Modified SHA-1 hash function (mSHA-1)

Abduvaliyev Abror, Sungyoung Lee and Young-Koo Lee
Dept. of Comp. Eng., Kyung Hee University, Global Campus, Suwon, Korea.
{abror, sylee}@oslab.khu.ac.kr, yklee@khu.ac.kr

Abstract

In this paper we propose a modification to SHA-1 hash function. For our scheme, we use the regularly distributed pseudo random function instead of logical functions. This changing produces unique hash values for unique messages and provides collision–resistance requirement to hash function.

1. Introduction

As we know, the hash function SHA-1 has been adopted by many government and industry security standards, in particular standards on digital signatures for which a collision-resistant hash function is required. In addition to its usage in digital signatures, SHA-1 has also been deployed as an important component in various cryptographic schemes and protocols, such as pseudorandom number generation, user authentication (Mercurial, Monotone) and key agreement, including TSL, SSL, PGP, SSH, S/MIME. Consequently, SHA-1 has been widely implemented in almost all commercial security systems and products [7, 8].

There were some analysis and publications for finding collisions in SHA-1. For example, In early 2005, Rijmen and Oswald devised an attack on a reduced version of SHA-1 — 53 out of 80 rounds — which finds collisions with a computational effort of fewer than 2^{80} operations. In February 2005, an attack by Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu was published.

The attacks can find collisions in the full version of SHA-1, requiring fewer than 2^{69} operations. (A brute-force search would require 2^{80} operations) [3, 4].

In this paper, proceeding from above results, we present new modifications to SHA-1. We use the pseudo random function instead of logical functions. The pseudo random function gives us the unique numbers depending on input message; it helps us to provide collision-resistant criteria for hash functions.

The rest of the paper is organized as follows. Section 2 contains related works. In Section 3, we give a description of SHA-1. Section 4 is dedicated to describing our proposed algorithm. Section 5 is devoted to conclusions for this paper.

2. Related Works

Yi-Shiung Yeh et al. [5] proposed two SHA-1 corrections to enhance the security of SHA-1. They re-wrote the original recursive equation into a general form and changed the indexes of the message schedule. The main advantage of this modification was the cost of time as it was in SHA-1. But on the other hand, it can not provide collision-resistant criteria for SHA-1.

Praveen Gauravaram and John Kelsey [2] also analyzed that the security of Damgard/Merkle variants which compute linear-XOR or additive checksums over message blocks, intermediate hash values, or both, and process these checksums in computing the final hash value have some weaknesses. Their result shows that, linear-XOR or additive checksums cannot protect hash functions from generic attacks.

3. Description of SHA-1

The hash function SHA-1 takes a message of length less than 2^{64} bits and produces a 160-bit hash value [1]. The input message is padded and then processed in 512-bit blocks in the Damgard/Merkle iterative structure. The purpose of message padding is to make the total length of a padded message a multiple of 512. The overall process of this algorithm can be explained using the following steps:

Step 1: Appending padding bits:

The message is padded so that its length in bits is congruent to 448 modulo 512. Suppose that the length of the message, *M*, is l bits. Append the bit "1" to the end of the message, followed by *k* zero bits, where *k* is the smallest, non-negative solution to the equation $l + 1 + k = 448 \bmod 512$. Then append the 64-bit block that is equal to the number l expressed using a binary representation. For example, the (8-bit ASCII) message "**abc**" has length $8 \cdot 3 = 24$, so the message is padded with a one bit, then $448 - (24 + 1) = 423$ zero bits, and then the message length, to become the 512-bit padded message.

Step 2: Initialization Vector:

A 160-bit buffer is used to hold intermediate and final results of the hash function. The buffer can be represented as five 32-bit registers (A, B, C, D and E). These registers are initialized with the following 32-bit hexadecimal values:

A: 0x67452301
B: 0xEFCDAB89
C: 0x98BADCFE
D: 0x10325476
E: 0xC3D2E1F0

Step3. Processing message in 512-bit blocks:

After this 16 blocks with the length of 32-bits (from M_0 to M_{15}) change to 80 blocks with the length 32-bits (from W_0 to W_{79}) with the help of below algorithm:

$W_i = M_i$, where $i = 0,1,\ldots,15$;

$W_i = (W_{i-3} \oplus W_{i-8} \oplus W_{i-14} \oplus W_{i-16}) << 1$ where $i = 16,17,\ldots,79$;

Step4. Defining functions and constants:

The main loop contains a sequence of logical functions $f_0, f_1, ..., f_{79}$ and four constants. Each f_i, $0 \le i \le 79$, operates on three 32-bit constants B, C, D and produces a 32-bit word as output. $f_i(B,C,D)$ is defined as follows: for constants B, C, D:

1. $f_i(B,C,D) = (B \wedge C) \vee ((\neg B) \wedge D)$, K_i = 0x5A827999, $i = 0,...,19$;
2. $f_i(B,C,D) = B \oplus C \oplus D$, K_i = 0x6ED9EBA1, $i = 20,...,39$;
3. $f_i(B,C,D) = (B \wedge C) \vee (B \wedge C) \vee (C \wedge D)$, K_i = 0x8F1BBCDC, $i = 40,...,59$;
4. $f_i(B,C,D) = B \oplus C \oplus D$, K_i = 0xCA62C1D6, $i = 60,...,79$;

Step5. The main loop:

In this step we will process message blocks with the help of below procedures. Firstly, we have to initialize five working variables *a, b, c, d, e* as below:

$a = A; b = B; c = C; d = D; e = E;$

Then we will begin processing in main loop, so the pseudo code of the main loop is:

```
for i = 0 to 79 {
T = (a <<< 5) + f_i(b,c,d) + e + K_i + W_i;
e = d; d = c; c = (b <<< 30);
b = a; a = t;
}
A = A + a; B = B + b; C = C + c; D = D + d;
E = E + e;
```

The last step is appending five working variables to one variable:

Hash = A||B||C||D||E;

4. Our proposed algorithm

4.1 Description of pseudo-random function.

As we mentioned above, hash functions have to be collision-resistant. There are also 2 more requirements in hash functions [7, 8]:

1. Deterministic. Two identical or equivalent inputs have to generate the same hash value or output.
2. Second pre-image resistance: It is infeasible to find any second input which has the same output as pre-specified input message.

According to these requirements we need to use the function which gives us a unique output for unique input. In this way, as a main function for modification to SHA-1 we choose a pseudo-random function which generates a unique pseudo-random number according to its input message. Full explanation of this function is given below:

It is known that if there is a sequence of pseudo random numbers in regular intervals distributed in an interval (0; 1), with the help of this sequence can be received the sequence of random variables with any laws of distribution. There are many algorithms for pseudo random number generators with the normal sequence, but the strongest is the following algorithm [6]:

$F(i) = \{x_i * Q\}$; where {.} - Number's fractional part. Q is the number which was chosen in special way and we can use the following values:

$$Q: \left\{ \sqrt{2};\ \frac{\sqrt{2}}{2};\ \frac{\sqrt{3}}{3};\ \sqrt{5};\ \frac{\sqrt{5}-1}{2}; \right\}$$

Here $\sqrt{2}$ is the best value for this algorithm. The reason is it gives us the strongest normal sequence and the longest repeating period. x_i is the value which was defined by users, in other words, we can say x_i is the key for this algorithm. The pseudo-code of this algorithm is given below:

```
temp = Math.sqrt(2) * x_i;
for i = 1 to k {
  arr[i] = toString(temp - Math.floor(temp));
  arr[i] = substring(arr[i], 2, 11);
}
```

For analyzing and evaluating the power of this algorithm we generated 1000,000 numbers. According to our findings, the repeating period equaled to 1000,000 with the first 8 digit of fractional part of values, so we can take the first 9 digits of fractional part of the number in terms of uniqueness. It means that all generated 1000,000 numbers are regularly distributed, unique and totally different from each other. The benefits of this algorithm are high speed of generating and the longest repeating period. The Figure 1 below shows the proof of regular distribution of pseudo-random number generator that shows number of random numbers which were distributed regularly in (0; 1) interval:

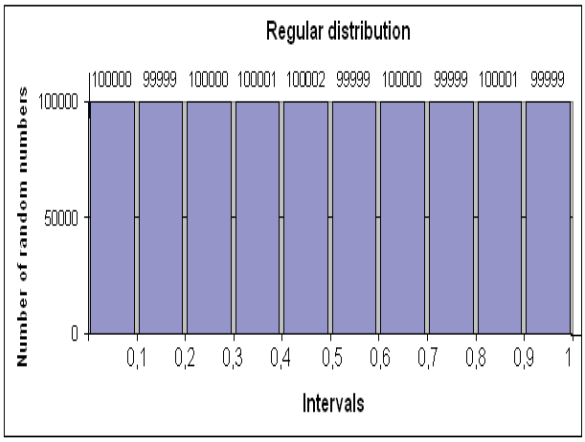


Figure 1. Regularly distributed pseudo random function.

Let's see this algorithm in a real example:

We have x_i =1, then the procedure looks like this:

$$\text{temp} = \sqrt{2} * 1 = 1.414213562;$$

$$\text{arr}[1] = \text{toString}(1.414213562 - 1.0) = 0.414213562;$$

$$\text{arr}[i] = \text{substring}("0.414213562", 2, 10) =$$

$$= 414213562$$

4.2 Modification to SHA-1

As we mentioned above, we are going to use pseudo-random function instead of logical functions. The logical functions in SHA-1 just shifts or rotates the bits to left-right and use the fixed numbers (constants). But we process the unique numbers which were generated with the help of above algorithm. Additionally, we can add one more variable to this function as a secret key K, so it can be used for message authentication. But a detailed discussion of this solution is outside the scope of this paper. So, we re-write *Step4* in SHA-1 as below:

The constants are the same as SHA-1, but the main function is changed to:

$$f_i(w_i) = F_i\{w_i * \sqrt{2}\},\ i = 0,1,...79;$$

This function gives us 80 pseudo-random values with the length of 32-bits for each number. Consequently, we will have the following changed pseudo code for the main loop:

$$for\ i = 0\ to\ 79\ \{$$

$$T = (a <<< 5) + f_i(W_i) + e + K_i + W_i;$$

$$e = d;\ d = c;\ c = (b <<< 30);$$

$$b = a;\ a = t;$$

$$\}$$

$$A = A + a; B = B + b; C = C + c; D = D + d;$$

$$E = E + e;$$

The last step is appending five working variables to one variable:

Hash = A||B||C||D||E;

The Figure 2 shows us the round of this hash function:

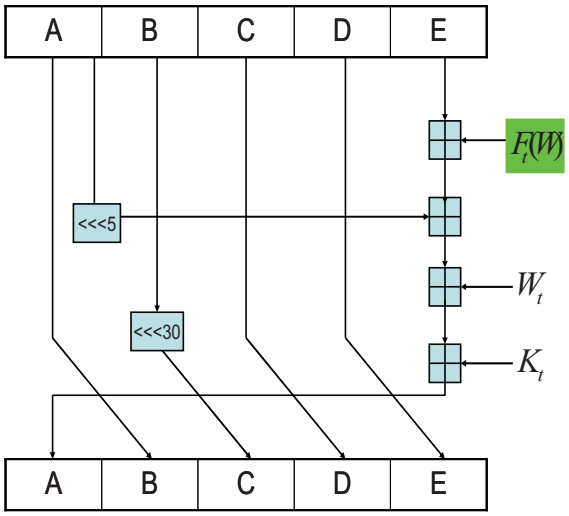
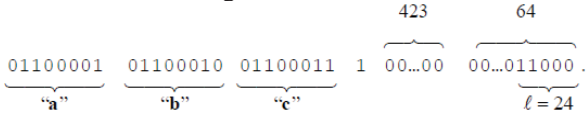


Figure 2. One round of hash function

Let's see this algorithm in a real example step by step:

Step1. Adding bits:

We have the message "abc". The binary representation of "abc" is 01100001 01100010 01100011. We need to convert it to 512-bit block; the description of converting is given in Section 3, *Step1.* So, we have the following 512-bit block:



Step2. Initialization Vector:

A: 0x67452301

B: 0xEFCDAB89
C: 0x98BADCFE
D: 0x10325476
E: 0xC3D2E1F0

Step3. Processing message in 512-bit blocks:

The 512-bit block divided to 16 $M_0 \ldots M_{15}$ 32-bit blocks are assigned to compute 80 $W_0 \ldots W_{79}$ 32-bit blocks:

$M_0 = 61626380$; $M_1 = 00000000$; … $M_{15} = 00000018$;

Consequently,

$W_0 = 61626380$; $W_1 = 00000000$; … $W_{79} = 2110912391$;

Step4. Computing main functions and main loop:

In this step, we show how to compute $f_i(w_i)$ function, in case $i = 0$:

$$f_0(w_0) = F_0\{w_0 * \sqrt{2}\};$$

$$F_0 = \{61626380 * 1.414213562\} \Rightarrow$$

$$= 87152862372965560 - 871528620 = 0.372965560$$

$$\Rightarrow substring(\text{"}0.372965560\text{"}, 2, 10) = 372965560;$$

$$\Rightarrow f_0(w_0) = 372965560;$$

Other $f_i(w_i)$ values also computed like the example above.

The following schedule shows the hex values of 5 values a, b, c, d and e which are taken from main loop:

t = 0; a = 73ab231e; b = ebc3462a; c = 53deb222; d = 2dac725c; e = 324ac647; …
t = 79; a = b53add17; b = a99444b1; c = 352dbca7; d = 2dac725c; e = c23a2244;

The final hash value is defined as:
A = 67452301 + b53add17 = cde899a5
B = efcdab89 + a99444b1 = 9961f03a
C = 98badcfe + 352dbca7 = 1c800018
D = 10325476 + 11adeb9c = 21e04012
E = c3d2e1f0 + c23a2244 = 860d0434;
Hash = A||B||C||D||E =
= cde899a5 9961f03a 1c800018 21e04012 860d0434;

5. Conclusions and future works

We have proposed the new algorithm which shows modification of SHA-1 hash function with the help of pseudo random function with real example. The greatest benefit of this algorithm is using unique numbers according to the input message. It means that the value of function depends on message only, not on constants as it was in original SHA-1. In future, we can use this algorithm for computing message authentication and integrity code by adding one more variable to pseudo random function as a secret key.

6. Acknowledgments

This research was supported by the MKE (Ministry of Knowledge Economy), Korea, under the ITRC (Information Technology Research Center) support program supervised by the IITA (Institute of Information Technology Advancement)" (IITA-2009-(C1090-0902-0002)) and is supported by the Brain Korea 21 projects and was supported by the Korea Science and Engineering Foundation(KOSEF) grant funded by the Korea government (MOST) (No. 2008-1342) and is supported by the IT R&D program of MKE/KEIT, [10032105, Development of Realistic Multiverse Game Engine Technology]

References

[1] NIST. *FIPS 180-2, Secure Hash Standard.* US Department of Commerce, Washington D.C., August 2002.

[2] Gauravaram, P., Kelsey, J. *Linear-XOR and Additive Checksums Don't Protect Damgard-Merkle Hashes from Generic Attacks*. In Malkin, T. (ed.) CT-RSA 2008. LNCS, vol. 4964, pp. 36–51. Springer, Heidelberg,2008

[3] Wang, X., Yin, Y.L., Yu, H.. *Finding collisions in the full SHA-1*. In Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 17–36. Springer, Heidelberg, 2005.

[4] Wang, X., Yu, H.. *How to Break MD5 and Other Hash Functions*. In Cramer, R.J.F. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 19–35. Springer, Heidelberg, 2005.

[5] Yi-Shiung, Y., Huang, T., Chen, I., Chou, Sh. *Analyze SHA-1 in message schedule.* Journal of Discrete Mathematical Sciences & Cryptography, 2007

[6] Rastrigin, L.A. *Static methods of searching*. M:Nauka, 1965

[7] Schneier, B. *Applied Cryptography*, John Wiley & Sons, New York, 1994

[8] Henri Gilbert & Helena Handschuh. *Security analysis of SHA-256 and sisters.* Lecture notes in computer science, Springer, Berlin, ISSN 0302-9743, 2008

[9] Henri Gilbert, Helena Handschuh. *Security Analysis of SHA-256 and Sisters*. In Selected Areas in Cryptography 2003, pp175–193

User keystroke authentication based on stable digraph pairs

Yoshihiro KANEKO Taku YAMAMOTO
Faculty of Engineering, Gifu University, Japan
E-mail: kaneko@info.gifu-u.ac.jp

Abstract

Both Joyce-Gupta method and Bergadano-Guntetti-Picardi method are well-known for user keystroke dynamics authentication by comparing latency of two consecutive keystroke, termed digraph, of test sample with that of reference sample. By millisecond units, one does not always type keys with the same keystroke latency. However, it seems that some digraphs are always faster than other slow digraphs, which is stable and unique to person. In this paper, we call such fast and slow digraph pair a stable digraph pair, based on which we propose a new method for user keystroke authentication. By combining the previous methods, we have tested our approach on 7 individuals, for a total of 35 samples achieving both false alarm rate and impostor pass rate of 0%, which is better result than those methods alone.

Keywords: Keystroke dynamics, user authentication, Joyce-Gupta method, Bergadano-Guntetti-Picardi method

1. Introduction

Recently there are various authentication systems using physical and behavioral features. Our study belongs to the latter and is in the area keystroke characteristics. In this area, two keys typed one after another is called a digraph. For instance, the type ”Japan” has four digraphs such as Ja, ap, pa, and an . More precisely, we use the latency (the elapsed time between the depression of the first key and that of the second) of digraphs for user authentication. We present a new method to utilize stable relationships between two digraphs such that some digraphs are always faster than other digraphs, which is unique to person. Throughout some experiments, we show our proposed method is valid.

2. Preliminaries

2-1. Relevant work

Much research on keystroke authentication has been done in the last decades. In [1], they outline keystroke user authentication methods until then. By the names of authors, their method is called JG method, which is well-known in this area and has affected so many other researches[2-6].

In most of researches so far, users first input short words repeatedly such as login-name, password, user-name and so on in order to make reference samples. In our study, input situation is different, where users are to input a longer words or sentences, but not so often. In [2], they deal with almost the same situation, but their authentication method described below is different from ours.

2-2. Joyce-Gupta method (J-G method)

This method is based on absolute difference of digraphs. Let N be a set of digraph. Suppose both samples T_1 and T_2 contain all elements of such N. For a digraph j, let t_j^1 and t_j^2 denote the mean of the latency of j in T_1 and T_2, respectively. Let us define the norm between those two samples by

$$\mathrm{Norm}\,(T_1, T_2) = \Sigma_{j\in N} \mid t_j^1 - t_j^2 \mid \qquad (1).$$

Suppose some user r have k samples. Then as a reference sample, let R be the collection of such k samples together. We calculate each norm between R and r's own samples and then get the mean and standard deviation of such k norms, denoted by N_m and N_d, respectively.

Using a positive scale S, we set the threshold I such as

$$I = N_m + S \times N_d \qquad (2).$$

If a test sample T of some person t satisfies Norm(R,T) $< I$, then we judge that the users r and t are the same person and otherwise they are different.

2-3. Bergadano-Guntetti-Picardi method

This method is based on relative difference of digraphs. Suppose, for instance, that ”Japan” contain the mean of the latency of four digraphs such as

Ja: 100mS, ap:110mS, pa:90mS, an:80mS.

Then those digraphs are sorted with respect to the latency in ascending order and ranked like

an: rank 1, pa: rank 2, Ja: rank 3, ap: rank 4.