack, we are able to construct a fraudulent (but meaningful) message which hashes to another legitimate (and meaningful) message using essentially $2^{n/2}$ operations and a table of $2^{n/2}$ entries.

7 Current Hash Functions

In recent years, there has been considerable effort and some successes of attacks on hash functions. In order to withstand to these attacks, new hash function design methodologies have been developed. The most prevalent of these methods is the iterated hash functions based on compression functions. The hash algorithm H involves repeated use of its compression function G. The compression function G takes two inputs: an m-bit data block and n-bit *chaining variable* which comes from the previous step. At the start of the hash algorithm, the initial value of the chaining variable is set to a fixed value. The input M is broken into m-bit blocks $M_1 M_2 \cdots M_L$ and applied to the compression function together with the initial fixed value of the chaining variable C_0 as follows:

$$\begin{aligned} C_0 &= \text{Fixed initial } n\text{-bit value} \\ C_i &= G(C_{i-1}, M_i) \text{ for } i = 1, 2, \ldots, L \\ \text{Final hash value} &= C_L \end{aligned}$$

Therefore, the hash of the entire message M (which is of Lm bits) is the final value of the chaining variable, C_L. All previous chaining variable values are discarded.

The hash methods which have been widely used and standardized are MD5 and SHA-1. MD5 method was proposed by RSA Security, part of the PKCS (Public-Key Cryptography

Standards). MD5 is an iterated hash function with output hash value which is 128 bits. The input block length of MD5 is 512 bits.

On the other hand, SHA was proposed by NIST, together with the Digital Signature Standard [2]. MD5 and SHA are based on same principles, however, the output of SHA-1 is 160 bits. The input block length of SHA-1 is also 512 bits.

Applying the birthday attack on MD5, we see that if we make

$$r = \sqrt{2 \times 2^3 \times 2^{128}} = 2^{66}$$

tries, then, we will find a collision with probability $1 - e^{-8} \approx 1$. Similarly, a birthday attack on SHA-1 requires approximately

$$r = \sqrt{2 \times 2^3 \times 2^{160}} = 2^{82}$$

in order to be successful with 99.97 % probability.

8 Long Term Security of Hash Functions

We need an estimation of work needed to perform 2^{66} or 2^{82} tries, in terms of the budget and the amount of time needed to conclude such computation. This analysis was performed here [1] based on the current state of art in computing today and in the future, using a particular model of estimation for computing power. This model is based on Moore's Law, which is a prediction made by one of the founders of Intel, nearly 20 years ago, which remains true until now. Moore's Law states that *density of components per integrated circuit doubles every 18 months*. A more popular interpretation of the Law is that *computing power per computer doubles every 18 months*. From economics point of view , Moore's Law can be stated *as computing power and RAM which can be purchased per dollar doubles every 18 months*. The last interpretation gives a good idea about the cost of building and operating a special-purpose hash breaking machine versus the time needed to break the same hash on a single PC. The results are shown in the following table.

Year	Number of Tries	Budget for Attack in 1 day	Years on PII 450MHz
2002	2^{72}	$ 160 million	45 millions
2005	2^{74}	$ 196 million	226 millions
2010	2^{78}	$ 277 million	3 billions
2015	2^{82}	$ 392 million	45 billions
2020	2^{86}	$ 555 million	654 billions

This table is interpreted as follows: In year 2002, in order to perform 2^{72} tries in a single day one needs a budget of $ 160 million to build a special-purpose computer. Or, it will take low-budget hacker with a single 450-MHz Pentium II PC approximately 45 million years to perform 2^{72} tries. What this implies is that a government or a large company can perhaps break a hash function of 144 bits, but it is beyond the grasp of a low-budget hacker.

9 New Hash Functions

The above analysis assumes that the particular hash function is safe from mathematical analysis. As of now, no mathematical attack has been found to completely break either MD5 or SHA-1. However, the current US Government standard [2] does not include MD5 anymore due to the its size which is 128 bits. It is definitely breakable by a midsize company or a government since 2^{66} tries are now within grasp of our technology with a few million dollars of budget.

SHA-1 is relatively safer: it can be broken with a budget of $ 392 million in the year 2015 (using the technology estimated to be available in 2015). This is obviously not sufficiently safe to build an multibillion dollar business on top of it, or to trust the nation's infrastructure. These considerations led NIST to search for bigger and better hashes, and in May 2001, three new hash functions were proposed: SHA-256, SHA-384, and SHA-512. As the names suggest, the output sizes of these hash functions are 256, 384, an 512 bits. These methods are currently open for review. It should be noted that these methods are not direct generalizations of SHA, and they are based on some new methods and constructs. We will need thorough security analyses of these new hashes before standardization and deployment.

10 Safeback Hashing Method

The hashing method used in the Safeback product is based two compression functions F and G as follows:

$$\begin{aligned} F &: \text{output size} = 16 \text{ bits} \\ &\quad \text{input size} = 60 \times 1024 \text{ bits} \\ G &: \text{output size} = 32 \text{ bits} \\ &\quad \text{input size} = 60 \times 1024 \text{ bits} \end{aligned}$$

A large message M is broken into L blocks such that each block is 60×1024 bits:

$$\text{message} : M_1 M_2 \cdots M_L$$

Each block M_i is hashed independently using the compression function F to obtain the hash values A_i as $A_i = F(M_i)$. Therefore, the first group of hash values are

$$A_1 A_2 \cdots A_L \quad \text{with} \quad A_i = F(M_i)$$

These values are kept as the output hash values. Additionally, a single final hash value is produced using the chaining method via the compression function G with the help of a 32-bit chaining variable C_i as follows:

$$\begin{aligned} C_0 &= \text{Fixed initial 32-bit value} \\ C_i &= G(C_{i-1}, M_i) \text{ for } i = 1, 2, \ldots, L \\ \text{Final hash value} &= C_L \end{aligned}$$

The final total hash of the entire message $M_1 M_2 \cdots M_L$ is obtained as

$$A_1 A_2 \cdots A_L \; C_L$$

such that each of A_i is 16 bits and C_L is 32 bits.

11 Analysis of the Safeback Compression Functions

In the following, we analyze the Safeback hashing mechanism as a blackbox, i.e., without analyzing the mathematical properties of the underlying functions. We first start analyzing the compression functions, and show how to obtain collisions.

The compression functions F and G used in the Safeback has output sizes 16 and 32, which are far below the required security levels of the compression functions. As a comparison, the output sizes of the compression functions for MD5 and SHA-1 are 128 and 160 bits.

In order to find a collision in F, we need to perform about 2^8 to 2^{10} hashing operations, while finding a collision in G requires 2^{16} to 2^{18} hashing operations. Both of these tries are trivially accomplished on a PC in a few seconds. This analysis is a worst-case analysis, and does not involve the mathematical properties of the either compression function.

Also note that the fact that the input size is so large (60×1024 bits) makes our job of finding collisions easier not harder. Since input data is so large, we are more likely to find *meaningful* collisions since there are more bit locations to tamper with (by complementing).

- For F, the input size is 60×1024 and the output is only 16 bits. Therefore, the number of inputs which map to the same hash value is

  $$\frac{2^{60 \times 1024}}{2^{16}} = \frac{2^{61440}}{2^{16}} = 2^{61424}$$

  This is indeed a large number of possibilities. Let M_i is a data block for which we would like to find a collision on F. We locate 8 to 10 bit locations in M_i (among 61440 bit locations) and complement the input bits and obtain the modified messages M_i' and check to see if

  $$F(M_i) = F(M_i')$$

  The chance of finding a collision is nearly 1.

- For G, the input size is $32 + 60 \times 1024$ plus 32 since G takes two inputs: C_{i-1} (32-bit chaining variable) and M_i (the data block). The output of G is 32 bits. Therefore, the number of inputs which map to the same has value is

  $$\frac{2^{32+60 \times 1024}}{2^{32}} = \frac{2^{61472}}{2^{32}} = 2^{614440}$$

  This is also a large number of possibilities. Let M_i is a data block for which we would like to find a collision on G. We locate 16 to 18 bit locations in M_i (among 61440 bit

locations) and complement the input bits and obtain the modified messages M_i' and check to see if

$$G(C_{i-1}, M_i) = G(C_{i-1}, M_i')$$

The chance of finding a collision is nearly 1.

- We will also consider the composite function FG, which takes two inputs: a chaining value C_{i-1} (which is 32 bits) and a data block M_i (which is 60×1024 bits). The output of the composite function is the 48-bit value which is the concatenation of A_i and C_i

$$A_i | C_i = FG(C_{i-1}, M_i)$$

such that $A_i = F(M_i)$ and $C_i = G(C_{i-1}, M_i)$. The number of inputs which map to the same has value is

$$\frac{2^{32+60\times1024}}{2^{48}} = \frac{2^{61472}}{2^{48}} = 2^{61424}$$

Since the output is 48 bits, it will require 4×2^{24} hashing operations to find a collision for FG with probability 0.9997. Therefore, finding an M_i' for FG is only as difficult as performing about 64 million hashes on a PC, which should take on the order of several minutes.

12 Attack on the Safeback Hashing Method

Based on these observations we construct an attack on Safeback hashing method. We assume that a message which consists of L blocks (such that each block M_i is of length 60×1024 bits) is given and together with its hash values

$$\begin{array}{rccccc} \text{message:} & M_1 & M_2 & \cdots & M_L & \\ \text{hash:} & A_1 & A_2 & \cdots & A_L & C_L \end{array}$$

We will now construct a new message which differs from the original message only in its first block. However, both the original message and the changed will have the same hash values.

$$\begin{array}{rcccc} \text{original message:} & M_1 & M_2 & \cdots & M_L \\ \text{changed message:} & M_1' & M_2 & \cdots & M_L \end{array}$$

Using the attack on F, we can easily find M_1' such that $A_1 = F(M_1) = F(M_1')$, while $M_1 \neq M_1'$. Since the other message blocks are not affected, their A_i values will remain the same.

$$\begin{array}{rccccc} \text{message:} & M_1' & M_2 & \cdots & M_L & \\ \text{hash:} & A_1 & A_2 & \cdots & A_L & C_L' \end{array}$$

However, this change in M_1' will possibly change the final hash value computed by the compression function G. Therefore, the question is whether we can find an M_1' that will

keep both A_1 and final C_L unchanged. In order to accomplish this, we need to closely examine the temporary values of the chaining variable C_i.

$$\begin{array}{rcccc} \text{message:} & M_1 & M_2 & \cdots & M_L \\ \text{hash:} & A_1 & A_2 & \cdots & A_L \\ \text{chain:} & C_1 & C_2 & \cdots & C_L \end{array}$$

Note that we have the relationships

$$\begin{aligned} C_0 &= \text{Fixed initial value} \\ C_1 &= G(C_0, M_1) \\ C_2 &= G(C_1, M_2) \\ &\vdots \\ C_L &= G(C_{L-1}, M_L) \end{aligned}$$

Since C_0 is the same for all data blocks, and we find a collision for M_1' such that both of these conditions hold

$$\begin{aligned} A_1 &= F(M_1) = F(M_1') \\ C_1 &= G(C_0, M_1) = G(C_0, M_1') \end{aligned}$$

This implies that we find a collision for the composite function FG

$$A_1|C_1 = FG(C_0, M_1) = FG(C_0, M_1')$$

As we have examined before in Section 11, finding collisions for the composite function is FG is a little more work than finding collisions in either F or G, however, it can be accomplished within several minutes on a PC.

Once we find M_1' which keeps the hash and chain values A_1 and C_1 the same, then the remaining hash and chain values will be unchanged:

$$\begin{aligned} C_0 &= \text{Fixed initial value (same)} \\ A_1|C_1 &= FG(C_0, M_1) = FG(C_0, M_1') \\ A_2|C_2 &= FG(C_1, M_2) \\ &\vdots \\ A_L|C_L &= FG(C_{L-1}, M_L) \end{aligned}$$

Therefore, the hash values of the original and changed message will be the same.

This attack can be generalized by finding a collision for any block of the message, rather than the first block. Assuming we want to construct a message which differs from the original message in the kth block.

$$\begin{array}{rccccccc} \text{original message:} & M_1 & \cdots & M_{k-1} & M_k & M_{k+1} & \cdots & M_L \\ \text{changed message:} & M_1 & \cdots & M_{k-1} & M_k' & M_{k+1} & \cdots & M_L \end{array}$$

First we notice that the hash and chain values until the $(k-1)$st block will be the same since we have not changed the message blocks $M_1 M_2 \cdots M_{k-1}$.

$$\begin{aligned} C_0 &= \text{Fixed initial value (same)} \\ A_1|C_1 &= FG(C_0, M_1) \\ A_2|C_2 &= FG(C_1, M_1) \\ &\vdots \\ A_{k-1}|C_{k-1} &= FG(C_{k-2}, M_{k-1}) \end{aligned}$$

Therefore, we need to find M'_k which is different from M_k, however, it collides with M_k on the composite function FG.

$$A_k|C_k = FG(C_{k-1}, M_k) = FG(C_{k-1}, M'_k)$$

Find such a collision is the same amount work as finding a collision for M_1. Since A_k and C_k are now unchanged (due to collision property), the remaining hash and chain values will remain the same as well

$$\begin{aligned} A_{k+1}|C_{k+1} &= FG(C_k, M_{k+1}) \\ A_{k+2}|C_{k+2} &= FG(C_{k+1}, M_{k+1}) \\ &\vdots \\ A_L|C_L &= FG(C_{L-1}, M_L) \end{aligned}$$

13 Conclusions

Hash functions are cryptographic mechanisms for the purpose of checking the integrity data blocks, files, messages, etc. The state of the art hash functions are iterative hash functions with output size at least 128 bits, and the US Government standards suggest output sizes 160 bits up to 512 bits. The input size is unlimited, however, the data is broken into blocks of fixed length, which is usually 512 bits. Making the input block size very small (less than 128) or very large (more than 1024 bits) weakens the hashing mechanism.

References

[1] A. K. Lenstra and E. R. Verheul. Selecting cryptographic key sizes. *Journal of Cryptology*, 14(4):255–293, 2001.

[2] National Institute for Standards and Technology. Digital Signature Standard (DSS), January 2000. FIPS 186-2.