Turk J Elec Eng & Comp Sci, Vol.20, No.1, 2012, © TÜBİTAK
doi:10.3906/elk-1003-412

Tradeoff tables for compression functions: how to invert hash values

Orhun KARA*, **Adem ATALAY**
TÜBİTAK - BİLGEM - UEKAE, National Research Institute of Electronics and Cryptology, PK 74 Gebze 41470 Kocaeli-TURKEY
e-mails: {orhun,adematalay}@uekae.tubitak.gov.tr

Received: 15.03.2010

Abstract

Hash functions are one of the ubiquitous cryptographic functions used widely for various applications such as digital signatures, data integrity, authentication protocols, MAC algorithms, RNGs, etc. Hash functions are supposed to be one-way, i.e., preimage resistant. One interesting property of hash functions is that they process arbitrary-length messages into fixed-length outputs. In general, this can be achieved mostly by applying compression functions onto the message blocks of fixed length, recursively. The length of the message is incorporated as padding in the last block prior to the hash, a procedure called the Merkle-Damgård strengthening. In this paper, we introduce a new way to find preimages on a hash function by using a rainbow table of its compression function even if the hash function utilizes the Merkle-Damgård (MD) strengthening as a padding procedure. To overcome the MD strengthening, we identify the column functions as representatives of certain set of preimages, unlike conventional usage of rainbow tables or Hellman tables to invert one-way functions. As a different approach, we use the position of the given value in the table to invert it. The workload of finding a preimage of a given arbitrary digest value is $2^{2n/3}$ steps by using $2^{2n/3}$ memory, where n is both the digest size and the length of the chaining value. We give some extensions of the preimage attack on certain improved variants of MD constructions such as using output functions, incorporating the length of message blocks or using random salt values. Moreover, we introduce the notion of "near-preimage" and mount an attack to find near-preimages. We generalize the attack when the digest size is not equal to the length of chaining value. We have verified the results experimentally, in which we could find a preimage in one minute for the 40-bit hash function, whereas the exhaustive search took roughly one week on a standard PC.

Key Words: *Cryptographic hash function; compression function; preimage resistance; trade-off; Hellman table; rainbow table; one-way function.*

*Corresponding author: TÜBİTAK - BİLGEM - UEKAE, National Research Institute of Electronics and Cryptology, PK 74 Gebze 41470 Kocaeli-TURKEY
The first author is supported in part by the European Commission through the project FP-7 ICE under the project grant number 206546.

1. Introduction

A cryptographic hash function H is a function mapping an infinite set of inputs to a finite set of n-bit hash values. Hash functions are one of the ubiquitous cryptographic functions used widely for various applications such as digital signatures, data integrity, authentication protocols, MAC algorithms, RNGs, stream cipher encryption, etc. One can detect any modification of a file or message transmitted by comparing hash values calculated before and after the transmission. Hash functions can also be used to uniquely identify a file content, directory trees, ancestry information, etc. Another related application is password verification. For obvious reasons, passwords are stored in digest form instead of cleartext form. The password sent by user is hashed and compared with the stored hash while authenticating a user. For both security and performance reasons, most digital signature algorithms require that only the hash of the message be "signed," not the entire message. Hash functions can also be used for the generation of pseudorandom bits.

Almost all the current practical hash functions are iterated hash functions using a compression function with a fixed-length input. H is supposed to provide the following three security criteria so as to be considered as a secure hash function:

- Collision resistance with a security level of $2^{n/2}$,
- Preimage resistance with a security level of 2^n and
- Second-preimage resistance with a security level of 2^n.

In particular, hash functions are supposed to be one-way functions, i.e., preimage resistant in almost all the applications due to the security requirements. This is the main difference between hash functions and check-sums. Preimage resistance means it must be computationally infeasible to find a preimage x for a given hash value y such that $H(x) = y$. More precisely, the overhead of finding such an x must be at least 2^n of H executions where n is the size of y (the hash size), if H is considered as a secure hash function in terms of preimage resistance. Evidently, check-sums are not preimage resistant. This security criterion is required in, for instance, password storage. Since the hash values of the passwords are stored rather than passwords themselves, it is (must be) infeasible to find valid passwords from a hash value. In another significant application, the digital signature of a hash value is a valid signature for any preimage of the hash value. So, the hash function used in digital signatures must be preimage resistant.

One of the most common methods of constructing hash functions is the Merkle-Damgård (MD) construction [1, 2]. Processing an arbitrary-length message into a fixed-length output can be accomplished by splitting the input message into fixed-length blocks and applying a compression function recursively (see Figure 1). The outputs (or, equivalently, the inputs, except the message blocks) of the compression function are called chaining values. A padding procedure is used to extend the message into a multiple of the block length. The padding involves the length of the message as well. This is called the Merkle-Damgård (MD) strengthening.

If the length of the chaining value of H is equal to the digest size, as in the case of most Merkle-Damgård constructions, then the security level of second-preimage resistance is diminished to 2^{n-k} for a given message of length 2^k. Indeed, Kelsey and Schneider proposed a method of finding second-preimage to a message of length 2^k with workload $k2^{n/2+1} + 2^{n-k+1}$ by making use of an "expandable message" [3]. Kelsey, this time with Kohno [4], exploited the relatively small chaining value size to mount the CTFP-"Chosen Target Forced Prefix" attack on MD constructions, exploiting the ease of finding multicollisions [5]. Both of the attacks use

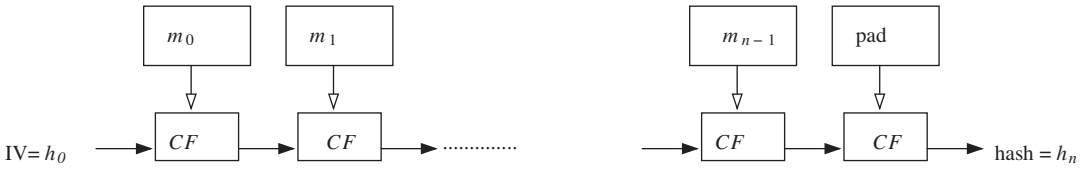

Figure 1. Merkle-Damgård (MD) Construction. CF stands for the Compression Function . The parameters h_i's are the chaining values and m_i's are the message blocks.

memory and have precomputation phase. On the other hand, there is no known structural attack yet better than brute force to find a preimage for hash functions. Hence the security level for preimage resistance has been indisputably settled for 2^n up to now. For instance, NIST requires n-bit security for preimage resistance whereas $(n-k)$-bit security is required for second-preimage resistance for a given message of length 2^k at the very recent SHA-3 competition [6].

Both the second preimage-image attack in [3] and the CTFP attack in [4] exploits the relatively small size of the chaining value in the MD construction. In both attacks, the collisions in chaining values are extended to the whole hash construction. Indeed, the workload of finding several collisions in chaining values is as much as that of the entire hash function since the chaining value and the hash value have the same size. This enables to find second preimages in less than 2^n. However, it has been not known how to exploit the relatively small size of the chaining value to find preimage for MD constructions. The main problem in finding preimages by using chaining values is the Merkle-Damgård strengthening. For example, Kelsey and Schneider introduce the notion of "expandable message" to overcome the MD strengthening in finding second preimages [3].

In this paper, we introduce a new way of exploiting a relatively small size of the chaining value of an MD construction to find a preimage to a given hash value. We make use of a rainbow table given in [7, 8] for a compression function, containing valid hash values. The hash values in the table are combined to each other through the compression function along any row of the table, even if the hash function utilizes the MD-strengthening as a padding procedure. We identify the column functions as representatives of certain set of preimages, unlike conventional usage of rainbow tables [7] or Hellman tables [8] to invert one-way functions. We introduce a new method to find a preimage which we call the "position-based inversion technique." We can immediately provide a preimage by recovering the position of a given hash value in the table by utilizing the new technique if the hash value is in the table. Indeed, we are not interested in the predecessor of the hash value in the table. This is a new way of exploiting rainbow tables.

The workload of the precomputation to prepare a table is as much as the workload of the brute force as in the case of other attacks using rainbow tables or Hellman tables. Once the table is ready, one can find a preimage of any given digest value in $2^{2n/3}$ steps by using $2^{2n/3}$ memory where n is both the digest size and the length of the chaining value.

The preimage attack through a rainbow table can particularly be a serious threat for hash functions of small digest sizes. An example where the new attack has significant results is the recent hash function DM-PRESENT-128, a construction by Bogdanov et al. [9], based on the block cipher PRESENT. DM-PRESENT-128 is a Davies-Meyer construction with 128-bit block length and 64-bit digest length, designed for the restricted environments where the property of collision resistance is not a security condition, like some applications of RFID-tags. As a practical example, a preimage to a given hash value through DM-PRESENT-128 can be found in 2^{43} PRESENT calls by using 2^{47} bytes of memory. Preparing the table costs 2^{64} PRESENT calls.

We give some extensions of the preimage attack on certain improved variants of MD constructions such as using output functions, incorporating the length of message blocks or using random salt values. Moreover, we introduce a new notion which we call "near-preimage" and mount an attack to find near-preimages. We also give some examples of applications where the near-preimage attack scenario is quite realistic. The security threshold of near-preimage resistance is much less than that of preimage resistance. Hence, the near-preimage attack through a rainbow table can be a practical attack for small-sized hash functions such as 64-bit hash functions. A near-preimage can be found in $2^{32.9}$ steps with 64.9 GB memory, assuming that a significant percentage of people do not generally notice the difference at a glance between two 16-digit vectors in hexadecimal whose first 4 digits are equal and whose number of different digits is at most five. The precomputation is $2^{34.8}$, which is much less than 2^{64}.

We suggest a security criterion so as to foil the preimage attacks we have introduced; the internal state size (the length of the chaining value) of a hash function should be at least twice as large as the digest size. This result is interesting when considering its analogous criterion on stream ciphers: the internal state size of a stream cipher must be at least twice as large as the key length.

We have verified the results experimentally by designing two hash functions having digest sizes 32-bit and 40-bit from MD5. We have prepared two tables for these hash functions. Then, we have computed the time complexities and the success rates from the experimental data collected by several tries. As a result, we have seen that we could find a preimage in one minute for the 40-bit hash function on a standard PC by using the rainbow table whereas the exhaustive search took roughly one week on the same PC.

The remainder of this paper is organized as follows. Section 2 is an overview to the Hellman tables and rainbow tables. We introduce the basic attack in Section 3 and give some extensions of the basic attack in the forthcoming section. We present the notion of "near-preimage" and describe how to find near-preimages by using the basic attack in Section 5. We discuss the security implications of the basic preimage attack in Section 6. Section 7 covers some results deduced experimentally and it turns out that the results are in parallel to the corresponding theoretical results. Finally, we conclude the paper with a discussion on the potential consequences of the attack.

2. Time-memory-tradeoff tables

There are essentially two extreme methods to invert a one-way function of n bit output. One of them is searching a preimage exhaustively. This method costs 2^n execution of the one-way function. The other method is storing all the input-output pairs of the one-way function in a table. This costs 2^n execution of the function on offline phase and 2^n units of memory. The on-line phase has a negligible time complexity.

The pioneering work by Hellman is the first example introducing a trade-off between time and memory complexities [8]. Hellman introduced tables of chains where only the starting and the end points of each chain is stored. A chain is a collection of the points obtained by applying the one-way function iteratively to a point. Then any point in any chain can be recovered and even inverted (except the starting point) with a cost of traveling on the chains. As a result, Hellman obtained the trade-off curve $M^2T = N^2$ for random functions where M is the memory allocated, T is the time complexity and N is the cardinality of the space. Remark that the curve is further improved to $MT = N$ for random permutations. The details can be found in [8].

For a given random function f, each table contains m chains where each chain has length t. A chain is constructed by applying several f functions. If s_0 is the starting point of the chain $s_0, s_1, \cdots, s_t$, then

$s_i = f^i(s_0)$ for $i = 1, \ldots, t$. Only the pair (s_0, s_t) is stored.

Each table contains m chains and hence mt points (see Figure 2). Note that any collision in a table merge. So, it is almost impossible to cover all the space in a table. Hellman proposes t tables, each containing $m \times t$ points such that $mt^2 = N$ in order to optimize the complexity and the success rate [8]. Every single table is constructed by a new function which is obtained by small manipulations of f. Hence merges by any collisions between tables are precluded. Searching each table costs t executions of f. There are t tables. Therefore, $T = t^2$ and $M = mt$. So, we get the curve $M^2T = N^2$.

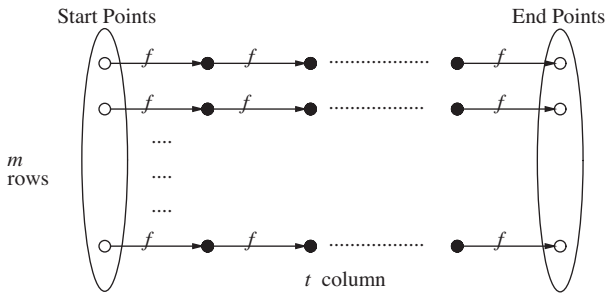

Figure 2. A diagram showing an $m \times t$ Hellman table.

One main drawback of Hellman tables is that they can not be large enough to cover the whole space. So, one needs several Hellman tables with different functions. This causes a dramatic increase on the time complexity. A new method of constructing one table is introduced by Oechslin [7]. Oechslin proposes to use a different function at each column of a table. These functions are produced by small manipulations of f. Hence, the collisions on different columns in the table do not merge. Oechslin calls these tables as rainbow tables. A rainbow table has m rows and t columns such that $mt = N$, i.e., covers all the space. However, to search one table costs $\frac{t(t+1)}{2}$ executions of f. Thus, $M = O(m)$ and $T = O(t^2)$ leading to the same curve $M^2T = N^2$, where a constant factor is disregarded.

3. Basic attack

Let H be a hash function produced by the MD construction. Let CF be its compression function. A message M is split into b-bit blocks as $M = M^1||M^2||\cdots||M^\ell$ where M^i is the i-th message block, M^ℓ is the padding block and $||$ is the concatenation. The padding is done as the MD strengthening: Add one "1" and enough number of "0"s and then the length of the message in bits. Each block is fed into the compression function CF with the current chaining value to produce the next chaining value. This is shown as

$$h_i = CF(h_{i-1}, M^i) \text{ for } i = 1, \ldots, \ell,$$

where h_0 is the initial chaining value. The hash value h of the message M is the last chaining value which is given as $H(M) = h_\ell = h$.

We mount a trade-off attack on H by using a rainbow table whose column functions are of a special type: We take the message block of the i-th column to be the i-th padding block and hence the column functions are functions of chaining values.

Let $S_1, \ldots, S_m$ be arbitrary fixed 1-block long messages. Assume that they are different from each other. Let $h_0^j = CF(h_0, S_j)$ for $1 \leq j \leq m$. Define P_i as the i-th padding which is the padding of a message having

i blocks in length and b bits in each block. That is, $P_i = 1||00\cdots0||ib$. Then, define the i-th chaining value of the j-th row as

$$h_i^j = CF(h_{i-1}^j, P_i).$$

The rainbow table is formed by the chaining values h_i^j's. The first column of the table is $h_1^1, \ldots, h_1^m$ and the last column of the table is $h_t^1, \ldots, h_t^m$. Note that each h_i^j is a valid hash value at the same time. The following statement supplies a preimage for each h_i^j.

Proposition 1 *The chaining values h_i^j's are all valid hash values and*

$$H(S_j||P_1||P_2||\cdots||P_{i-1}) = h_i^j \text{ for } 2 \le i \le t \text{ and } 1 \le j \le m$$

and $H(S_j) = h_1^j$ which corresponds to the case $i = 1$.

Proof We prove the statement by induction on i for $i = 1, \ldots, t$. In fact, when $i = 1$ we have

$$H(S_j) = CF(CF(h_0, S_j), P_1) = CF(h_0^j, P_1) = h_1^j.$$

So, h_1^j is a valid hash value and S_j is a preimage for h_1^j for any $j = 1, \ldots, m$.

Assume that the statement is true for some k where $1 \le k < t$. That is,

$$H(S_j||P_1||P_2||\cdots||P_{k-1}) = h_k^j \text{ for all } j = 1, \ldots, m.$$

Hence, h_k^j is the final chaining value of $S_j||P_1||P_2||\cdots||P_k$ since P_k is the padding of the message $S_j||P_1||P_2||\cdots||P_{k-1}$. So we have,

$$H(S_j||P_1||P_2||\cdots||P_k) = CF(h_k^j, P_{k+1}).$$

On the other hand, $CF(h_k^j, P_{k+1}) = h_{k+1}^j$ by definition. Hence, h_{k+1}^j is the hash value of the message $S_j||P_1||P_2||\cdots||P_k$. That is, the statement is also true for $k+1$. This completes the induction. □

The (i, j)-th entry of the rainbow table is h_i^j. The table is illustrated in Figure 3. The i-th column function is given as

$$f_i(h) = CF(h, P_i).$$

The main idea to utilize a rainbow table in the preimage calculation as a way of constructing the i-th column function f_i. Rainbow tables are used to invert one-way functions in general. However, we are not interested in inverting f_i's. The principal property of f_i's is that both their input and outputs are valid hash values. On the other hand, it is enough to know the position of a hash value in the table to find a preimage according to Proposition 1. We call the new technique to find a preimage by Proposition 1 as the "position-based inversion technique."

The table has m rows and t columns with $m \cdot t = N = 2^n$. On the other hand, the total memory used is $M \approx m$ and the time complexity is $T \approx t^2$. So, we have the trade-off curve $M^2 \cdot T = N^2$. The best point in terms of the minimum workload is $M = T = 2^{2n/3}$.

We do not need to store the first column unlike common rainbow tables. Only S_j's and the last column of the table, $h_t^1, \ldots, h_t^m$, are stored in the memory, sorted according to the last column. For a given hash value

h, first check whether h is in the last column. If not, then check whether $CF(h, P_t)$ is in the last column. If not, then check whether $CF(CF(h, P_{t-1}), P_t)$ is in the last column and continue in this manner. The precise description of the procedure is given in Algorithm 1. If the hash value is in the table, then it is enough to determine the position of the hash value to give a preimage.

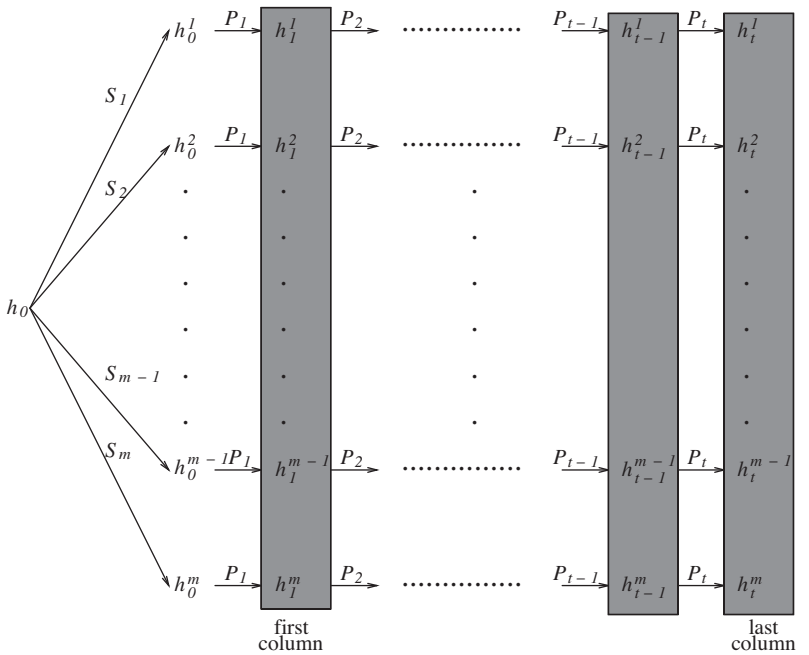

Figure 3. The rainbow table. The arrows point at the next chaining values. The next chaining values are obtained from the current chaining values shown at the back of the arrows and the message blocks shown on the arrows.

Algorithm 1: Finding a preimage of a given hash value from the rainbow table of size $m \times t$.

Data: hash value h
Result: a preimage of given hash value
for $j \leftarrow t$ **DownTo** 0 **do**
  $k \leftarrow j$;
  $CV \leftarrow h$;
  while $t > k$ **do**
    $CV = CF(CV, P_k)$ (P_k is the padding of k blocks);
    $k++$;
  if $(CV = h_t^i$ **for some** $i)$ **and** $(H(S_i||P_1||P_2||\cdots||P_{j-1}) = h)$ **then**
    return $S_i||P_1||P_2||\cdots||P_{j-1}$

If $h = h_i^j$ for some i and j, then we can detect this equality by calling the compression function $(t-i)(t-i+1)/2$ times. In this case, a preimage of h is given as

$$H(S_j||P_1||\cdots||P_{i-1}) = h$$

by Proposition 1. Therefore, in worst case, we have roughly $t^2/2$ CF queries to check whether h is in the table.

The overall cost of finding a preimage is roughly $2^{2n/3}$ steps by using roughly $2^{2n/3}$ memory.

4. Extensions of basic attack and security implications

Some extensions of the basic attack given in Section 3 can be mounted on the certain improved variants of MD constructions. We classify these variants into three groups:

- There is an additional output function applied to the final chaining value.
- Randomized hashing by using a salt value. The compression function takes a random salt value as an input as well.
- The number of message blocks (or bits) is incorporated into the compression function as well.

We explain the extensions of the basic attack in the following three subsections.

4.1. MD construction with an output function

Let the hash function H be an MD construction with a compression function CF as defined in the beginning of Section 3. Let H' be a new hash function defined as

$$H'(M) = G(H(M)) \text{ where } G : GF(2)^n \longrightarrow GF(2)^{n'}.$$

So, H' is a hash function of digest size n' and G is its output function.

Let h' be given as a hash value computed by H'. We first find an element of $G^{-1}(h')$ to find a preimage for h' through H'. Let $G(h) = h'$. If h is in the rainbow table prepared for H, then we can find a preimage for h through H which is also a preimage for h' through H'. This is a simple attack to find a preimage for H'.

We can improve the attack by searching an arbitrary element of a set in the rainbow table. Assume that the workload of finding a preimage for h' through G is C. Find a set $P_{h'}$ of preimages for h' through G of cardinality 2^k. Then, the workload of forming $P_{h'}$ is 2^kC. On the other hand, it is enough to find a preimage for one of the elements of $P_{h'}$ through H to find a preimage for h' through H'. So, use a rainbow table containing 2^{n-k} elements. That is, the number of rows times the number of columns is $m \times t = 2^{n-k}$. Then, one of the elements of $P_{h'}$ is contained in the rainbow table with a significant probability (which is approximately $1 - \exp(-1) \approx 0.63$) by the birthday paradox.

The time complexity for tracing the table is roughly $T \approx t^2 2^k$ whereas the memory is $M \approx m$. Then, $m \times t = 2^{n-k}$ gives the trade-off curve as $M^2T = 2^{2n-k}$ where the optimal point on the curve is $M = T = 2^{(2n-k)/3}$. As a result, we find a preimage in $2^{(2n-k)/3} + 2^kC$ time by using $2^{(2n-k)/3}$ memory.

Remark. Assume that C (the cost of finding a preimage through G) is negligible. For example, G may be a truncation function or a linear function from n-bit to n'-bit. Then, the time complexity is dominated by $2^{(2n-k)/3}$. Assume that $n > n'$ and take $k = n - n'$. That is, we take $P_{h'}$ as the set of all the preimages of h' through G, $P_{h'} = G^{-1}(h')$. In this case, the total complexity is $2^{(n+n')/3}$ and hence exceeds the complexity of the brute force attack, namely $2^{n'}$, if $n \geq 2n'$.

4.2. Randomized hashing

Let H' be given as

$$H'(M, r) = H(r||M^1 \oplus r|| \cdots ||M^\ell \oplus r),$$

which is the `RMX` mode proposed by Halevi and Krawczyk [10]. Here, r is a random salt value. The randomized hash can be constructed in other ways as well. For example, a randomized hash function can be generated through the HAIFA construction, proposed by Biham and Dunkelman [11]. In this case, the problem of finding a preimage for given hash value can be solved by fixing a value r_0 and then applying the attack in Section 3 when r_0 is used as a salt. On the other hand, if the problem of finding preimage is stated as finding M for given h and r such that $H'(M, r) = h$, then the basic attack in Section 3 will not work. We think that it is an interesting and challenging open problem to prepare a table which can be utilized to find a preimage for any given salt value.

4.3. When compression functions are fed by number of blocks

Assume that the number of message blocks (or bits) is incorporated into the compression function CF' of a hash function H'. That is, CF' takes the number of message blocks (or bits) hashed up to that point as the third parameter.

The attack in Section 3 can be applicable to H' if last call of CF' is not treated differently from the previous calls. That is, if CF' is applied to the final block (the padding block) same as it is applied to the previous blocks. Then, take $S_1, \ldots, S_m$ as arbitrary 1-block length different messages. Let $h_0^j = CF'(h_0, S_j, 1)$ for $1 \leq j \leq m$, where 1 indicates the length of message block hashed so far (take $h_0^j = CF'(h_0, S_j, b)$ if the length of message bits is incorporated). Define P_i as the i-th padding which is the padding of a message having i blocks in length and b bits in each block. That is, $P_i = 1||00\cdots0||ib$. Then, the i-th chaining value of j-th row of the rainbow table is given as $h_i^j = CF'(h_{i-1}^j, P_i, i+1)$ (take $h_i^j = CF'(h_{i-1}^j, P_i, b(i+1))$ if the length of message bits is incorporated). The basic attack given in Section 3 is still applicable in this case.

5. Near-preimages

Finding a message whose hash value is very close to the given value must be a difficult problem. More precisely, the workload of finding a message M for a given h such that $d(H(M), h) \leq \epsilon$ must be at least 2^{n-e} where $d()$ is the Hamming distance and 2^e is the cardinality of the ϵ-neighborhood of h (the set of values whose Hamming distances to h are at most ϵ). This security requirement is analogous to near-collision resistance and hence we call it as "near-preimage resistance."

The "near-preimage resistance" is a required security criterion in practice for some applications and cryptographic protocols where hash values are checked by human eye. For example, the users compare the hash values of the challenges of the other peer manually to authenticate each other during Diffie-Hellman key exchange protocol in some applications. Also, the hash values on a PKI certificate are checked manually in most cases. Moreover, the hash value is truncated in some protocols. near-preimages can result in preimages after truncation for these protocols.

Finding a near-preimage by using a rainbow table is similar to finding a preimage. If the rainbow table contains one of the elements h' of the ϵ-neighborhood of h, then one can find a preimage for h' immediately

through the table. The table must contain $m \times t = 2^{n-e}$ elements so as to supply a nonempty collision set with the ϵ-neighborhood of h with a significant probability by the birthday paradox. We remark that 2^{n-e} is also the time complexity of the precomputation phase. So, we gain a factor of 2^e during the preparation of the tables in near-preimage attacks. The table is searched for each element in the neighborhood. Hence, the time complexity is $T \approx t^2 2^e$. The trade-off curve is given as $M^2T = 2^{2n-e}$. Then, $M = T = 2^{(2n-e)/3}$ is the optimal point on the curve.

If the alphabet is the binary alphabet, then the number of elements of the ϵ-neighborhood of a value is given in logarithm as

$$e = \log_2 \left(\sum_{i=0}^{\epsilon} \binom{n}{i} \right).$$

In general, the alphabet may contain z elements. Also it may be required that first f digits contain no error since the first digits are checked more attentively in the manual comparison. In this case, the number of elements of the ϵ-neighborhood of a value in logarithm, denoted by $e = e(z, f, \epsilon)$, is given as

$$e = e(z, f, \epsilon) = \log_2 \left(\sum_{i=0}^{\epsilon} (z-1)^i \binom{n-f}{i} \right).$$

Example. Consider a digest size of 64-bit. Let any digest value be checked by human eye over its 16-digit hex value. Assume that a significant percentage of people do not generally notice the difference at a glance between two 16-digit vectors in hexadecimal whose first 4 digits are equal and whose number of different digits (Hamming distance) is at most five. Then, for a given hash value h, one can defeat the security of the system by finding a message M such that $d(H(M), h) \leq 5$ in hex and first four digits of $H(M)$ and h are equal. Then it can be perceived that $H(M)$ and h are equal, which may falsely bring about a successful authentication. The number of elements of the ϵ-neighborhood of h in logarithm, denoted as $e = e(16, 4, 5)$, is given as

$$e = e(16, 4, 5) = \log_2 \left(\sum_{i=0}^{5} 15^i \cdot \binom{12}{i} \right) \approx 29.2.$$

So, the on-line phase costs $2^{32.9}$ steps with 64.9 GB memory to recover the message M, within quite practical limits. This is an interesting example combining exploiting human behavior with cryptanalysis.

6. Security implications

We have examined the extensions of basic preimage attack on several improved variants of the MD construction in the previous sections. It turns out that the basic attack is still better than the brute force attack in some circumstances.

Recall that both the time complexity and the memory complexity of the preimage attack is $2^{(n+n')/3}$ when the n-bit hash value is shortened to n'-bit by a simple function. So, the preimage attack is better than the brute force attack if $n < 2n'$. Therefore, we have a new security criterion in order to foil our attack: The length of the chaining value should be at least twice as large as the length of the digest value. This is indeed a surprising result. The analogous result is given for stream ciphers so as to make them resistant against certain trade-off attacks (e.g. [12, 13]) and adopted by a design criterion for security.

7. Success rates and experimental results

We designed two toy MD constructions with digest sizes 32-bit and 40-bit in order to verify the basic attack given in Section 3. The length of the message blocks was 128-bit for both of the hash functions. These hash functions were deduced from MD5 and behaved as random functions in our statistical tests.

The success rate of a table (the probability that an arbitrary hash value is in the table) is given as

$$P_s(m,t) = 1 - \prod_{i=1}^{t} \left(1 - \frac{m_i}{2^n}\right)$$

where m_i is the number of different elements in the i-th column. This probability is bounded above by $1-(1-m/2^n)^t$ which is achieved when $m_i = m$ for all $i = 1, \ldots, t$ (such a table is called a "perfect table"). On the other hand,

$$1 - (1 - \frac{m}{2^n})^t \approx 1 - \exp(-\frac{mt}{2^n})$$

which is approximately $1 - \exp(-1) \approx 0.63$ when $mt \approx 2^n$. This success rate diminishes to 0.55 for arbitrary tables (see [8]). The success rates were 0.556 among 15698 tries and 0.565 among 1414 tries, as expected, for the 32-bit hash and for the 40-bit hash respectively in the experiment.

The probability that a near-preimage of an arbitrary hash value is in the perfect table is given as

$$1 - (1 - \frac{2^e m}{2^n})^t \approx 1 - \exp(-\frac{2^e mt}{2^n})$$

since $mt \approx 2^{n-e}$ this value is approximately equal to $1 - \exp(-1) \approx 0.63$. In our experiments, we ran 10000 tries and we found 6411 near-preimages whose distances are less than or equal to 1. The experimental success rate is 0.641 as expected since we use a much smaller table having almost no internal collisions.

A rough estimation for the time complexity is given as $T \approx t^2$ where the constants and the overhead due to false alarms is disregarded. In a recent study, a rigorous formula for the average time complexity for one rainbow table is given as [14]

$$\begin{aligned} T_{av} &= \frac{m}{2^n} \sum_{k=1}^{t} \left(1 - \frac{m}{2^n}\right)^{k-1} \\ &\cdot \left(\frac{k(k-1)}{2} + \sum_{i=t-k+1}^{t} i \left(1 - \frac{m}{2^n} - \frac{i(i-1)}{t(t-1)}\right)\right) \\ &+ \left(1 - \frac{m}{2^n}\right)^t \\ &\cdot \left(\frac{t(t-1)}{2} + \sum_{i=1}^{t} i \left(1 - \frac{m}{2^n} - \frac{i(i-1)}{t(t-1)}\right)\right). \end{aligned} \quad (1)$$

In our experiment, we had two cases: $(n,m,t) = (32, 2^{21}, 2^{11})$ and $(n,m,t) = (40, 2^{26}, 2^{14})$. The theoretical average time complexities for these cases are $2^{19.25}$ and $2^{25.25}$ respectively according to Equation 1. The experimental results verified these numbers. Table 1 summarizes the results. For example, the average time complexity for the 40-bit hash is $2^{25.5}$ in the table, which is very close to the theoretical result, $2^{25.25}$. On the

Table 1. Experimental results. SA: Among Successful Attempts. UA: Among Unsuccessful Attempts. TA: Among Total Attempts. ATC: Average Time Complexity. ANFA: Average Number of False Alarms.

	N	SA	UA	TA
ATC	2^{32}	$2^{18.6}$	$2^{19.9}$	$2^{19.3}$
	2^{40}	$2^{24.8}$	$2^{26.1}$	$2^{25.5}$
ANFA	2^{32}	$2^{8.3}$	2^{10}	$2^{9.3}$
	2^{40}	$2^{11.3}$	2^{13}	$2^{12.3}$

other hand Table 1 shows the average number of false alarms both for 32-bit and for 40-bit hash values. For example, the average number of false alarms for the 32-bit hash is $2^{9.3}$.

A false alarm occurs whenever the first part of the if statement in Algorithm 1 holds and the second part does not hold. This corresponds to the fact that chains starting at different chaining values collide and merge. In general, the expected number of false alarms is bounded above by $mt(t+1)/2^{n+1}$ [8]. For our case, $mt = 2^n$. Hence, the average number of false alarms in the worst case (all the table is searched) is $(t+1)/2$. The average number of false alarms is given in Table 1.

8. Conclusion and discussion

We have developed a new way of constructing rainbow tables for compression functions, containing valid hash values in order to find preimages. In this way, we can reduce the security level to $2^{2n/3}$ for conventional MD constructions where n is both the length of the chaining value and the digest size. On the other hand, one significant drawback of the rainbow tables is that the precomputation step costs as much as the workload of brute force. However, precomputation is performed offline and only once.

We have introduced the "position-based inversion technique" to overcome the MD strengthening and hence utilize the outputs of a compression function as valid hash values while preparing the rainbow tables. In this way, the hash values in the table are combined to each other through the compression function along any row of the table. We have developed an algorithm (Algorithm 1) to recover the position of a given hash value in the table and to provide a preimage from its position in the table. The connection between the position of a point and its preimage is given in Proposition 1. This property is the main difference between our rainbow tables and the conventional rainbow tables or Hellman tables and helps in overcoming MD strengthening. Indeed, the position does not help to invert the point in the conventional usage of the rainbow tables.

We have extended our attack on several variants of MD constructions such as using output functions, incorporating the length of message blocks or using random salt values. We have calculated the complexity of finding preimages when these new parameters were introduced. Moreover, we have introduced a new notion which we call "near-preimage" and mounted an attack to find near-preimages. Moreover, we have given some examples of applications where the near-preimage attack scenario is quite realistic. The security threshold of near-preimage resistance is much less than that of preimage resistance. Hence, the near-preimage attack through a rainbow table can be a practical attack for small-sized and moderate-sized hash functions.

We have verified the results experimentally by designing two hash functions having digest sizes 32-bit and 40-bit from MD5. We have prepared two tables for these hash functions. Then, we have computed the

time complexities and the success rates from the experimental data collected by several tries.

It may be argued that rainbow tables can never be sufficiently effective in practice for large digest sizes. It is simply impossible to prepare a table of size, for instance 2^{512}. On the other hand, not all the hash function applications require large digest sizes. Indeed, rainbow tables can be real threats for some hash functions of small digest sizes to find preimages with workloads within practical bounds.

Several ubiquitous applications of cryptographic schemes using hash functions of small digest sizes have emerged recently both in the literature and in industry. Some of the electronic payment schemes, especially micropayment schemes such as "PayWord" and "MicroMint" [15], "Millicent" [16] and "NetBill" [17], require small sized hash functions due to efficiency reasons. Moreover, the authentication protocols based on human comparison of short strings in pervasive computing utilizes hash functions of small digest sizes.

One pervasive application domain is the usage of RFID-tag deployed devices. Some of the RFID schemes make use of small hash functions to provide privacy and authentication [18, 19]. In particular, the security protocols in many tag based applications do not require the property of collision resistance. The security is based on one-wayness property. For example, user privacy is protected through hash chain mechanism, which requires particularly preimage resistance, in some RFID schemes [18]. DM-PRESENT-128 is one of the recent hash functions designed for such protocols [9]. DM-PRESENT-128 is a Davies-Meyer construction based on the block cipher PRESENT. It has 128-bit block size and 64-bit digest size. Hence, it is possible to find preimage in approximately 2^{43} PRESENT calls with 2^{47} bytes of memory, quite within the practical domain after preparing the rainbow table even though it has relatively large block length.

There may be essentially two ways of foiling the preimage attack we have presented. The first way is that the length of the chaining value (the internal state size) must be at least twice as large as the length of the digest size. The results in [20] due to Lucks also support this argument. Indeed, this criterion is even necessary to supply full security against finding a second preimage to a given message of arbitrary length. In this manner, the brute force of finding a second preimage for an n-bit hash function costs 2^n steps rather than 2^{n-k} steps for a given message of length 2^k. Nevertheless, some parts of the cryptology community deem 2^{n-k} as the security level of second preimage resistance such as NIST for SHA-3 contest [6].

The second way of avoiding the attack is introducing a random salt and determining the initial chaining value by the random salt before incorporating any message into a compression function. However, randomized hash functions may not be compatible with some protocols using hash functions having only one parameter.

References

[1] R.C. Merkle. One way hash functions and des. In G. Brassard, editor, *Advances in Cryptology, CRYPTO 1989*, volume 435 of *Lecture Notes in Computer Science*, pages 428–446. Springer, 1989.

[2] I. Damgård. A design principle for hash functions. In G. Brassard, editor, *Advances in Cryptology, CRYPTO 1989*, volume 435 of *Lecture Notes in Computer Science*, pages 416–427. Springer, 1989.

[3] 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 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 474–490. Springer, 2005.

[4] J. Kelsey and T. Kohno. Herding hash functions and the nostradamus attack. In S. Vaudenay, editor, *Advances in Cryptology, EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 183–200. Springer, 2006.

[5] A. Joux. Multicollisions in iterated hash functions. In M.K. Franklin, editor, *Advances in Cryptology, CRYPTO 2004*, volume 3152 of *Lecture Notes in Computer Science*, pages 306–316. Springer, 2004.

[6] NIST. Cryptographic hash algorithm competition, 2007. URL http://csrc.nist.gov/groups/ST/hash/sha-3/index.html.

[7] P. Oechslin. Making a faster cryptanalytic time-memory trade-off. In D. Boneh, editor, *Advances in Cryptology, CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 617–630. Springer, 2003.

[8] M. E. Hellman. A cryptanalytic time-memory trade-off. *IEEE Trans. on Information Theory*, 26(4):401–406, 1980.

[9] C. Paar A. Bogdanov, G. Leander, A. Poschmann, M. J. B. Robshaw, and Y. Seurin. Hash functions and rfid tags: Mind the gap. In Oswald and P. Rohatgi, editors, *CHES 2008*, volume 5154 of *Lecture Notes in Computer Science*, pages 283–299. Springer, 2008.

[10] S. Halevi and H. Krawczyk. Strengthening digital signatures via randomized hashing. In C. Dwork, editor, *Advances in Cryptology, CRYPTO 2006*, volume 4117 of *Lecture Notes in Computer Science*, pages 41–49. Springer, 2006.

[11] E. Biham and O. Dunkelman. A framework for iterative hash functions: Haifa. In *Second NIST Cryptographic Hash Workshop*, 2006.

[12] S. Babbage. Improved exhaustive search attacks on stream ciphers. In *European Convention on Security and Detection*, number 408 in IEE Conference publication, pages 161–166. IEE, 1995.

[13] J. Golić. Cryptanalysis of alleged a5 stream cipher. In W. Fumy, editor, *EUROCRYPT'97*, volume 1233 of *Lecture Notes in Computer Science*, pages 239–255. Springer, 1997.

[14] P. Junod G. Avoine and P. Oechslin. Characterization and improvement of time-memory trade-off based on perfect tables. *ACM Trans. Inf. Syst. Secur.*, 11(4), 2008. URL http://doi.acm.org/10.1145/1380564.1380565.

[15] R.L. Rivest and A. Shamir. Payword and micromint: Two simple micropayment schemes, 2001. URL http://people.csail.mit.edu/rivest/RivestShamir-mpay.pdf.

[16] M. S. Manasse. Millicent (electronic microcommerce), 1995. URL http://www.research.digital.com/SRC/personal/Mark_Manasse/uncommon/ucom.html.

[17] NETBILL. The netbill electronic commerce project, 1995. URL http://www.ini.cmu/NETBILL/home.html.

[18] K. Suzuki M. Ohkubo and S. Kinoshita. Cryptographic approach to "privacy- friendly" tags. In *RFID Privacy Workshop*, MIT, USA, 2003.

[19] E. Dysli G. Avoine and P. Oechslin. Reducing time complexity in rfid systems. In B. Preneel and S. Tavares, editors, *SAC 2006 Selected Areas in Cryptography*, volume 3897 of *Lecture Notes in Computer Science*, pages 291–306. Springer, 2006.

[20] S. Lucks. A failure-friendly design principle for hash functions. In B. Roy, editor, *Advances in Cryptology, ASIACRYPT 2005*, volume 3788 of *Lecture Notes in Computer Science*, pages 474–494. Springer, 2005.