Cryptanalysis of T-function-Based Hash Functions

Applications to MySQL Password Algorithms

Frédéric Muller[1] and Thomas Peyrin[2]

[1] HSBC, Paris, France,
frederic.muller@hsbc.fr
[2] France Télécom R&D, Issy les Moulineaux, France,
thomas.peyrin@orange-ft.com

Abstract. T-functions are a useful new tool to design symmetric-key algorithms, introduced by Klimov and Shamir in 2002. They have already been used to build stream ciphers and new applications for block ciphers and hash functions have been recently suggested.
In this paper, we analyze the security of several possible constructions of hash functions, based on T-functions. We show that most natural ideas are insecure. As an application, we describe a practical preimage attack against the dedicated hash function used in the MySQL password-based authentication mechanisms.

Key words: T-functions, hash functions, MySQL

1 Introduction

Following many recent cryptanalysis results against popular algorithms like MD5 [23] or SHA-1 [22], the design of secure hash functions is now in question. Some issues have been raised about the theoretical security of the Merkle-Damgård mode of operation [2, 4, 5] but the heart of the problem seems to be the design of "good" compression functions. The typical design method is to use a block cipher in the Davies-Meyer mode, however the large block length necessary to prevent birthday attacks often requires the use of *ad hoc* block ciphers, instead of relying on AES for instance.

An interesting new research direction is to find new paradigms to build compression functions. As an example, one can mention SMASH, a recent attempt by Knudsen [10] which was later broken by Pramstaller *et al* [20], or the Butterfly transforms [18]. Another interesting idea consists in **building a compression function which relies on another primitive than a block cipher.** Some hash functions already rely on number-theoretic assumptions, however it might also be interesting to find constructions that rely on simple and efficient symmetric-key primitives.

In this paper, we investigate the security of hash functions based on T-functions. Our analysis is motivated by the observation that many popular functions (like MD5 and SHA-1) are close to a T-function (only the circular rotations break the triangular structure). Also, we observed that some *ad hoc* designs (like the MySQL dedicated hash function, for instance) were based on T-functions.

First, we investigate several "natural" constructions of T-function-based hash functions that a designer could consider, and we show why they are insecure. This

Appeared in M.S. Rhee and B. Lee (Eds.): ICISC 2006, LNCS 4296, pp. 267–285.

analysis is actually quite simple, but we show that it has a devastating effect when taking the concrete example of the MySQL dedicated hash function, whose compression function is based on a T-function. We describe a practical preimage attack against this hash function, which allows to retrieve instantly the user's passwords.

2 Possible Use of T-functions in Hash Functions

2.1 Definition of T-Functions

T-functions are a new family of primitives that has recently emerged, especially for stream ciphers design. Initially introduced by Klimov and Shamir in 2002 [7], a T-function maps an n-bit input to an n-bit output, where the i-th bit of the output depends only on the i Least Significant Bits (LSB) of the input. T actually stands for "Triangular", as illustrated in Figure 1.

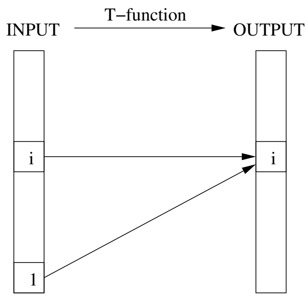

Fig. 1. General structure of a T-function

More precisely, let $x = (x_1, \ldots, x_n)$ be the input of the T-function F, and $y = (y_1, \ldots, y_n)$ its output. The definition implies that y_i must be computable only from $(x_1, \ldots, x_i)$. This idea is quite natural, since most usual software operations are T-functions, like boolean operations (IF, AND, XOR) and arithmetic operations (+, *) taken modulo a power of 2. Besides, T-functions can be composed, while still preserving the Triangular property. Also, provided n is equal to the word size of the underlying processor, implementation of such T-functions in software can be made very efficient.

For practical purpose, it is customary to use a **multi-word T-function**. It is a mapping applied to m words of n bits, which has the property that the i-th bits of each output depends only on the i LSB's of each input. Multi-word T-functions fit better on 32-bit or 64-bit architectures than single-word T-functions. There has also been some work on suitable properties for T-functions in stream ciphers (like invertibility and cycle length) [8]. More recently, it was suggested to use T-functions to build MDS nonlinear mapping in block ciphers, and self-synchronizing hash functions [9].

Appeared in M.S. Rhee and B. Lee (Eds.): ICISC 2006, LNCS 4296, pp. 267–285.

2.2 Compression functions

Typically, a hash function H maps an arbitrary input $M \in \{0,1\}^*$ to a fixed-size output $H(M) \in \{0,1\}^n$. A mode of operation is often provided, in order to handle arbitrary inputs and to extend the domain of a smaller function, also called **compression function** and noted h, which works on fixed-size inputs. Figure 2 summarizes this construction method, also known as Merkle-Damgård construction.

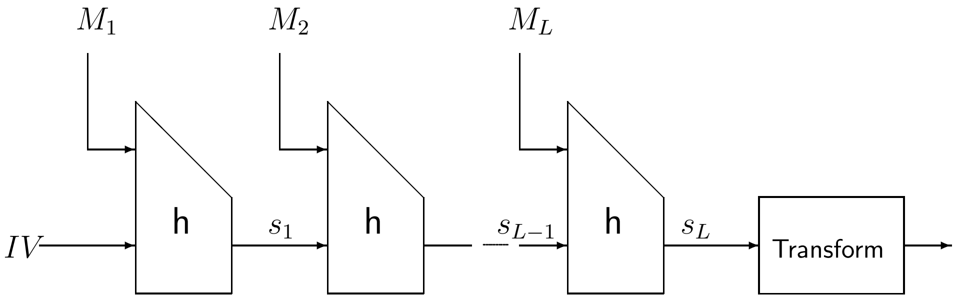

Fig. 2. Description of the hash function H

While there are theoretical background for using it, the final transform is often omitted, and the last chaining variable s_L becomes the hash of the message M. IV is a n-bit constant, which is part of the specifications of the hash function, while the variables $(M_1 \ldots M_L)$ (of length t bits each) are the "chunks" of the message. A padding may need to be specified to avoir some attacks and to handle messages whose length is not a multiple of t.

To summarize, the compression function h maps an input of length $t+n$ bits to an output of n bits. A natural idea is to **build a compression function based on a T-function** F. By definition, F has the same input and output domain, so one needs to find a way to decrease the output length : there are several natural ideas to achieve it.

2.3 The truncation method

In stream cipher designs, T-functions are thought of as mappings such that knowing the Most Significant Bits (MSB) of the output reveals little information about the input. Roughly, the MSB's are used as the keystream in a stream cipher design. This was the original idea by Klimov and Shamir [8, 13], although more complex constructions have been preferred in practice [3, 11, 14].

In the context of compression function design, the natural idea to decrease the output size of the T-function is therefore to truncate the output's LSB. Call F the T-Function and $t+n$ its domain length. The t LSB's of the output of F are truncated in order to build the compression function h. The output size of h is indeed $t+n-t=n$ bits, as needed (also see Figure 3).

Appeared in M.S. Rhee and B. Lee (Eds.): ICISC 2006, LNCS 4296, pp. 267–285.

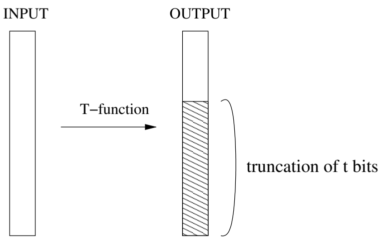

Fig. 3. The truncation method

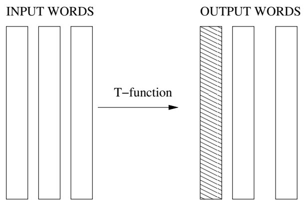

Fig. 4. The reduction method

This idea easily extends to multi-word T-functions. For instance, suppose that $t+n$ is a multiple of 32. Then F is a multi-word T-function operating on $l = (t+n)/32$ words of 32 bits each. To build a compression function h, the t/l LSB's of each output word would be truncated.

2.4 The reduction method

Another natural solution is to reduce the number of output words of a (multi-word) T-function. Therefore F becomes a mapping which has different input and output domain. For instance, if both t and n are multiples of 32, one could design a Triangular mapping which takes an input of $(t+n)/32$ words and outputs $n/32$ words only. Such a design can be built from scratch, or from a pre-existing T-function[3] (also see Figure 4).

A good example of this method is the MySQL dedicated hash function, where the input of h is of length 4 words (3 words from the chaining variable, and 1 word which is a chunk of the password) and the output length is 3 words. All operations in h are Triangular, therefore h itself is a T-function (see Section 5.2 for more details).

3 Analysis of the "natural" solutions

It is quite clear that all the solutions proposed above do not present a "pseudo-random" behavior. For instance, flipping only the most significant bit(s) of the input will modify only the most significant bit(s) of the output. However, it does not immediately extend to collision or preimage attacks. In this Section, we exploit the structure of T-functions to propose generic collision and preimage attacks against all the "natural" constructions proposed in Section 2.

3.1 Attacking the truncation method

Let h be a compression function built from the truncation of a l-word T-function F. The input x of F is composed by l-bit values noted $x = (x_1 \ldots x_{(t+n)/l})$ and its output y is composed by l-bit values noted $y = (y_1 \ldots y_{(t+n)/l})$. By definition,

$$h(x) = y' = (y'_1 \ldots y'_{n/l}) = (y_{(t/l)+1} \cdots y_{(t+n)/l})$$

[3] In this case, it comes down to the truncation method described in Section 2.3, where entire output words are removed, instead of the LSB's of each word

Appeared in M.S. Rhee and B. Lee (Eds.): ICISC 2006, LNCS 4296, pp. 267–285.

We describe a **pseudo-preimage attack against** h, *i.e.* for a random input challenge defining an output y' of h, we find an arbitrary input x such that $h(x) = y'$. It is called "pseudo-preimage" attack because there is no constraint on the n input bits which correspond to the chaining variable. Therefore, they are not necessarily equal to the IV of the hash function (see Figure 2). However, it is well-known that such attacks can be extended to preimage attacks against the whole hash function, using a meet-in-the-middle approach [12].

An attacker can compute a pseudo-preimage, *layer by layer*. The sketch of our attack against the truncation method is the following :

- Fix an arbitrary value for $x_1, x_2, \ldots, x_{t/l}$ and fix $pos = (t/l) + 1$.
- For all the 2^l candidates of x_{pos}, compute y_{pos} (possible since F is triangular).
- Choose arbitrarily a candidate x_{pos} such that $y_{pos} = y'_{pos-(t/l)}$, increment pos and continue to the next layer.
- If no such candidate exists, return to the first step.
- If $pos = (t+n)/l$, the input $x = (x_1, \ldots, x_{(t+n)/l})$ clearly satisfies $h(x) = y'$.

Since there are 2^l possible choices for x_{pos}, it is likely that one of the candidates matches the word $(pos - (t/l))$ of the challenge, at each step. Otherwise we draw at random another value of $x_1, x_2, \ldots, x_{t/l}$ until a solution is eventually found. More precisely, we have a probability of P to find a valid candidate at one precise step, where:

$$P = 1 - \frac{\text{number of mappings where } y'_{pos-(t/l)} \text{ does not appear}}{\text{total number of mappings}} = 1 - \left(\frac{2^l - 1}{2^l}\right)^{2^l}.$$

Finally, **this algorithm costs about $\frac{n}{l} \times 2^l \times P^{-\frac{n}{l}}$ computation steps to find a pseudo-preimage** while the expected strength is 2^n. We show in Appendix A that $P^{-1/l} < \sqrt{2}$, which validates our attack since $P^{-1/l} < 2$.

Note that this attack can be utilized to find a pseudo-collision, i.e. two different inputs x, x' such that $h(x) = h(x')$. By choosing arbitrarily an input x and running the previously described attack for the challenge $y = h(x)$, we eventually get an input x' such that $h(x) = h(x') = y$. The complexity of this attack is the same as the preimage one while the expected strength is $2^{\frac{n}{2}}$, and we show in Appendix A that $P^{-1/l} < \sqrt{2}$.

3.2 Attacking the reduction method

The reduction methods suffer from the same weaknesses as the truncation method, as an attacker can compute a preimage layer by layer. Suppose that the input is composed of l words and the output of $l' < l$ words.

First, for all possible least significant bits of the input (2^l candidates), he computes the least significant bits of the output and searches for a match with the bits from the challenge. If a solution is found, he jumps to the next layer (*i.e.* the second least significant bits), etc .... Since $l > l'$, it is very likely that a solution exists at each step.

With the same reasoning as for the truncation method, we can conclude that **this algorithm costs about $\frac{n}{l'} \times 2^{l'} \times P^{-\frac{n}{l'}}$ computation steps to find a**

Appeared in M.S. Rhee and B. Lee (Eds.): ICISC 2006, LNCS 4296, pp. 267–285.

pseudo-preimage, where:

$$P = 1 - \left(\frac{2^{l'} - 1}{2^{l'}} \right)^{2^l} .$$

Using the proof in Appendix A, we have $P^{-1/l'} < 2$. A good illustration is given in the next Sections with the MySQL dedicated hash function, where $l = 4$ and $l' = 3$.

This attack also leads to a pseudo-collision one, in the same manner as for the truncation method. The complexity of this attack is the same as the preimage one, and $P^{-1/l} < \sqrt{2}$ is a direct consequence of the proof in Appendix A .

3.3 Some possible strengthening

All these attacks result from the following fact : due to their triangular structure, T-functions allow to attack some parts independently. More generally, functions for which some parts can be separated independently are bad candidates for hash functions. For example, this problem is present when one wants to build a multiple-length hash function from simple-length primitives [19]. Naturally, one can think of some possible strengthening in order to thwart such attacks :

- the **feed-forward** consists in XORing some input bits to the output of the compression function. It is typically used to break the invertibility of block cipher-based compression functions. The feed-forward is also triangular, so it would not help here.
- the **folded feed-forward** consists in exchanging the least and most significant part of the input before the feed-forward. Such folding has already been used in stream ciphers design to strengthen the output function [6]. Such solutions would require a separate analysis, but we believe trade-off attacks could allow to invert the compression function for less than 2^n work.
- **tweaking** the T-functions to make the most significant bits very difficult to invert, when the least significant bits are hidden. However, this requires major changes to T-function designs.

3.4 Summary

To summarize, most “natural” constructions based on T-function turn out to be weak regarding collision and preimage attacks. The attacks we described are very simple, but they are generic : they are direct consequences of the structure of T-functions. Despite this simplicity, the impact in practice can be devastating. As an illustration, we analyze the dedicated hash function used in MySQL password-based authentication mechanisms, whose compression function is triangular.

4 Application to (old) MySQL Authentication Mechanisms

MySQL is a very popular database open source technology [16]. It was initially conceived (and is still maintained today) by a Swedish company called MySQL

Appeared in M.S. Rhee and B. Lee (Eds.): ICISC 2006, LNCS 4296, pp. 267–285.

AB. Among many other things, MySQL contains some security mechanisms. They are used to authenticate a client that wishes to connect to the database server. The client and the server initially share **a password** and the authentication is a challenge-response, based on the password only, without the use of any public-key cryptography. The server generates a fresh random challenge, generally called **the salt** and sends it over to the client. Upon reception, the client builds an authentication message (*aka* scrambled message) by mixing the salt and the password and sends its response to the server. Finally, the server recomputes the scrambled message, and compares this result with what he received from the client. Depending on this verification, he accepts or rejects the connection. This protocol is represented in Figure 5.

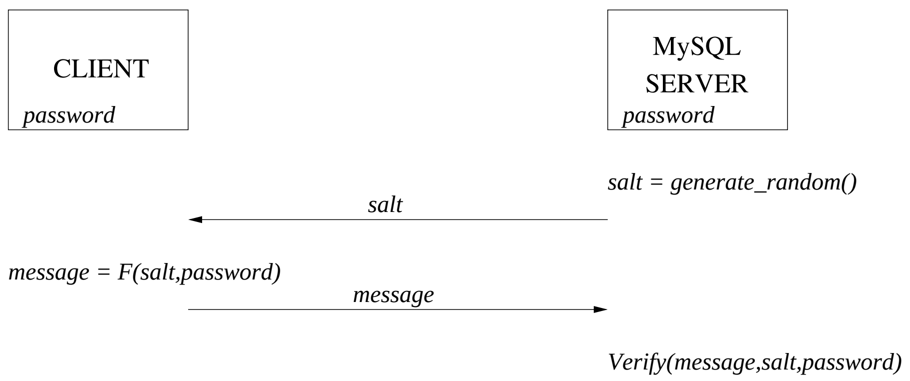

Fig. 5. Password-based Challenge-response Authentication in MySQL

Actually, the real construction is slightly different : indeed, one wants to avoid the loss of all user's passwords in case of intrusion on the server. Therefore, the server stores only a hash of each user's password, instead of the password itself. A hash function H is specified for that purpose in the MySQL specifications. The function F is applied to $H(password)$ instead of the actual $password$, so what the client and the server both compute is actually :

$$message = F(salt, H(password))$$

As we can see, all an attacker needs to know in order to impersonate the legitimate client is the value of $H(password)$.

Until version 3.23, MySQL used a dedicated hash function H and a dedicated **scrambling** function F. H is a usual iterated construction based on a compression function, while F is a pseudo-random number generator, whose seed is derived from the salt and $H(password)$. The last version using these old mechanisms (3.23.58) was released in September, 2003. Since then, new versions have been developed (4.0 and above), for which they were replaced by constructions based on the popular hash function SHA-1 [17]. Still, the old security algorithms continue to be implemented, to ensure compatibility with old versions. The description of H and F can be obtained on the MySQL website [15], by downloading the source code (in particular, see the `password.c` file). A good informal description of the protocol can also be found at the address `http://www.redferni.uklinux.net/mysql/`.

The old MySQL password algorithms (those used in versions 3.23 and earlier) have already been studied in 2002 [1]. An attack against the F function was proposed, which allows an attacker to recover $H(password)$, provided he intercepts about 10 pairs $(salt, message)$. This is sufficient for practical purpose, which motivated the introduction of new security algorithms. However, **studying the dedicated hash function remains an open research topic**.

In this paper, we propose a preimage attack against this hash function. Although this was not apparently a design criteria, we observe that its compression function is based on a T-function and we apply the attacks described in Section 3. First, we describe the MySQL authentication mechanisms. Then we show an attack that finds preimages for the compression function, and finally we perform a meet-in-the-middle attack against the full hash function. This allows to mount a **square-root dictionary attack** : a password with entropy of n bits is recovered with $2^{n/2}$ computation steps. This attack has been implemented in practice on a standard PC and takes a few seconds, independently of the password length : if the password is too long, an "equivalent" password (i.e. leading to the same hash value) will be retrieved instead.

5 Description of the Mechanisms

5.1 The Old MySQL Hash Function

The old MySQL hash function H takes an input of arbitrary size and outputs two words of 30 bits each. Therefore it is a mapping

$$H : \{0,1\}^* \longrightarrow \{0,1\}^{60}$$

In practice, the input is a password which is split into a sequence of n bytes. No padding is specified, so the input length must be a multiple of 8 bits. H is an iterated hash function. At the i-th iteration, it updates a chaining variable s^i of size 96 bits (the concatenation of 3 words of 32 bits which are called n_1^i, n_2^i and add^i: $s^i = n_1^i || n_2^i || add^i$), by

$$s^i = h(s^{i-1}, c^i)$$

As usual, h is called the **compression function** and c^i is the i-th byte of the password. The initial value s^0 consists of the following constants :

$$\begin{aligned} n_1^0 &= 1345345333 \\ n_2^0 &= 305419889 \\ add^0 &= 7 \end{aligned}$$

When all password bytes have been used, after k iteration of the compression function, the final value of the state is truncated from 96 to 60 bits, by keeping only the 30 least significant bits of n_1^k and n_2^k. These 60 bits constitute the hash of the password. Also refer to Figure 2 for a description of H, where the final transform is a truncation from 96 to 60 bits.

Due to the limited output size, H can offer at most a security of 2^{30} regarding collision attacks and 2^{60} regarding preimage attacks.

Appeared in M.S. Rhee and B. Lee (Eds.): ICISC 2006, LNCS 4296, pp. 267–285.

5.2 The Compression Function

The compression function h is a mapping

$$h : \{0,1\}^{96} \times \{0,1\}^{8} \longrightarrow \{0,1\}^{96}$$

which has the property to combine only triangular operations. More precisely, it mixes basic arithmetic and logical operations :

$$\begin{aligned} n_1^i &= n_1^{i-1} \oplus (((n_1^{i-1} \wedge 63) + add^{i-1}) \cdot c^i + (n_1^{i-1} \ll 8)) \\ n_2^i &= (n_2^{i-1} \ll 8) \oplus n_1^{i-1} \\ add^i &= add^{i-1} + c^i \end{aligned}$$

where the additions are taken modulo 2^{32} and $\ll 8$ denotes a shift by 8 bits to the left[4].

Looking at the chaining variable and the password chunk c^i as input words, this compression function follows exactly the **reduction method** introduced in Section 2.4, with input domain of 4 words and output domain of 3 words. There is a slight technical difference here, since the chaining variable words have length 32 bits, while the password chunk has length of 8 bits only. But this does not change fundamentally the analysis.

5.3 The Scrambling Function

The scrambling function F operates on a random value *salt* (which is a sequence of bytes, like the password) and on the hash of the password $H(password)$, by

$$message = F(salt, H(password))$$

Actually, F can be seen as a Pseudo-Random Number Generator (PRNG) : its *seed* is derived by :

$$seed = H(salt) \oplus H(password)$$

The scrambled message is then generated from several consecutive output bits of the PRNG. More precisely, *seed* is represented as two words of 30 bits each : $seed_1$ and $seed_2$. The update works as follows :

$$\begin{aligned} seed_1' &= (3 \cdot seed_1 + seed_2) \bmod (2^{30} - 1) \\ seed_2' &= (seed_1 + seed_2 + 33) \bmod (2^{30} - 1) \end{aligned}$$

After each update, a pseudo-random output is extracted by :

$$output = \left\lfloor \frac{seed_1'}{34638622} \right\rfloor \in \{0, \ldots, 30\}$$

This output is added to 64 and mapped to a printable character using the ASCII table. This operation is performed l_s times where l_s is the length (in bytes) of the salt. The typical length is $l_s = 8$, although other length (up to $l_s = 20$) are handled by the protocol. A final masking is applied : the random number generator is run

[4] This is equivalent to a multiplication by $2^8 = 256$, modulo 2^{32}

Appeared in M.S. Rhee and B. Lee (Eds.): ICISC 2006, LNCS 4296, pp. 267–285.

one extra round, and the resulting output byte (called *mask*) is XORed to every byte of the message.

The result is the authentication message which is sent by the client to the MySQL server. Depending on the version of MySQL, two security protocols (protocols 9 and 10) are possible. If protocol 9 is used, the scrambling function is slightly weakened : the entropy of the initial state is reduced and the final masking is removed. In this paper, we focus only on protocol 10, *i.e.* the "full" scrambling function, and we assume that $l_s = 8$. The attack would easily translate to situations where $l_s > 8$ or where the "weakened" version is used.

5.4 The New Construction

In more recent MySQL versions (4.0 and above), all these dedicated mechanisms have been replaced by a new scrambling function $F(salt, password)$ based on SHA-1 [17].

$$v = \text{SHA-1}(\text{SHA-1}(password))$$
$$F(salt, password) = v \oplus \text{SHA-1}(salt||v)$$

where || denotes the concatenation. The security of this new mechanism relies on the security of SHA-1. New results have recently been published [22], however they concern collision resistance, while the problem here is essentially preimage resistance.

5.5 The cryptanalysis scenario

First, we suppose that an attacker wants to attack the old authentication protocol of MySQL. He listens to communications between the client and the server, and learns pairs $(salt, message)$. His first goal is to retrieve the value of $H(password)$. This information is sufficient to later impersonate the legitimate client. We call it the **unscrambling attack**. It basically comes down to breaking the PRNG, given several output bits. Such an attack was already proposed in [1]. We further analyze the security of the MySQL scrambling function in Appendix B.

If for some reason, the attacker needs to find the actual client's password, he needs to **break the one-wayness** of H, *i.e.* to retrieve $password$ from $H(password)$. This scenario can arise after the unscrambling attack, or in case of intrusion on the database server : in this case, the hash of each user's password is compromised.

6 Preimage Attacks against the MySQL Dedicated Hash Function

In this section, we show how to break the one-wayness of the MySQL dedicated hash function. We are given a hash of the user password and retrieve the password almost instantly. This attack has been implemented on a regular PC (Pentium IV, 2.5 Ghz with 2Gb of RAM), and takes negligible amount of time.

We first show how to compute pseudo-preimage against the compression function, following the framework given in Section 3. Then we perform a *meet-in-the-middle* attack against the whole hash function.

Appeared in M.S. Rhee and B. Lee (Eds.): ICISC 2006, LNCS 4296, pp. 267–285.

6.1 Breaking the compression function

The compression function h is described by the following equations :

$$\begin{aligned} n_1^i &= n_1^{i-1} \oplus (((n_1^{i-1} \wedge 63) + add^{i-1}) \cdot c^i + (n_1^{i-1} \ll 8)) \\ n_2^i &= (n_2^{i-1} \ll 8) \oplus n_1^{i-1} \\ add^i &= add^{i-1} + c^i \end{aligned}$$

where the incoming chaining value $(n_1^{i-1}, n_2^{i-1}, add^{i-1})$ and the output (n_1^i, n_2^i, add^i) are composed with 3 words of length 32 bits and c^i is the password character to be hashed.

This compression function follows the reduction method, introduced in Section 2.4. We apply the pseudo-preimage attack described in Section 3.2 : the general idea is to proceed *layer by layer*. Actually, we can directly write the method that allows to compute all preimages corresponding to a given challenge, noted (n'_1, n'_2, add'). Each preimage is a pair $\{c, (n_1, n_2, add)\}$ that maps to the challenge through h. We start by enumerating all possible password characters c. Then, we do the following for each c :

- First, observe that $add = add' - c$
- Guess the 6 LSB of n_1. We get that

  $$n_1 \wedge 255 = (n'_1 \wedge 255) \oplus (c \cdot ((n_1 \wedge 63) + add))$$

  So we obtain the 8 LSB of n_1. This must match the previous guess on the 6 LSB, otherwise we reject it. In general, only 1 guess out of $2^6 = 64$ remains at this point.
- The value of $x = ((n_1 \wedge 63) + add) \cdot c$ is known and :

  $$n_1 = n'_1 \oplus (x + (n_1 \ll 8))$$

  which allows to derive the value of n_1 byte by byte
- Finally, we derive n_2 by :

  $$n_2 \ll 8 = n'_2 \oplus n_1$$

There is no way to determine the 8 MSB of n_2 since they are not used in the compression function. This inversion of h has been implemented on a regular PC and takes a negligible amount of time. In general, the solution obtained is unique, for each guess on c.

6.2 Application to the full hash : meet-in-the-middle

When it is possible to compute preimages for the compression function, it is well known that a *meet-in-the-middle attack* allows to extend the attack to the full hash (see [21] for instance). Our basic assumption is that the password is an unknown vector of 8 bytes, *i.e.* its entropy is 64 bits. We observe that if the password is longer, it is very likely that we will found an alternative 8-byte password that maps to the correct hash, since the output size is only 60 bits. Then, the framework of the attack is the following :

- Split the search space in two halves (of 4 characters each for instance)

– Start from the initial chaining value, and apply 4 times the compression function, for each possible sequence of 4 characters.
– Start from the password hash and invert 4 times the compression function with the previously explained technique, for each possible sequence of 4 characters.
– Find a match between the forward computation and the backward computation.

The complexity of this attack is only $(2^8)^4 = 2^{32}$ in time and memory. This represents a **square-root dictionnary attack**. Indeed, the complexity is only the square-root of the size of the search space.

6.3 Analysis

We implemented the previous attack, but several modifications were introduced compared to the theoretical analysis :

– First, we did not consider a character as an 8-bit unknown, but we restricted our analysis to a subset of about 100 characters (alphanumerical, plus some usual symbols) that are easily rechable from the keyboard.
– Secondly, instead of splitting the 8 characters in $4+4$, we split it into $3+5$. We stored in memory the candidates for the backward computation (about 100^3 candidates), then we sorted this list (using quick-sort). Finally, we enumerated the list of 100^5 "forward candidates". This is a more reasonable trade-off, since memory requirements are more difficult to satisfy in implementations than time requirements.
– Finally, there is no use in identifying the candidates using the whole n_2 value, since its MSB are not used by the compression function. As a result, we slightly decrease the memory requirements, but we also increase the number of matches we obtain.

Typically, our implementation takes less than one second to retrieve a password randomly chosen by the user. If the password has length > 8 characters, we generally find many alternative candidates.

To give an example, if we pick the password "MySQL123". The hash of the password is ($n_1^k \wedge K = \texttt{0x1b03947c}, n_2^k \wedge K = \texttt{0x1d6d177b}$), with $K = \texttt{0x3fffffff}$. Exploring all passwords of length up to 8, we found two solutions that map to this hash : "MySQL123" and "RGp0mA23". The second candidate is an alternative password that could be later substituted to the original one.

7 Conclusion

We considered several natural solutions to build hash functions based on T-functions, a new and popular primitive in symetric-key designs. Following recent cryptanalysis results, new ideas might be necessary for constructing hash functions, but our results show that **the "natural" solutions based on T-functions fail to provide a satisfactory level of security**. While the attacks we describe are very simple, their practical effect can be devastating.

As an illustration, we focused on the old MySQL password-based authentication mechanisms. Until 2003, they included a dedicated hash function for which

Appeared in M.S. Rhee and B. Lee (Eds.): ICISC 2006, LNCS 4296, pp. 267–285.

no analysis had yet been published. We observed that its compression function is based on a T-function and therefore we apply our generic analysis. We described a square-root dictionnary attack to find preimages. It retrieves the user's password (or an equivalent one if the password is too long from a hash value, in less than a second). These results raise some interesting perspectives about the design of T-function-based hash functions.

8 Acknowledgement

The authors would like to thank the anonymous referees and Yannick Seurin for their valuable comments and for spotting an error in the original complexity analysis.

References

1. I. Arce, A. Azubel, E. Kargieman, G. Richarte, C. Sarraute, and A. Waissbein. An attack on the MySQL authentication protocol. 2002. Technical report from Core Security Technologies.
2. J-S. Coron, Y. Dodis, C. Malinaud, and P. Puniya. Merkle-Damgård Revisited: How to Construct a Hash Function. In V. Shoup, editor, *Advances in Cryptology – Crypto'05*, volume 3621 of *Lectures Notes in Computer Science*, pages 430–448. Springer, 2005.
3. J. Hong, D. Lee, Y. Yeom, and D. Han. A New Class of Single Cycle T-functions. In H. Gilbert and H. Handschuh, editors, *Fast Software Encryption – 2005*, volume 3557 of *Lectures Notes in Computer Science*, pages 68–82. Springer, 2005.
4. A. Joux. Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions. In M. Franklin, editor, *Advances in Cryptology – CRYPTO'04*, volume 3152 of *Lectures Notes in Computer Science*, pages 306–316. Springer, 2004.
5. J. Kelsey and B. Schneier. Second Preimages on n-Bit Hash Functions for Much Less than 2^n Work. In R. Cramer, editor, *Advances in Cryptology – Eurocrypt'05*, volume 3494 of *Lectures Notes in Computer Science*, pages 474–490. Springer, 2005.
6. A. Klimov. *Applications of T-functions in Cryptography.* PhD thesis, Weizmann Institute of Science, 2004. http://www.wisdom.weizmann.ac.il/ ask/.
7. A. Klimov and A. Shamir. A New Class of Invertible Mappings. In B. Kaliski, Ç. Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems (CHES) – 2002*, volume 2523 of *Lectures Notes in Computer Science*, pages 470–483. Springer, 2002.
8. A. Klimov and A. Shamir. Cryptographic Applications of T-functions. In M. Matsui and R. Zuccherato, editors, *Selected Areas in Cryptography – 2003*, volume 3006 of *Lectures Notes in Computer Science*, pages 248–261. Springer, 2004.
9. A. Klimov and A. Shamir. New Applications of T-functions in Block Ciphers and Hash Functions. In H. Gilbert and H. Handschuh, editors, *Fast Software Encryption – 2005*, volume 3557 of *Lectures Notes in Computer Science*, pages 18–31. Springer, 2005.
10. L. Knudsen. SMASH - A Cryptographic Hash Function. In H. Gilbert and H. Handschuh, editors, *Fast Software Encryption – 2005*, volume 3557 of *Lectures Notes in Computer Science*, pages 228–242. Springer, 2005.
11. S. Künzli, P. Junod, and W. Meier. Distinguishing Attacks on T-Functions. In E. Dawson and S. Vaudenay, editors, *International Conference on Cryptology in Malaysia (MyCrypt 2005)*, volume 3715 of *Lectures Notes in Computer Science*, pages 2–15. Springer, 2005.

Appeared in M.S. Rhee and B. Lee (Eds.): ICISC 2006, LNCS 4296, pp. 267–285.

12. A. Menezes, P. van 0orschot, and S. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1996.
13. J. Mitra and P. Sarkar. Time-Memory Trade-Off Attacks on Multiplications and T-functions. In P. Lee, editor, *Advances in Cryptology - Asiacrypt'04*, volume 3329 of *Lectures Notes in Computer Science*, pages 468–482. Springer, 2004.
14. F. Muller and T. Peyrin. Linear Cryptanalysis of the TSC Family of Stream Ciphers. In B. K. Roy, editor, *Advances in Cryptology - Asiacrypt'05*, volume 3788 of *Lectures Notes in Computer Science*, pages 373–394. Springer, 2005.
15. MySQL Downloads. See `http://dev.mysql.com/downloads`.
16. MySQL Website. See `http://www.mysql.com`.
17. National Institute of Standards and Technology (NIST). Secure Hash Standard FIPS Publication 180-1, April 1995. Available at `http;//www.itl.nist.gov/fipspubs/fip180-1.htm`.
18. J. Patarin and P. Camion. Design of Near-Optimal Pseudorandom Functions and Pseudorandom Permutations in the Information-Theoretic Model. Cryptology ePrint Archive, Report 2005/135, 2005. http://eprint.iacr.org/.
19. T. Peyrin, H. Gilbert, F. Muller, and M. Robshaw. Combining Compression Functions and Block Cipher-Based Hash. In *Advances in Cryptology – Asiacrypt'06*, 2006. To appear.
20. N. Pramstaller, C. Rechberger, and V. Rijmen. Breaking a New Hash Function Design Strategy Called SMASH. In B. Preneel and S. E. Tavares, editors, *Selected Areas in Cryptography – 2005*, volume 3897 of *Lectures Notes in Computer Science*, pages 233–244. Springer, 2005.
21. B. Preneel. *Analysis and design of cryptographic hash functions.* PhD thesis, Katholieke Universiteit Leuven, 1993.
22. X. Wang, Y. Yin, and H. Yu. Finding Collisions in the Full SHA1. In V. Shoup, editor, *Advances in Cryptology – Crypto'05*, volume 3621 of *Lectures Notes in Computer Science*, pages 17–36. Springer, 2005.
23. X. Wang and H. Yu. How to Break MD5 and Other Hash Functions. In R. Cramer, editor, *Advances in Cryptology – Eurocrypt'05*, volume 3494 of *Lectures Notes in Computer Science*, pages 19–35. Springer, 2005.

A Complexity Analysis

We want to show that $P^{-1/l} < \sqrt{2}$, with $P = 1 - \left(\frac{2^l-1}{2^l}\right)^{2^l}$ and $l \geq 1$.

First, we note $f(x) = \frac{2^{x/2}-1}{2^{x/2}}$ and $g(x) = \left(\frac{2^x-1}{2^x}\right)^{2^x}$. With $x \geq 1$, both functions are increasing and if $x \longrightarrow +\infty$, we have:

$$\begin{aligned} f(x) &\longrightarrow 1 \\ g(x) &\longrightarrow \tfrac{1}{e} \end{aligned}$$

Since $f(1) > g(1)$ and $f(2) > \frac{1}{e}$, we conclude that $\frac{2^{l/2}-1}{2^{l/2}} > \left(\frac{2^l-1}{2^l}\right)^{2^l}$, with $l \geq 1$. Thus we have:

Appeared in M.S. Rhee and B. Lee (Eds.): ICISC 2006, LNCS 4296, pp. 267–285.

$$\begin{array}{cccc}
 & \frac{2^{l/2}-1}{2^{l/2}} & > & \left(\frac{2^l-1}{2^l}\right)^{2^l} \\
\Longrightarrow & 2^{l/2}-1 & > & 2^{l/2}\cdot\left(\frac{2^l-1}{2^l}\right)^{2^l} \\
\Longrightarrow & 1 & < & 2^{l/2}\cdot(1-\left(\frac{2^l-1}{2^l}\right)^{2^l}) \\
\Longrightarrow & 1 & < & 2^{l/2}\cdot P \\
\Longrightarrow & P^{-1/l} & < & \sqrt{2}.
\end{array}$$

B Unscrambling

Since the scrambling function is a PRNG, the situation here is close to the **crypt-analysis of a stream cipher**, although the number of available output bits from the PRNG is very limited here. An analysis of the PRNG was already given in [1], which works well in practice. In this section, we propose a more systematic approach to unscramble.

The scrambled message is supposed to have length 8 bytes, and is noted $m = (m_0, \ldots, m_7)$. The intermediate values of the state are noted $(seed_1^i, seed_2^i)$ for $i = 1 \ldots 8$. If we retrieve the initial state, we obtain $H(password)$ by simply XORing $H(salt)$. Unfortunately the problem is under-defined : the initial seed is 60 bits long, while the scrambled message takes only 31^9 possible values. So, in theory, there are :

$$\frac{2^{60}}{31^9} \simeq 2^{15.41}$$

solutions. Several pairs $(salt, message)$ are therefore necessary to determine uniquely $H(password)$.

B.1 Unmasking

The scrambled message m is built from 9 consecutive outputs that we note $O_0, \ldots, O_8$. We have $m_i = O_i \oplus O_8$ for $i \in \{0, \ldots, 7\}$. We first want to find O_8 in order to unmask the real PRNG outputs. We use an important flaw in the underlying PRNG, *i.e.* the fact that **all output sequences are not equally likely**. To observe this phenomenon, we look at the state update equations :

$$\begin{aligned}
seed_1^{t+1} &= (3 \cdot seed_1^t + seed_2^t) \bmod (2^{30}-1) \\
seed_2^{t+1} &= (seed_1^{t+1} + seed_2^t + 33) \bmod (2^{30}-1)
\end{aligned}$$

We **focus on the interval where $seed_1$ and $seed_2$ lie modulo** 34638622, and define :

$$N_1^t = \left\lfloor \frac{seed_1^t}{34638622} \right\rfloor \text{ and } N_2^t = \left\lfloor \frac{seed_2^t}{34638622} \right\rfloor$$

which are integers in the interval $[0, 30]$. Knowledge of N_1 and N_2 before the update allows to predict with good probability their value after the update. Indeed, :

$$3 \cdot N_1^t + N_2^t \leq \frac{3 \cdot seed_1^t + seed_2^t}{34638622} < 3 \cdot N_1^t + N_2^t + 4$$

Appeared in M.S. Rhee and B. Lee (Eds.): ICISC 2006, LNCS 4296, pp. 267–285.

Besides

$$N_1^{t+1} = \left\lfloor \frac{(3 \cdot seed_1^t + seed_2^t) \bmod 2^{30} - 1}{34638622} \right\rfloor$$
$$= \left\lfloor \frac{3 \cdot seed_1^t + seed_2^t}{34638622} \bmod 31 \right\rfloor$$

So we have only 4 candidates, *i.e.* $N_1^{t+1} = (3 \cdot N_1^t + N_2^t + i) \bmod 31$, with $i = 0, 1, 2$ or 3, instead of 31 candidates. Similarly, we can identify a reduced number of candidates for N_2^{t+1}, *i.e.* $(3 \cdot N_1^t + 2 \cdot N_2^t + i) \bmod 31$, with $i = 0, 1, 2, 3$ or 4.

The same analysis can be recursively applied to all iterations of h. Since we know the value of $O_i = N_1^i$ we can eliminate an important fraction of the initial guesses, at each step. For $i = 1$, the probability for a guess to be kept is 4 out of 31, since we have 4 candidates for O_1. It is easy to see that when we iterate the process, we get 6 candidates for O_2, ..., and 18 candidates for O_8. Therefore, for each initial guess (31 choices for O_8 and 31 choices for N_2^0), the O_i's must all be among this list of candidates, so the probability p for an initial guess to be accepted is :

$$p = \frac{4}{31} \times \frac{6}{31} \times \ldots \times \frac{18}{31} = 2^{-13.16}$$

This is small enough, since there are $31^2 \simeq 2^{9.9}$ initial guesses. So the good value of the mask O_8 should be uniquely identified by our method. We implemented this attack (see Section B.4) and indeed managed to retrieve the correct mask with high probability. In the (rare) case of a false alarm, we just do not exploit the message, and repeat the attack with another value of the salt.

B.2 Inverting the PRNG

After unmasking the scrambled message, we now have 9 consecutive outputs of the random number generator $(O_0, \ldots, O_8)$ for which we search all the possible initial seeds. Our attack is exactly similar to the unmasking attack, *i.e.* we use **interval bounding techniques**. The only difference is that we use thiner intervals, in order to learn the initial state with better precision. So we wish to split the interval $[0, 2^{30} - 1]$ into m intervals of length l. Luckily, $2^{30} - 1$ is smooth :

$$2^{30} - 1 = 3^2 \times 7 \times 11 \times 31 \times 151 \times 331$$

so we have many possible choices for m and l, such that $m \cdot l = 2^{30} - 1$. Then, we guess the initial value of

$$N_1^t = \left\lfloor \frac{seed_1^0}{l} \right\rfloor \text{ and } N_2^t = \left\lfloor \frac{seed_2^0}{l} \right\rfloor$$

which are now integers in the interval $[0, m]$ so there are m^2 possible guesses. We write :

$$3 \cdot N_1^t + N_2^t \leq \frac{3 \cdot seed_1^t + seed_2^t}{l} < 3 \cdot N_1^t + N_2^t + 4$$

and we observe that $N_1^{t+1} = \left\lfloor \frac{3 \cdot seed_1^t + seed_2^t}{l} \bmod m \right\rfloor$. Therefore we obtain 4 candidates after the first update, ..., and 18 candidates after the 8-th update (exactly like in the previous section).

Appeared in M.S. Rhee and B. Lee (Eds.): ICISC 2006, LNCS 4296, pp. 267–285.

If l is chosen small enough, these candidates all lie in the same interval modulo 34638622, so they correspond to a unique output of the PRNG. Hence, we can eliminate a fraction $31^9 \simeq 2^{44.59}$ of the initial guesses by simply testing if they correspond to a valid output sequence. Roughly, we need $18 \cdot l \ll 346386222$ for this to work. A good choice is to pick $m = 3243933$ intervals of length $l = 331$. The number of initial guesses is $m^2 = 2^{43.26}$. In theory, this is sufficient to obtain a unique candidate since $2^{44.59} > 2^{43.26}$.

B.3 Early-abort

Instead of implementing the basic attack, we used a simple refinement, which consists in **repeatedly refining the interval length** l. We start by working with $m = 31$ and $l = 34638622$ (like in the unmasking attack). We obtain a list of candidate intervals of length l where $seed_1^0$ and $seed_2^0$ may lie. Then, we increase m to $m' > m$, such that m' remains a divisor of $2^{30} - 1$.

We do not need to explore all the intervals of size $\frac{2^{30}-1}{m'}$, but only $\frac{m'}{m}$ sub-intervals, for each solution that was identified with parameters m and l. Thanks to this trick, we avoid an enumeration that would cost $(m')^2$. This **early-abort strategy** allows to dramatically improve the running time of our attack.

B.4 Experimental results

We implemented this interval bounding technique on a regular desktop PC. We started by splitting the search space into 31 intervals of length 34636833 (unmasking attack). Then, we gradually reduced the size of the intervals. Surprisingly, our experiments revealed some important deviations with the theoretical predictions. Indeed, the number of candidates at all intermediate stages (and in particular at the end of the analysis) is much higher than expected (see Table 1). In the end,

Interval length l	Number of intervals m	Average number of solutions
34636833	31	1.677
104643	10261	$723.559 \simeq 2^{9.499}$
693	1549411	$138905.785 \simeq 2^{17.084}$
63	17043521	$5788944.172 \simeq 2^{22.465}$
9	119304647	$\simeq 2^{27.842}$
3	357913941	$\simeq 2^{30.983}$
1	$2^{30} - 1$	$\simeq 2^{32.5911}$

Table 1. Experimental results averaged over 1000 different passwords

for an arbitrary output sequence, there are much more solutions for the initial seed than the theoretical prediction of :

$$\frac{2^{60}}{31^9} \simeq 2^{15.41}$$

Appeared in M.S. Rhee and B. Lee (Eds.): ICISC 2006, LNCS 4296, pp. 267–285.

while we observe on average $2^{32.5911}$ solutions. It means that there exist output sequences which are impossible, or with a small number of solutions. It would be interesting to explore further on this phenomenon. However, our attack still works fine, despite the unexpected deviations : **we manage to enumerate the $2^{32.5911}$ candidates in several minutes on a regular PC**, thanks to the early-abort strategy.

B.5 Analysis

To identify the actual hash from the $2^{32.5911}$ candidates, it is necessary to analyze several pairs $(salt, message)$. Two pairs are not sufficient, because it is very likely to observe collisions between two lists of size $2^{32.5911}$ containing 60-bit values. In theory, 3 or 4 pairs should be sufficient. However, in practice, there remains about 100 candidates, even if we consider 10 pairs. Actually, the same phenomenon was observed in [1].

The underlying reason is that the LSB of $H(password)$ have little influence on the scrambling function. Therefore, even when our prediction on the LSB is incorrect, we have still a good chance to compute the correct scrambled message. On the one hand, this property is a problem for the attacker, since he needs to compute preimages for all candidates in order to retrieve the password. On the other hand, it also means that the attacker has good chance to authenticate correctly even when $H(password)$ is invalid.

Our attack has been implemented. It takes several minutes on a typical PC, while the attack described in [1] requires about one hour running time on a PC. This attack retrieves a hash of the user's password. While this may be enough for practical purpose, it is interesting to analyze the security of the hash function.

Appeared in M.S. Rhee and B. Lee (Eds.): ICISC 2006, LNCS 4296, pp. 267–285.