Chapter 1

On the Secure Hash Algorithm family

Written by *Wouter Penard, Tim van Werkhoven.*

1.1 Introduction

This report is on the Secure Hash Algorithm family, better known as the SHA hash functions. We will try to analyse how these functions work, focussing especially on the SHA-1 hash function. After dissecting the SHA-1 hash function itself, we will relate and compare it to the other hash functions in the SHA family, SHA-0 and SHA-2.

Besides analysing the specific SHA functions, we will also treat the generic structure of hash functions, as these are relevant for the security of a hash function. We will treat the different aspects in some detail and illustrate how naive implementation of hash functions can be hazardous for the security, some attacks can be mounted against any hash function if they are not implemented correctly.

The main part of this report will focus on the alleged weaknesses found in the SHA-0 hash function by different scientists. We first explain how the SHA-0 function was initially broken by treating the so-called differential collision search against this function. After explaining how the idea works, we will explain why this attack is less successful against the SHA-0 successor, SHA-1. Finally we will look at SHA-2 and summarize the cryptanalysis results for the SHA family.

The last part will deal with the implications of broken hash functions, and how dangerous these actually are. Although the weaknesses found may not seem all too severe, these attacks can indeed have serious consequences, as we will see in the final section.

1.1.1 Purpose of Cryptographic Hash Algorithms

A hash function is a function which takes an arbitrary length input and produces a fixed length 'fingerprint' string. Common usage of such a function is as index into

a hashtable. Cryptographic hash functions have several additional properties which makes them suitable to use as a means to check the integrity of a message and as part of digital signature schemes. Such a scheme consists of a secret key k_s, a public key k_p and two functions $Sign(M, k_s)$, which produces signature S, and $Verify(M, S, k_p)$, which returns a boolean indicating if the given S is a valid signature for message M. A function requirement is that $Sign(M, Sign(M, k_s), k_p) = true$ for a key pair (k_s, k_p). On the other hand it should be impossible to create a forged signature. It is possible to distinguish between two types of forgeries, *Existential* and *Universal* forgeries. In the first case the attacker creates a valid M, S pair, given the public key k_p. Note that the attacker cannot influence the computed message, and thus, in general, the produced M will be a random string. To create a universal forgery the attacker computes a valid signature S given M and k_p.

A public-private key cryptosystem, like for example RSA [9], can be used to place such a signature. Here secret key (n, d) is used to sign the message and the public key (n, e) is used to verify the signature. In order to create a universal forgery it would be required to effectively compute the private component of the RSA key system, which is assumed to be infeasible. However finding an existential forgery is trivial, for a random S we can find a corresponding message by calculating $M = S^e \,\%\, n$. Another drawback is that RSA can only sign messages with limited length, a straightforward, but bad solution would be to split the message in blocks and sign each block individually. However now it is possible for an attacker to rearrange the individual blocks, which would result in a new message with a valid signature. Finally RSA is a relatively slow algorithms.

Cryptographic hash functions can be used to resolve these problems. As mentioned earlier such a hash function, H, takes an arbitrary length message as input and produces a fixed length message digest D. Now we compute the message digest for a message and sign this digest instead of the actual message. To create a existential forgery it is required to find message M, given D such that $H(M) = D$.

1.1.2 Applications

1.1.3 Short History

The Secure Hash Algorithm (SHA) was developed by the NIST in association with the NSA and first published in May 1993 as the Secure Hash Standard. The first revision to this algorithm was published in 1995 due to a unpublished flaw found, and was called SHA-1. The first version, later dubbed SHA-0, was withdrawn by the NSA. The SHA hash function is similar to the MD4 hash function, but adds some complexity to the algorithm and the block size used was changed. SHA was originally intended as part of the Digital Signature Standard (DSS), a scheme used for signing data and needed a hash function to do so.

In 1998, two French researchers first found a collision on SHA-0 with complexity 2^{69} [1], as opposed to the brute force complexity of 2^{80} [1]. This result was improved in

[1] As the reader might recall, complexity is defined as the number of hash operations needed for a specific attack.

drastically improved by Wang e.a.which could find a collision with complexity 2^{39} [13], which is within practical reach. It took only five years for the initial SHA function to be broken, and after another seven years, the best attack possible is only half of the (logarithmic) complexity of the original hash function. Fortunately, the NSA already foresaw this in 1995 and released SHA-1.

Cryptanalysis on SHA-1 proved to be much more difficult, as the full 80-round algorithm was broken only in 2005, and this attack still has a complexity of 2^{63}[12]. This restricts the attack only to the theoretical domain, as a complexity of 2^{63} is still unfeasible on present-day computers. Nonetheless, this collisional attack requires less than the 2^{80} computations needed for a brute-force attack on SHA-1. We will explain why it took 7 years longer to break SHA-1 in section 1.4.2.

In addition to the SHA-1 hash, the NIST also published a set of more complex hash functions for which the output ranges from 224 bit to 512 bit. These hash algorithms, called SHA-224, SHA-256, SHA-384 and SHA-512 (sometimes referred to as SHA-2) are more complex because of the added non-linear functions to the compression function. As of January 2008, there are no attacks known better than a brute force attack. Nonetheless, since the design still shows significant similarity with the SHA-1 hash algorithms, it is not unlikely that these will be found in the (near) future. Because of this, an open competition for the SHA-3 hash function was announced on November 2, 2007[2]. The new standard is scheduled to be published in 2012. Fortunately, the SHA-2 hash functions produce longer hashes, making a feasible attack more difficult. Consider for example the SHA-512 hash function, producing 512 bit hashes and thus having an approximate complexity against collisional attacks of 2^{256}. Even if the logarithmic complexity would be halved (from 256 to 128), this would still be out of reach for practical purposes for the coming decade or so.

1.2 Description of the SHA Algorithms

In this section we will describe how the SHA-1 algorithm works and relate it to both its predecessor, SHA-0 and its successor SHA-2. In each of the algorithms we identify two phases: first message expansion followed by a state update transformation which is iterated for a number, 80 in SHA-1, of rounds. In the next sections we make use of the following operators: $\ll$ and $\gg$, the left and right shift operator, and $\lll$ and $\ggg$ the bitwise left- and right rotate operator.

1.2.1 SHA-1

SHA-1 takes as input a message with maximum length $2^{64} - 1$ bits and returns a 160-bit message digest. The input is processed in parts of 512 bit each, and is padded using the following scheme. First a 1 is appended and then padded with 0's until bit 448, finally the length of the message is inserted in the last 64-bits of the message, the most significant bits padded with 0's. The reason that first a 1 is appended is that otherwise

[2] `http://csrc.nist.gov/groups/ST/hash/documents/FR_Notice_Nov07.pdf`

collisions occur between a sample messages and the same message with some zeros at the end.

The intermediate results of each block are stored in five 32-bit registers denoted with $h_0, \ldots, h_5$. These five registers are initialized with the following hexadecimal values:

$H_0 =$ `0x67452301`
$H_1 =$ `0xEFCDAB89`
$H_2 =$ `0x98BADCFE`
$H_3 =$ `0x10325476`
$H_4 =$ `0xC3D2E1F0`

Next, the algorithm uses four constants K0, K1, K2 and K3, with values:

K0 = `0x5A827999`
K1 = `0x6ED9EBA1`
K2 = `0x8F1BBCDC`
K3 = `0xCA62C1D6`

Finally we define four functions: f_{exp}, f_{if}, f_{maj} and f_{xor}. The first takes the 512-bit message as argument, each of the other functions take three 32-bit words (b, c, d) as argument.

The f_{exp} function expands the initial 512-bit input message M, consisting of 16 32-bit words M_i with $0 \leq i \leq 15$ to 80 32-bit words W_i with $0 \leq i \leq 79$.

$$W_i = \begin{cases} M_i, & \text{if } 0 \leq i \leq 15 \\ W_{i-3} \oplus W_{i-8} \oplus W_{i-14} \oplus W_{i-16} \lll 1, & \text{if } 16 \leq i \leq 79 \end{cases}$$

The other functions are defined as:

$$\begin{aligned} f_{if}(b,c,d) &= b \wedge c \oplus \neg b \wedge d \\ f_{maj}(b,c,d) &= b \wedge c \oplus b \wedge d \oplus c \wedge d \\ f_{xor}(b,c,d) &= b \oplus c \oplus d \end{aligned}$$

The SHA-1 round is graphically depicted in Figure 1.2.

1.2.2 SHA-0

The design proposal for SHA-0 was retracted by the NSA shortly after publishing it. The only difference between SHA-0 and SHA-1 is in the message expansion phase.

$$W_i = \begin{cases} M_i, & \text{if } 0 \leq i \leq 15 \\ W_{i-3} \oplus W_{i-8} \oplus W_{i-14} \oplus W_{i-16}, & \text{if } 16 \leq i \leq 79 \end{cases} \tag{1.1}$$

Note the absence of the leftrotate operation.

1.2.3 SHA-2

SHA-2 is a common name for four additional hash functions also referred to as SHA-224, SHA-256, SHA-384 and SHA-512. Their suffix originates from the bit length of the message digest they produce. The versions with length 224 and 384 are obtained by

SHA-0(M):
(* Let M be the message to be hashed *)
for each 512-bit block B in M **do**
$W = f_{exp}(B)$;
(* Initialize the registers with the constants. *)
$a = H_0$; $b = H_1$; $c = H_2$; $d = H_3$; $e = H_4$;
for $i = 0$ **to** 79 **do**
(* Apply the 80 rounds of mixing. *)
if $0 \leq i \leq 19$ **then**
$T = a \lll 5 + f_{if}(b, c, d) + e + W_i$+K0;
else if $20 \leq i \leq 39$ **then**
$T = a \lll 5 + f_{xor}(b, c, d) + e + W_i$+K1;
else if $40 \leq i \leq 59$ **then**
$T = a \lll 5 + f_{maj}(b, c, d) + e + W_i$+K2;
else if $60 \leq i \leq 79$ **then**
$T = a \lll 5 + f_{xor}(b, c, d) + e + W_i$+K3
$e = d$; $d = c$; $c = b \lll 30$; $b = a$; $a = T$;
(* After all the rounds, save the values in preparation of the next data block. *)
$H_0 = a + H_0$; $H_1 = b + H_1$; $H_2 = c + H_2$; $H_3 = d + H_3$; $H_4 = e + H_4$;
(* After all 512-bit blocks have been processed, return the hash. *)
return concat(H_0, H_1, H_2, H_3, H_4);

Algorithm 1.1: The SHA-1 algorithm.

truncating the result from SHA-256 and SHA-512 respectively. SHA-256 uses a block size of 512 bits, and iterates 64 rounds, while SHA-512 uses a 1024 bit block size and has 80 rounds. Furthermore, SHA-512 uses an internal word size of 64 bits instead of the 32 bit used by all other SHA variants. The SHA-2 algorithms follow the same structure of message expansion and iterated state update transformation as SHA-1, but both message expansion and state update transformation are much more complex. We will discuss SHA-256 in more detail to indicate differences between SHA-1 and SHA-2.

SHA-256 uses sixty-four constants $K_0, .., K_{63}$ of 32 bits each and eight registers to store intermediate results H_0, .., H_7. Their values can be found in [6]. The function definitions for SHA-256 are:

$$W_i = \begin{cases} M_i, & \text{if } 0 \leq i \leq 15 \\ \sigma_1(W_{i-2}) + W_{i-7} + \sigma_0(W_{i-15}) + W_{i-16}, & \text{if } 16 \leq i \leq 63 \end{cases} \tag{1.2}$$

with

$$\begin{aligned} \Sigma_0(x) &= x \ggg 2 \oplus x \ggg 13 \oplus x \ggg 22 \\ \Sigma_1(x) &= x \ggg 6 \oplus x \ggg 11 \oplus x \ggg 25 \\ \sigma_0(x) &= x \ggg 7 \oplus x \ggg 18 \oplus x \gg 3 \\ \sigma_1(x) &= x \ggg 17 \oplus x \ggg 19 \oplus x \gg 20 \end{aligned}$$

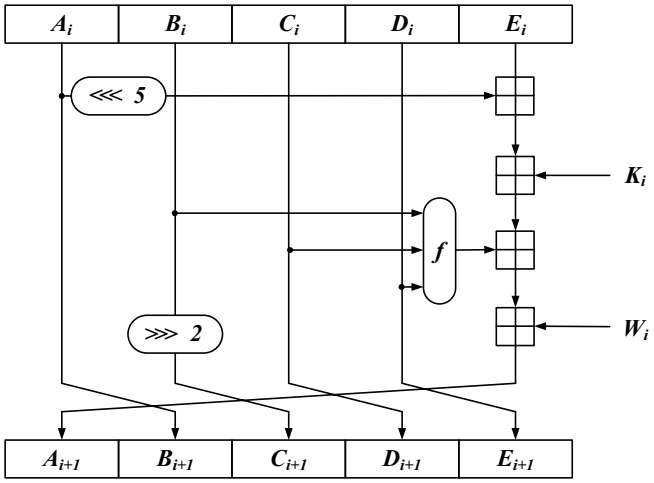


Figure 1.2: Schematic overview of a SHA-0/SHA-1 round.

and

$$\begin{aligned} f_{if}(b,c,d) &= b \wedge c \oplus \neg b \wedge d \\ f_{maj}(b,c,d) &= b \wedge c \oplus b \wedge d \oplus c \wedge d \end{aligned}$$

A round of the stronger SHA-2 hash function is shown in Figure 1.4.

SHA-256(M):
(* Let M be the message to be hashed *)
for each 512-bit block B in M **do**
$W = f_{exp}(B)$;
(* Initialize the registers with the constants. *)
$a = H_0$; $b = H_1$; $c = H_2$; $d = H_3$; $e = H_4$; $f = H_5$; $g = H_6$; $h = H_7$;
for $i = 0$ **to** 63 **do**
(* Apply the 64 rounds of mixing. *)
$T_1 = h + \Sigma_1(e) + f_{if}(e,f,g) + K_i + W_i$;
$T_2 = \Sigma_0(a) + f_{maj}(a,b,c)$;
$h = g$; $g = f$; $f = e$; $e = d + T_1$; $d = c$; $c = b$; $b = a$; $a = T_1 + T_2$;
(* After all the rounds, save the values in preparation of the next data block. *)
$H_0 = a + H_0$; $H_1 = b + H_1$; $H_2 = c + H_2$; $H_3 = d + H_3$;
$H_4 = e + H_4$; $H_5 = e + H_5$; $H_6 = e + H_6$; $H_7 = e + H_7$;
(* After all 512-bit blocks have been processed, return the hash. *)
return concat(H_0, H_1, H_2, H_3, H_4, H_5, H_6, H_7);

Algorithm 1.3: The SHA-256 algorithm.

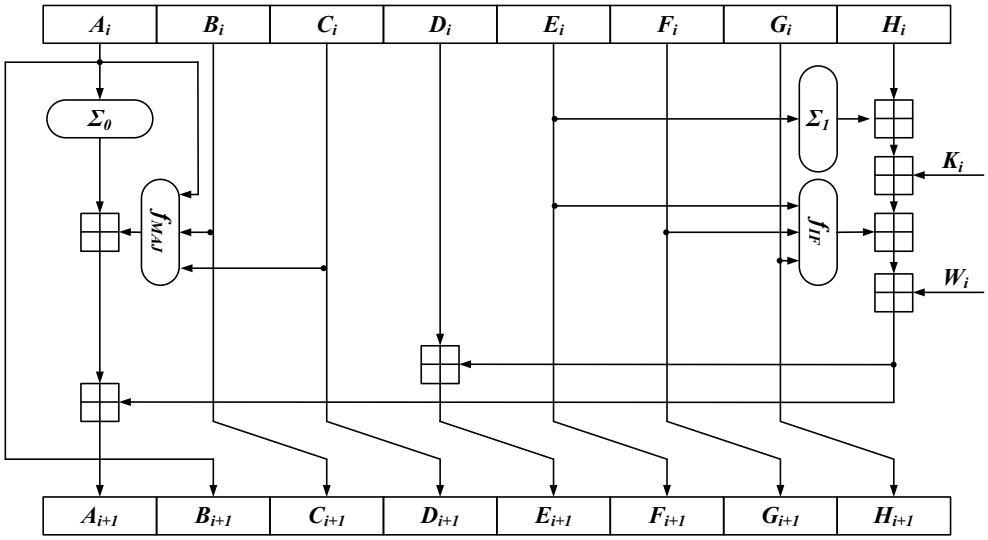


Figure 1.4: Schematic oveview of a 0 SHA-2 round. Note the added non-linear functions in comparison with SHA-1.

1.3 Generic Attacks

1.3.1 Brute Force Attacks

When considering attacks on hash functions, there are several different attacks possible, with varying difficulty. We define several aspects of a hash function below (see also [4]).

Definition 1.1 *Hash function H is* one-way *if, for random key k and an n-bit string w, it is hard for the attacker presented with k, w to find x so that $H_k(x) = w$.*

Definition 1.2 *Hash function H is* second-preimage resistant *if it is hard for the attacker presented with a random key k and random string x to find $y = x$ so that $H_k(x) = H_k(y)$.*

Definition 1.3 *Hash function H is* collision resistant *if it is hard for the attacker presented with a random key k to find x and $y = x$ such that $H_k(x) = H_k(y)$.*

It is clear that the last definition implies the second definition, if it is hard to find a collision between two chosen strings x and y, it is even harder to find an x given a y with the same hash. It is in general more difficult to find a relation between one-wayness of a hash function and the collisional resistance. If a function is one-way, it does not mean it is difficult to find a collision. For example take a function which takes a string of arbitrary length and returns the last 10 characters of this string. Clearly, from these ten characters, it is impossible to reconstruct the input string (if this string was longer than 10 characters), but it is also easy to see that collisions can be generated without any trouble.

Now if we want to attack a hash function on the second definition, the (second-)preimages attack, the best method that works on any hash function (i.e. a generic attack) is an exhaustive search. Given a hash function H_k, i.e. given w, k find x such that $H_k(x) = w$ (with k l-bit and w n-bit), would on average take 2^{n-1} operations (yielding a 50% chance of finding a preimage for w). If we are dealing with a random H and treat it as a black box, a preimage attack is as difficult as a second-preimage attack. This means that knowing that y with $H_k(y) = w$ does not give an advantage for finding x such that $H_k(x) = H_k(y) = w$.

On the other hand, if we are looking for a collision for a hash function H, things are a lot easier. Because of the "birthday paradox" the complexity of such a problem is only about $2^{n/2}$. Given 23 random people in a room, the chance that there are two people with the same birthday is a little over 50%, much higher than intuition would suggest, hence the name. A simple explanation is that the chance two people do not share their birthdays is 364/365. Now given p people the chance that two people share a birthday is given by

$$P = 1 - \prod_{k=1}^{p-1} \left(1 - \frac{k}{365}\right) \quad \text{for} p < 366 \tag{1.3}$$

and for $p = 23$ this yields a chance of 50.7%. Of course for $p \geq 366$ the chance is exactly 1.

The birthday paradox can be used on cryptographic hashes because the output of a hash function can be considered (pseudo-)random. Therefore if the hash is N bits long, there are 2^N hashes, but after trying only a fraction of that, we have a high chance for a collision. If we generalize 1.3 to range 2^N instead of 365, and count trials with t, we get the following expression

$$P(t; 2^N) = 1 - \prod_{k=1}^{t-1} \left(1 - \frac{k}{2^N}\right) \quad \text{for} t < 2^N \tag{1.4}$$

Now equating 1.4 to 0.5 results in the following expression

$$\begin{aligned} t &\approx \frac{1}{2} + \sqrt{\frac{1}{4} + 2 \cdot 2^N \cdot \ln 2} \\ &\approx 1.172^{N/2} \end{aligned} \tag{1.5}$$

i.e. the result is that we only have to try about $2^{N/2}$ hashes before finding a collision. This means that finding a collision is much simpler than a preimage attack. Again using the birthday illustration, finding two people with the same birthday is relatively easy, but finding someone else who shares *your* birthday is much harder.

1.3.2 Structure of hash functions

To understand the impact of the different attacks possible, we will elaborate a bit on the structure of hash functions. Most hash functions consist of two parts, one part being the *compression function*, which takes a key (k) and some fixed-length input message

(x) and outputs a hash. The second part is called the *domain extender* and links several compression functions together in such a way that the total hash function can operate on an input string of arbitrary length.

The compression function usually works on blocks on data, much like DES or AES do. The compression function takes a piece of data n bits long and runs several rounds of mixing on it. Before the data is hashed, it is padded so that the total length is an integer multiple of the block size, as explained in section 1.2.1. Usually, the mixing is done with the data itself as well as with some key of length k. The output is then a hash of these two input vectors and hopefully, the input vectors are not recoverable from the output hash. If the compression function is not invertible and calculating collisions is not easier than the birthday attack, then the compression function is strong and usually the resulting hash function is too.

Before the data block is used in the (non-linear) functions, it is sometimes expanded. In SHA, for example, the 512 bit input block (16 32-bit words) is expanded to 80 32-bit words (2560 bits). Then in each of the 80 rounds, a different part of the expanded message is used. This makes some kinds of attacks more difficult on the hash function, as we shall see in 1.4.1. In fact, the only difference between SHA-0 and SHA-1 is the message expansion, which makes the difference between a complexity of 2^{39} for SHA-0 versus 2^{69} for SHA-1.

In many hash functions, including SHA, the key used in the first round is some initialisation vector. After the first round, the output of the hash function is used as key for the next rounds. This construction is part of the Merkle-Damgård scheme, explained below.

A domain extender that is used a lot, and also in the SHA family, is the Merkle-Damgård domain extender, which works as follows. The compression function is used on an initialisation vector (IV) used as key and the first block of the input message. The output hash is then used as key for the next iteration where the second block of the data is used as input. In this way, a compression function can be used on a message of any length. Figure 1.5 shows the Merkle-Damgård scheme in action.

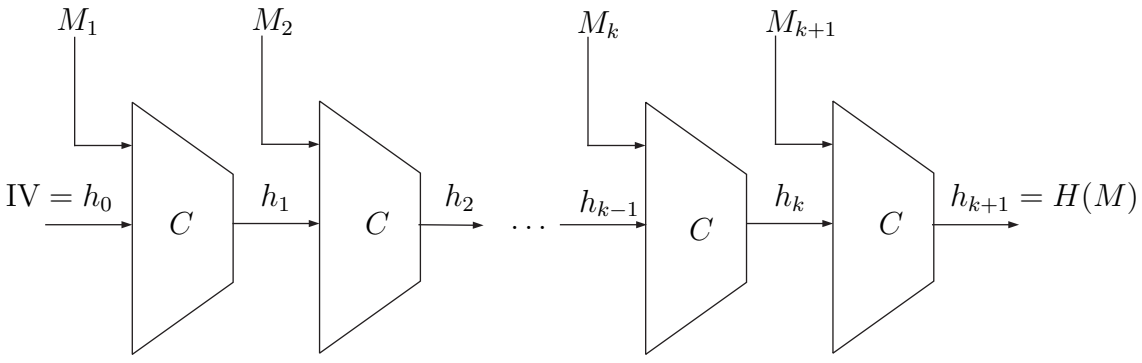


Figure 1.5: The Merkle-Damgård domain exten

1.3.3 Domain Extender Attacks

There are however several attacks possible against this domain extender, regardless of the hash function used. It should be noted that the domain extender does *not* add security to the hash scheme. A requirement of a secure hash scheme is that the compression function is safe (i.e. non-invertible and strong collision resistant), only then is the combination of compression function with domain extender secure.

MAC Weaknesses

An example of the way the Merkle-Damgård scheme can be attacked is the following. Consider the signing of a message. If we have a hash function H which works like $H_k(M) = t$ with k the key, M the message and t the signature. If we now would sign our message, and we would protect the key, one would assume that signing M would be be secure in this way.

However, if the hash function uses the domain extender explained above (like SHA-1), this is not true. Let k be 80 bits long, and M 256 bits. Consider now the scenario where the adversary captures M and the 512 bit signature t. The tag captured is equal to

$$t = \text{SHA-1}(k||M) = C(k||M||\underbrace{100\ldots0}_{112\text{ bits}}||\underbrace{0\ldots101010000}_{64\text{ bits}}), \tag{1.6}$$

with $C()$ the compression function used in SHA-1 and $||$ the concatenation operator. If we now want to forge a message with the same signature t, we can easily construct a new message M' which is also 'signed' with k, without knowing it. Consider the message $M' = k||M||100\ldots101010000||T$, with T arbitrary text we can choose to our liking. Now when we hash this using SHA-1, this results in:

$$\begin{aligned} t' &= \text{SHA-1}(k||M||100\ldots101010000||T||\text{padding}) \\ &= C(C(k||M||100\ldots0||0\ldots101010000), T||\text{padding}) \qquad (1.7) \\ &= C(t, T||\text{padding}) \qquad (1.8) \end{aligned}$$

This hashing calls the compression function twice because the message length exceeds the block length, and therefore the domain extender is employed. However, if we look at the first iteration of the compression function, it is exactly equal to t! Since this is known we can feed this to the compression function in the second round of the domain extender as key, and then hash our own message T with that, and we have a message signed with key k. Clearly, this method of signing messages is completely broken.

Poisoned Block Attack

Another way to attack hash functions is with the so called 'poisoned block attack'. The idea is to find a collision using the relatively fast birthday attack described above. Once such a pair of colliding messages are found, embed these in a bigger file which has some way to hide this data. Let some party sign one of the messages, and use this signature with the other message which the signing party has never seen.

To better explain this attack, consider a colliding pair of messages N and N'. Now consider the two messages T and T' which are constructed as follows:

$$\begin{aligned} T &= M_1||M_2||\ldots||N||\ldots||M_n \\ T' &= M_1||M_2||\ldots||N'||\ldots||M_n, \end{aligned}$$

with M_i a block of data. If we now let a trusted party sign T, this signature will also work for T', since N and N' collide. Now this may not seem very useful, but if we use the conditional possibilities of for example the Postscript format, we can influence the message shown depending on whether we embed N or N':

$$(R1)\ (R2)\ \texttt{eq}\ \{\text{instruction set 1}\}\ \{\text{instruction set 2}\}\ \texttt{ifelse}. \tag{1.9}$$

If $R1$ and $R2$ are equal, Postscript executes the first instruction set, and the second otherwise. Now if we use the poisoned blocks we can choose between these instruction sets, and thus influence the output of the Postscript document. This attack is not only theoretical, but has been carried out in practice by Magnus Daum and Stefan Lucks in 2005. They constructed two files with the same MD5 hash [2]. In 2006, Stevens, Lenstra and de Weger constructed two colliding X.509 certificates [10].

The implication of the poisoned block attack is obvious. If you can find a collision for a hash function using the Merkle-Damgård scheme, you can embed these blocks in bigger files. When using some higher level formatting language (Postscript or another language with conditionals), this block can be used to determine the flow of the formatting, branching into two different outputs. The bottom line is: don't use broken hash functions.

1.4 Specialized attacks

Besides generic attacks which work on all hash functions, or all hash functions using some domain extender, there are also attacks which target a single hash function. One can then refine the attack by looking at the specific structure of the studied hash function. These attacks have the disadvantage that they only work on one hash function, but are usually much faster. In 1998, such an attack was launched successfully against SHA-0 and it was broken.

1.4.1 Attacks Against SHA-0

The SHA-0 hash function is 160 bits long, which means that a birthday attack needs about 2^{80} hash operations to be successful. In 1998 this hash function was broken by the two French researchers Chabaud and Joux[1]. In this section, their approach is explained.

The method Chabaud and Joux used on SHA-0 can be summarized as follows:

- Linearize the hash function

- Find collisions for the simplified version
- From these collisions, find collisions for the real hash function

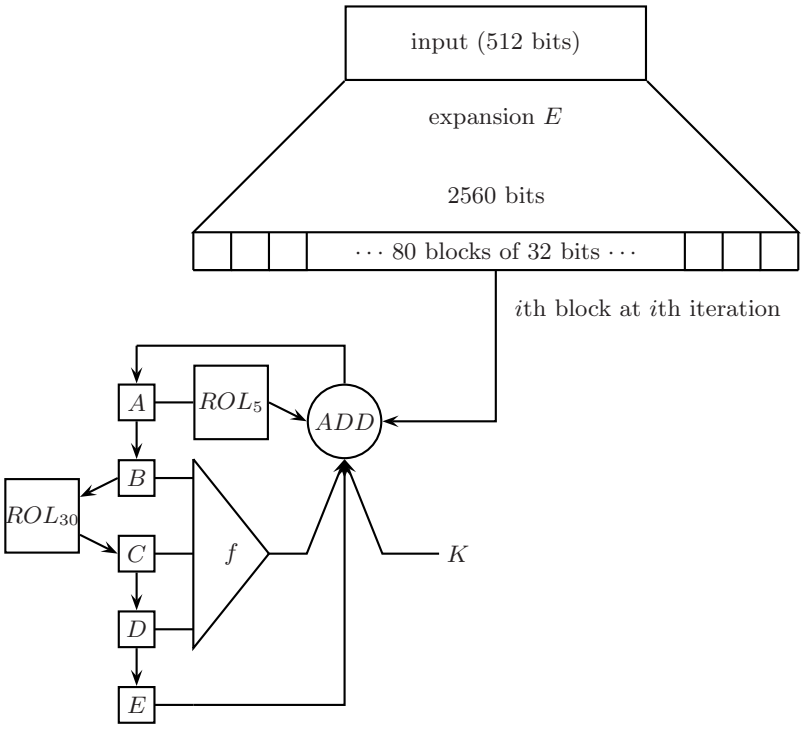


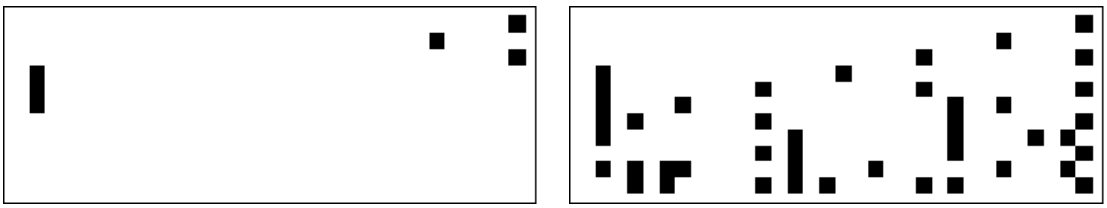
Figure 1.6: One round of the SHA-0 algorithm.

The linearization means that the non-linear functions are replaced by linear onces for simplicity. In the Chabaud and Joux analysis, the f_{if}, f_{maj} and ADD functions are replaced by the linear $\oplus$ function. In addition, the expansion function f_{exp} is ignored for the time being. This simplified hash function is called SHI-1. Now the trick is to look at the effect of a disturbance inserted in the expanded message. Recall that the input message block was 512 bit wide, which are 16 32-bit words. The expansion function expands this to 80 32-bit words. Now call $W^{(i)}$ the ith 32-bit word resulting from the expansion. Let $W_j^{(i)}$ be the jth bit in the ith word, with $0 \leq i \leq 79$ and $0 \leq j \leq 31$. For example $W_0^{(0)}$ would be the first bit in the first word of the expansion process.

Using these definitions, what happens if we negate bit $W_0^{(i)}$? This change will negate bit 1 in $A^{(i+1)}$, i.e. the same bit A, albeit a round later. The next change is in bit 1 of $B^{(i+2)}$ and then bit 31 is changed in $C^{(i+3)}$ (because of the rotation over 30 bits to the left, $\lll 30$). Bit 31 is also changed in $D^{(i+4)}$ and $E^{(i+5)}$. If we want to prevent further changes, we need to change some bits on the expanded input message, W. The bits we need to negate are then $W_6^{(i+1)}$ to counter the effect of the change in $A^{(i+1)}$ (note the $\lll 5$), $W_1^{(i+2)}$ for $B^{(i+2)}$, $W_31^{(i+3)}$ for $C^{(i+3)}$, $W_31^{(i+4)}$ for $C^{(i+3)}$ and $W_31^{(i+5)}$ for $C^{(i+3)}$. If we change these values in W we can prevent further change in the hash function and we have a local collision.

Now for SHI-1, we can construct two colliding W's. First we choose any (random) W, and after that we construct a second W'. Now for every bit negated in W and W', we apply the correction mechanism to W'. After we apply this correction over W', these two expanded messages have the same SHI-1 has. Because for a change in round i we need to change bits up to round $i+5$, this means that we cannot have differences in the last five rounds, because these cannot be corrected.

Below are some graphics illustrating the above mechanism. We start with a change in the first bit of the first round (located at the top right). After applying the corrections on W, we get the result shown in Fig. 1.7(a) for the first ten 32-bit words of W. If we do not apply the correction to the input message W, we get the error propagation shown in Fig. 1.7(b). Displayed in that figure is the state of A for the first ten rounds. If one looks closely, the pattern shown in Fig. 1.7(a) is visible, and in addition there is the result of not correcting the error in time, it obviously gets out of hand.



(a) (b)

Figure 1.7: (a) The mask necessary for the correction of the negation of bit 1 in round 1. (b) The result of negating bit 1 in round 1 if no correction is applied for SHI-1.

Now we know how to construct colliding W's for SHI-1 using a mask, but this is actually the result of the expansion process, and thus not directly under control. Only the first 16 32-bit words can be influenced directly, and the rest is generated in accordance with formula (1.1). Fortunately, this function is linear in the sense that the bits do not mix. Thus we want to go back from a mask which we can apply over W to a (much shorter) mask which we can apply over M. This M will then be a colliding message for SHI-1.

As said above, the functions are linear and this thus means that we can exhaustively search all possibilities, there are only 2^{16} for each bit, so in total there are $32 \cdot 2^{16} = 2^{21}$ expansion functions we have to carry out, which is doable. After iterating over all possible input messages, we find masks to apply over M such that a collision is found, this completes breaking the SHI-1 hash function.

Of course the SHI-1 hash function is a very weak hash and is not used at all. Therefore the next step is to insert the non-linear functions again and see what effect this has on the error propagation.

Chabaud and Joux continue to define two more simplified versions of SHA-0, SHI-2 and SHI-3. They use these to study the behavior of the non-linear functions in SHA-0. In SHI-2 they add the non-linear functions f_{if} and f_{maj} compared with SHI-1 and in SHI-3 they re-introduces the non-linear ADD to update the state of variable a, but replace f_{if} and f_{maj} again with f_{xor} (compare with SHA-0/1 in Section 1.2.1). They analyze with what probability the f_{if} and f_{maj} will behave as f_{xor} and find that due to the property of the f_{if} no consecutive perturbations may occur before round 16.

Given a change in c and d the output of f_{xor} does not change, while the output of f_{if} always changes. Since the f_{if} is active from round 0 to 19 a perturbation in both round 16 and round 17 will be propagated to state variables c and d by round 20. Finally they are able to find a valid pattern which behaves as correctly under all constraints with a probability of $1/2^24$. Analyzing the ADD function we see that so called 'carries' occur, which means that a perturbation in a certain bit position might cause a bit in the subsequent position to change.

1.4.2 Attacks Against SHA-1

1.5 Implications of attacks

As discussed earlier an attack has aim to find two messages x and y such that their hash result collide, i.e. $H(x) = H(y)$. We can quickly conclude that in order to make a successful attack the attacker needs to control both messages. If one of the messages is fixed, the attacker needs to perform a second-preimage attack, which is, as mentioned in section (TODO: ref), a lot harder and there are no such attacks known on SHA-1 at this moment. Because of this all messages, which are signed before a feasible collision attack became known, are safe. Also, because the attacker needs control over both messages, any party who creates and signs documents himself does not have to worry about collision attacks. However there is danger in any scenario where another party creates or controls the document which is to be signed; or any scenario where there is dependence on a document created and signed by another party. In the first scenario a malicious party could offer a document m for signing for which he has calculated a document m' with colliding hashes. Once received the signature on document m will be valid for m', and can thus be simply copied. In the second scenario, where the malicious user produces and signs a document m himself, it is possible for the attacker to deny ever having signed m by presenting m' and arguing he is victim of a forgery. Even though current attacks heavily constrain the difference between messages and often fix large parts of them to meaningless random values we have seen that only one such colliding block can be used to construct a poisoned block attack (see 1.3.3). The previous discussion applies in general to all hash functions for which collision attack (but no second pre-image attacks) are known. Thus it, at the moment of writing, applies to SHA-1, but the SHA-2 versions are safe.

1.6 Conclusion

As we have illustrated, SHA-0 is completely broken. The best attacks are becoming feasible to calculate within a reasonable time and as such, this function is completely insecure. Fortunately this was already foreseen at the time when SHA-1 was published. This hash remedies parts of the problems with SHA-0, but at this time there are already

attacks possible that are faster than the birthday attack. This is not very remarkable as the hash functions are almost identical, the only difference being the message expansion. The chance that this hash function will also succumb is therefore not unlikely.

The SHA-2 hash functions on the other hand have as of now not been broken. The best attacks possible only work on a reduced version, which gives hope for the strength of these functions. Again, the fact that this hash function is not broken yet is not very remarkable, as it is much more complex than SHA-1, something that can easily be seen by comparing Figures 1.4 and 1.2. Even if SHA-2 is broken though, it will be quite some time until these attacks will be feasible. Since a SHA-0 hash is 160 bits, a birthday attack has a complexity around 2^{80}, but since the SHA-2 functions produce much longer hashes, the complexity of birthday attacks against these functions range from 2^{112} to 2^{256}. Even if the latter would be broken with the complexity reduced to 2^{128}, which would be quite a successful attack, it would still be much much stronger than the original SHA-1 hash function.

This leaves the authors to wonder why such relatively short hash functions are still used. SHA-1 is very common and produces 160-bit hashes. Although a complexity of 2^{80} is not yet reachable, it is quite close to what is actually feasible. If the hash length would be doubled, the complexity would raise to 2^{160} which is completely impossible to attack. Even when it would be broken severely, it would still pose no practical problems for years to come. Furthermore, the cost of the additional hash length is small, a naive doubling of the hash length would cost a factor of two more in computing time. If the current rise of 64-bit processors is taken into account, optimized implementations could perhaps even be faster.

In any case, the story of SHA seems to be a story true for any hash function. MD5, the hash function used before SHA (and which is still used a lot today) also suffered the same fate. First a seemingly unbreakable hash function is released, using all kinds of complex functions to mix the input with some key. Then later some cracks begin to appear in the design and finally the hash function is broken. It seems that widely used hash functions are not much more than a collection of rounds, message expansion and some complex linear functions consisting of AND, OR and XOR gates. The result is something that looks impressive, but has never been proven to be secure.

This problem could be solved by using hash functions based on RSA or ElGamal, which have been proven to be secure (if at least the discrete log is hard, which is very probably is). The security of these hash functions would be beyond doubt and the only worry we would face is that the keys at some point are too short and can be brute forced, but this is of course true for any cryptographic function. The reason that these number theoretical routines are not used is of course that they are way too slow, and the second best people settle for is a seemingly complex function that runs fast on computers.

The cat and mouse game is not at an end yet, as in ...

- sha strength - problem with hashes in general (not number theoretically safe) - outlook and future prospect of SHA

Bibliography

[1] Florent Chabaud and Antoine Joux. Differential collisions in sha-0. In *Advances in Cryptology CRYPTO '98*, volume 1462, pages 253–261. Springer Berlin / Heidelberg, february 1998.

[2] Magnus Daum and Stefan Lucks. Attacking hash functions by poisoned messages. Eurocrypt 2005 presentation, 2005. `http://www.cits.rub.de/MD5Collisions/`.

[3] Krystian Matusiewicz, Josef Pieprzyk, Norbert Pramstaller, Christian Rechberger, and Vincent Rijmen. Analysis of simplified variants of sha-256. In *WEWoRC 2005 - Western European Workshop on Research in Cryptology*, pages 123–134, 2005.

[4] Ilya Mironov. Hash functions: Theory, attacks, and applications. Technical report, Microsoft Research, Silicon Valley Campus, november 2005.

[5] NIST. Fips publication 180-1: Secure hash standard. Technical report, National Institute of Standards and Technology (NIST), April 1995.

[6] NIST. Fips publication 180-2: Secure hash standard. Technical report, National Institute of Standards and Technology (NIST), August 2002.

[7] Vincent Rijmen. Current status of sha-1. Technical report, Rundfunk und Telekom Regulierungs-GmbH, Austria, february 2007.

[8] Vincent Rijmen and Elisabeth Oswald. Update on sha-1, 2005.

[9] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[10] Marc Stevens, Arjen Lenstra, and Benne de Weger. Target collisions for md5 and colliding x.509 certificates for different identities. Cryptology ePrint Archive, Report 2006/360, 2006. `http://eprint.iacr.org/`.

[11] Makoto Sugita, Mitsuru Kawazoe, Ludovic Perret, and Hideki Imai. Algebraic cryptanalysis of 58-round sha-1. In *Fast Software Encryption*, pages 349–365, 2007.

[12] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding collisions in the full sha-1. In *Advances in Cryptology - CRYPTO 2005*, pages 17–36, 2005.

[13] Xiaoyun Wang, Hongbo Yu, and Yiqun Lisa Yin. Efficient collision search attacks on sha-0. In *Advances in Cryptology - CRYPTO 2005*, pages 1–16, 2005.