Discarding the Endpoints makes the Cryptanalytic Time-Memory Trade-Offs even Faster

Gildas Avoine, Adrien Bourgeois, Xavier Carpent

Université catholique de Louvain
B-1348 Louvain-la-Neuve
Belgium

Abstract. Cryptanalytic time-memory trade-offs were introduced by Hellman in 1980 in order to perform key-recovery attacks on cryptosystems. A major advance was presented at Crypto 2003 by Oechslin, with the *rainbow table* variant that outperforms Hellman's seminal work.
This paper introduces the *fingerprint tables*, which drastically reduce the number of false alarms during the attack compared to the rainbow tables. The key point of our technique consists in storing in the tables the *fingerprints* of the chains instead of their endpoints. The fingerprint tables provide a time-memory trade-off that is about two times faster than the rainbow tables on usual problem sizes. We experimentally illustrate the performance of our technique, and demonstrate that it is faster than Ophcrack, a Windows LM Hash password cracker considered so far to be the fastest one ever implemented.

Keywords: Cryptanalysis, Time-memory trade-off, Rainbow tables

1 Introduction

1.1 Motivations

A *cryptanalyic time-memory trade-off* (TMTO) is a technique introduced by Hellman in 1980 [14] that allows an adversary to practically manage brute-force attacks. A TMTO particularly makes sense when a known plaintext attack is performed more than once, with a key-space that does not allow the adversary to store all the pairs (key, ciphertext). A TMTO is also quite relevant when the attack is time-constrained once the adversary receives the ciphertext.

Cryptanalytic time-memory trade-offs are thus the keystone of many practical attacks, for example against A5/1 (GSM) in 2000 [6], LILI-128 in 2002 [23], Windows LM Hash in 2003 [21], Unix passwords in 2005 [19], and Texas Instruments DST in 2005 [8]. In 2010, the attack against A5/1 was resurrected and a world-wide distributed TMTO-based attack was launched during Black Hat 2010 [20]. The weakness exploited in all these cases is twofold: the key entropy is not high enough, and a known (constant) plaintext attack is feasible.

In spite of the wide use of cryptanalytic time-memory trade-offs, few significant advances have been done since Oechslin introduced the rainbow tables at Crypto 2003 [21], illustrated with the instant cracking of alphanumerical Windows LM Hash passwords. Yet, any time-improvement may render attacks more practical, especially when they are time-constrained.

1.2 Background

A Hellman-type cryptanalytic time-memory trade-off consists of a *precomputation phase* that is done once, and an *online phase* that is expected to be much faster than a brute-force attack.

Precomputation Phase. To invert a function $h : A \to B$, a set of m chains of t elements in A is precomputed. A chain starts with an arbitrary value S_j belonging to A, known as *starting point*. Each subsequent element is computed by iterating a function $f : A \to A$, $x \mapsto r(h(x))$, where $r : B \to A$ is a *reduction function*, which aim is to assign an arbitrary value in A to any value of B. The i-th ($1 \leq i \leq t$) element of the j-th ($1 \leq j \leq m$) chain is denoted $X_{j,i}$, where $X_{j,i+1} = f(X_{j,i})$ and $X_{j,1} = S_j$. For the sake of simplicity, the values $X_{j,t}$ are denoted E_j and called the *endpoints*. The trick of the TMTO technique consists in only storing the m pairs (S_j, E_j) in a so-called *table*. As highlighted by Theorem 1, a single table cannot fully cover the set A, and several tables are consequently required to reach a success rate close to 1.

Online Phase. Given a value y in B, the goal of the online phase is to retrieve x in A such that $h(x) = y$. As the table does not contain the intermediate points, the adversary computes instead a chain from y and searches for a match with an endpoint. More precisely, she starts by reducing y and searching through the m endpoints. If there exists j such that $r(y) = E_j$, then she re-computes $X_{j,t-1}$ from S_j and verifies whether $h(X_{j,t-1}) = y$. If this holds, the attack succeeds and $x = X_{j,t-1}$. Otherwise, the match is called a *false alarm*. When a false alarm occurs, or when no matching endpoint is found, the attack proceeds on the next column: the attacker computes $y \leftarrow f(y)$ and repeats the operations. The attack goes on until a correct value is found, or until the end of the table is reached, in which case the attack fails.

1.3 State of the art

A major drawback of Hellman's method is that two colliding chains in a given table lead to a *fusion*. Such artifacts substantially decrease the trade-off performance. Two significative improvements have been introduced to mitigate this problem: the *distinguished points* in 1982 by Rivest [11] and the *rainbow tables* in 2003 by Oechslin [21].

Distinguished points. In this variant due to Rivest [11], the precomputation of a chain stops once a point matches a certain criterion – for instance its last ten bits are zeroes – instead of computing chains of fixed length. Such a point that matches a certain criterion is called a *distinguished point*. This technique offers two advantages. First of all, it allows to build tables where fusions can be easily discarded, called *perfect tables*. Moreover, the number of lookups in the table during the online phase is divided by t in comparison with Hellman's tables, because a lookup is performed only when a distinguished point is reached. A direct consequence is that the number of false alarms decreases. However, the chain length is no longer constant, which increases the cost to rule out the false alarms[1]. Several papers analyze the distinguished points method [9, 24] or present FPGA implementations [24].

Rainbow tables. Oechslin introduced in 2003 [21] an improvement that outperforms the distinguished points. In this variant, a different reduction function is used per column, which leads to the so-called rainbow tables. This new organization of the tables eases the detection of fusions, while keeping constant the chain length, and also divides the number of lookups by a factor t in comparison with Hellman's tables. The online phase is similar to the one of Hellman's method, but the difference is that it is necessary to start the online chains iteratively from the last column at the right of the table to the first column at the left of the table. Indeed, as the column that contains the expected key is unknown, the first reduction function that must be applied is unknown as well, and each possibility must be tested. This means that in the worst case, $t^2/2$ operations are necessary to browse all the table, without taking false alarms into account. A

[1] Given that the length of the chains is not known, a matching endpoint requires to recompute the chain until the expected value x is reached or up to a given upperbound.

thorough analysis of the rainbow tables have been done by Avoine, Junod, and Oechslin [2]. The rainbow tables are currently used by most of the password crackers, and have been implemented by Mentens, Batina, Preneel, and Verbauwhede [19] using FPGAs to retrieve UNIX passwords.

In 2005, Avoine, Junod, and Oechslin [1] introduced a new feature to the rainbow tables, known as the *checkpoints*. The authors observed that more than 50% of the cryptanalysis time is devoted to rule out the false alarms. Their technique consists in storing information (e.g., parity bit) on some intermediate points of the chains alongside the endpoints. During the online phase, when a match with an endpoint occurs, its checkpoints must be compared with the checkpoints of the chain whose construction is ongoing. When there is a match of the points, but no match of the checkpoints, the adversary can conclude that a false alarm occurred without re-computing the colliding chain. Although the checkpoints increase the performance of the trade-off, their impact is limited given that their storage consumes additional memory.

Other works. Hellman's method has been the subject of several minor improvements. Kusuda and Matsumoto explain in [17] a method to optimize the parameters of the original time-memory trade-off introduced by Hellman [14], and present a lower bound for computing its probability of success. This bound was later refined by Ma and Hong in [18]. Kim and Matsumoto propose in [16] an improvement that increases the success probability of Hellman's trade-off without increasing the online search time nor the size of the tables.

A variant of distinguished points, variable distinguished points, was presented in [15] by Hong, Jeong, Kwon, Lee and Ma. The performance of this variant is said to be "on par" with other existing solutions, but no average-case analysis has ever been performed.

Hellman's technique is designed to invert random functions. Fiat and Naor provide in [12] a construction for inverting any function, at the price of a less efficient trade-off. De, Trevisan and Tulsiani propose in [10] a similar construction for inverting any function on a fraction of their input. They also suggest using time-memory trade-offs for distinguishing the ouput of pseudorandom generators from random.

Time-memory trade-offs have also been applied to stream cipher independently by Babbage in [3] and Golić in [13]. These trade-offs typically require more data than attacks on block ciphers. Biryukov and Shamir improve in [5] the efficiency of this attack on stream ciphers by combining the Babbage/Golić and the Hellman techniques. Finally, Biryukov, Mukhopadhyay and Sarkar generalize these different approaches in [7], and propose a way to use Hellman's technique with multiple data.

In [4], Barkan, Biham, and Shamir showed that the performance of existing time-memory trade-offs can not be improved by more than a logarithmic factor.

1.4 Contributions

In this paper, we revisit the rainbow tables and introduce the *fingerprint tables*, which are approximately two times faster than the rainbow tables. The keystone of the fingerprint tables is that the endpoints are no longer stored in the table. Instead, a *fingerprint* of each chain is stored along with the starting point.

The fingerprint of a chain is the concatenation of the outputs of *ridge functions* applied to every point of the chain. A typical ridge function outputs a truncation of the input point (e.g. its least significant bit), and each column can use a different ridge function. The truncation can be drastic, including discarding all the bits of the input point – which is the case in practice for most of the columns.

In typical configurations, *two thirds of the cost due to false alarms can be saved* thanks to the fingerprints. Also, fingerprints are typically shorter than the endpoints found in rainbow

tables, which saves additional memory. These two characteristics make the fingerprint tables approximately *twice as fast as the rainbow tables*.

The fingerprint tables are described in Sect. 2 and their analysis is provided in Sect. 3. The theoretical evaluation is then applied to several configurations in Sect. 4 in order to illustrate the practical impact of the fingerprint tables. Finally, Sect. 5 presents a comparison between our theoretical evaluations and practical measures, and compare them.

2 Fingerprint Tables

If we defining the *characterization* of a chain as the information stored in the table that allows to distinguish it from another chain, the characterization of a chain in a rainbow table is its endpoint. Consequently, when a chain computed during the online phase merges with a different precomputed chain, they cannot be distinguished. This leads to the false alarms, which are the pet hate of the time-memory trade-offs. The fingerprint tables drastically reduce this problem by providing a better way to characterize the chains.

2.1 Chain Characterization

The keystone of the fingerprint tables is to replace the endpoints in the rainbow tables with a better chain characterization, called fingerprint. Such a table contains pairs of (starting point, fingerprint). We define a fingerprint F_j as the concatenation of the outputs of ridge functions Φ_i applied to each element $X_{j,i}$ of the chain:

$$F_j = \Phi_1(X_{j,1}) \,||\, \Phi_2(X_{j,2}) \,||\quad \dots \quad ||\, \Phi_t(X_{j,t})$$

where $j \in [1, m]$, "||" denotes the concatenation, and Φ_i $(1 \leq i \leq t)$ is the ridge function used in column i. A ridge function is such that:

$$\Phi_i : A \to \begin{cases} \{0,1\}^{\sigma_i} & \text{if } \sigma_i > 0 \\ \epsilon & \text{otherwise} \end{cases}$$

with $0 \leq \sigma_i \leq \lceil \log_2 N \rceil$ where N is the size of the problem. The output of the ridge function is called a *ridge*. A *chain fingerprint is thus the concatenation of the ridges of its points*. Note that a ridge function is expected to have a uniform distribution of its output, as it is the case with reduction functions.

Example 1. The results provided in Tab. 3(c) in the case $N = 2^{48}$ have been obtained using 11 ridge functions that output the least significant bit of their input, 1 ridge function that outputs the 32 least significant bits of its input, and 2.98×10^6 (all the others) ridge functions that output nil.

2.2 Precomputation and Search

Precomputation Phase. The fingerprint tables are precomputed similarly to what is done with perfect rainbow tables, but a ridge function is applied to each point of the chains during their precomputation. Starting points, fingerprints, and endpoints are then *temporarily* stored. Once the chains are all precomputed, they are cleaned to make perfect tables[2]. The endpoints are finally stripped from the table and the chains are sorted according to their fingerprints[3].

[2] A table is perfect when it does not contain any merging chains. As a side effect, an endpoint cannot appear twice in a perfect table.

[3] The cost of the final sort is marginal given that it is done once the table is perfect.

Online Phase. The online phase for the fingerprint tables differs from the rainbow tables, given that a look-up in a rainbow table to retrieve a matching endpoint is replaced by a look-up to retrieve a matching *partial* fingerprint. The fingerprint comparison is done partially since the online chain is only partially computed. Note that a fingerprint can appear more than once in a table contrarily to an endpoint.

3 Analysis

This section provides a thorough analysis of the fingerprint tables, and demonstrates that rainbow tables are a special case of fingerprint tables that is not optimal.

3.1 Notation

For the sake of clarity, the notations provided in [2] and [21] for rainbow tables are also used below, and summarized in Tab. 1.

Table 1: Notations used in this paper.

Symbol	Meaning
$h : A \to B$	The function to invert
N	$\lvert A \rvert$
m	Number of chains in one table
t	Number of columns per table
ℓ	Number of tables

3.2 Success Rate

Since the fingerprint tables are perfect, most of the results from [2] still hold. In particular, the success probability of both fingerprint tables and rainbow tables only depends on m, N, ℓ, and t. Theorem 1 is thus directly obtained from the results (Theorem 2) available in [2].

Theorem 1. *The success probability of a set of ℓ fingerprint tables is*

$$P^* = 1 - \left(1 - \frac{m}{N}\right)^{\ell t}.$$

When using tables of maximum size, this can be approximated by $P^* \approx 1 - e^{-2\ell}$.

Proof. See Theorem 2 in [2].

3.3 Complexity

A major advantage of the fingerprint tables is the significant reduction of the number of false alarms. In order to precisely analyze the average performance of the fingerprint tables, we introduce the probability ϕ_c that two different points have the same ridge in a given column c.

$$\phi_c := \Pr\left[\Phi_c(x) = \Phi_c(y) | x \neq y\right].$$

Theorem 2. *If $N \equiv 0 \pmod{2^{\sigma_c}}$, the probability that two different values have the same ridge in a given column c is:*

$$\phi_c = \frac{N/2^{\sigma_c} - 1}{N - 1}. \tag{1}$$

Proof. Given a value x and its corresponding ridge $\Phi_c(x)$, there are $N-1$ different possible values $y \neq x$. Among those, there are on average $N/2^{\sigma_c}-1$ values y such that $\Phi_c(x) = \Phi_c(y)$. □

This theorem assumes that both the points in a column and the ridges are uniformly distributed. This is for instance ensured if the reduction and ridge functions are modulos, and if the h function has a uniformly distributed output, which is the case in virtually all practical cases. Note that this assumption is already made in previous analyses, such as [2] and [21].

Theorem 3. *The average amount of evaluations of h during the online phase using the fingerprint tables is:*

$$T = \sum_{k=1}^{\ell t} \frac{m}{N}\left(1-\frac{m}{N}\right)^{k-1}(W_k + Q_k) + \left(1-\frac{m}{N}\right)^{\ell t}(W_{\ell t} + Q_{\ell t}), \tag{2}$$

with

$$c_i = t - \left\lfloor \frac{i-1}{\ell} \right\rfloor, \qquad q_c = 1 - \prod_{i=c}^{t}\left(1-\frac{m_i}{N}\right),$$

$$W_k = \sum_{i=1}^{k}(t-c_i), \qquad P_c = \sum_{i=c}^{t}\left[\prod_{j=c}^{i-1}\phi_j\right](q_i - q_{i+1}),$$

$$Q_k = \sum_{i=1}^{k}(c_i-1)(P_{c_i} + E_{c_i}), \qquad E_c = (m-q_c)\prod_{i=c}^{t}\phi_i.$$

Proof. Given the online phase described in Sect. 2, we have:

$$T = \sum_{k=1}^{\ell t} p_k T_k + p_{\text{fail}} T_{\ell t}, \tag{3}$$

with p_{fail} and p_k the probabilities that the attack fails, and that it succeeds after k steps respectively. Here, T_k denotes the average amount of evaluations of h when the attack stops after k steps.

Determination of p_k and p_{fail}:
The probability that the current point is found in a column is m/N, and p_k is the probability that the current point is found in a column, but is not found in the preceding $k-1$ searches:

$$p_k = \frac{m}{N}\left(1-\frac{m}{N}\right)^{k-1}. \tag{4}$$

The probability of failure p_{fail} is simply the probability that the current point is not found in any of the ℓt previous searches, that is:

$$\left(1-\frac{m}{N}\right)^{\ell t}. \tag{5}$$

Determination of T_k:
At each step, h is computed for the following reasons: when building the online chain (we denote this work by W), and when filtering false alarms (noted Q), hence $T_k = W_k + Q_k$.

Determination of W_k:
At step k, we begin the fingerprint comparison at column

$$c_k = t - \left\lfloor \frac{k-1}{\ell} \right\rfloor .$$

The number of h computations needed for the online chain is the number of columns separating c_k from t, that is $t - c_k$. Consequently,

$$W_k = \sum_{i=1}^{k} (t - c_i). \tag{6}$$

Determination of Q_k:
Similarly, for each false alarm, a chain from column 1 to column c_k has to be computed, needing $c_k - 1$ computations. Hence, we have:

$$Q_k = \sum_{i=1}^{k} (c_i - 1) F_{c_i}, \tag{7}$$

with F_c the average number of false alarms at column c. False alarms in fingerprint tables are of two types. Type-I false alarms are the same as the ones in rainbow tables and occur because of merges induced by reduction functions. Type-II false alarms occur because two chains may have the same partial fingerprint. We will respectively note P_c and E_c the average number of Type-I and Type-II false alarms at column c.

Determination of P_c:
Because tables are perfect, there can be at most one Type-I false alarm per step. This false alarm, if it exists, is due to a chain merging with the online chain, somewhere between c and t. Moreover, for this chain to cause a false alarm, the partial fingerprint must match before the merge as well. We thus have:

$$P_c = \sum_{i=c}^{t} \Pr(\text{same ridges between } c \text{ and } i-1 \mid \text{merge in } i) \Pr(\text{merge in } i).$$

Since the merge occurs in i, all points between c and $i-1$ differ, and therefore the first factor of each term is simply $\prod_{j=c}^{i-1} \phi_j$. The probability of having a merge between columns c and t, already identified in [2], is $q_c = 1 - \prod_{i=c}^{t} \left(1 - \frac{m_i}{N}\right)$. The probability of having a merge exactly in column i is thus $q_i - q_{i+1}$, which yields:

$$P_c = \sum_{i=c}^{t} \left[\prod_{j=c}^{i-1} \phi_j \right] (q_i - q_{i+1}) . \tag{8}$$

Determination of E_c:
Considering that the probability that a chain merges is q_c, there are, on average, $m - q_c$ non-merging chains. Among those, each creates a Type-II FA with probability $\prod_{i=c}^{t} \phi_i$, which gives:

$$E_c = (m - q_c) \prod_{i=c}^{t} \phi_i. \tag{9}$$

Using equations (4), (5), (6), (7), (8), and (9) into (3) gives (2), allowing us to conclude. □

3.4 Fingerprint Tables: a Generalization of Rainbow Tables

The differences between the rainbow tables and the fingerprint tables can be summarized as follows: (i) the size of the fingerprints is not entirely determined by the size of the problem, they can be thus smaller or greater than $\lceil \log_2 N \rceil$, (ii) the information contained in the fingerprints can be related to any point of the chain. A rainbow table is actually equivalent to a fingerprint table where $\Phi_t(X) = X$ and $\sigma_i = 0$ $(1 \leq i < t)$. Consequently Equation (2) also generalizes the average case performance for rainbow tables. The special case of rainbow tables gives:

$$\begin{cases} \phi_i = 1 & \forall i < t \\ \phi_t = 0, \end{cases}$$

which leads to $E_c = 0$ and $P_c = q_c$. This is indeed the formula established by Avoine, Junod, and Oechslin in [2]. Sect. 4 shows that this configuration is not the optimal one, which justifies that rainbow tables are less efficient than fingerprint tables when the latter are used in their optimal configuration.

4 Analytical Performance

4.1 Theoretical Results

This section addresses the expected number of evaluations of the function h using the fingerprint tables and the rainbow tables for different problem parameters. Table 2 presents the gain $1 - T_{\text{fingerprint}}/T_{\text{rainbow}}$ for different sizes of key space and memory dedicated to the trade-off. The value $\ell = 4$ is considered in both cases, and m and t are set for rainbow tables to the optimal values as presented in [2]. For the fingerprint tables, a *Hill Climbing* local search technique is used to find the optimal configurations of the ridge functions, as detailed in Sect. 4.2.

Table 2: Gain of fingerprint tables over rainbow tables for various N and M.

	2^{38}	2^{40}	2^{42}	2^{44}	2^{46}	2^{48}
2GB	32.48%	35.94%	39.01%	41.73%	44.16%	46.35%
4GB	30.76%	34.36%	37.54%	40.36%	42.88%	45.14%
8GB	29.04%	32.76%	36.05%	38.97%	41.58%	43.93%

Table 2 is filled with memory sizes that belong to a reasonable range for an average personal computer. The problem sizes are also driven by practical considerations, to avoid a prohibitive online search time. Analyzing the results leads to two trends. First of all, the advantage of the fingerprint tables over the rainbow tables tends to increase as the memory decreases. This can be mainly explained by the fact that E_c increases linearly with M. Secondly, the gain increases with the problem size. This behavior can be explained by the fact that when N is large, there is more freedom in the configuration of the ridge functions.

4.2 Finding the Optimal Configurations

As we have seen in Sect. 3, the performance of the fingerprint tables strongly depends on the nature of the fingerprint.

One way to find the best configuration for the fingerprint is to apply a brute-force technique to compute T for the $(1 + \lceil \log_2 N \rceil)^t$ possibilities, and keep the one that minimizes T. This is obviously not feasible for any practical instance. Instead, we used a local search technique called Hill Climbing [22] to compute these configurations efficiently. This may lead to suboptimal solutions, but gives very good results (i.e. drastic decrease of the false alarms) nonetheless. In the following, the "optimal configurations" refer to the best solutions found using this approach.

4.3 Optimal Configurations

Tab. 3 lists differences in the parameters and results between fingerprint tables in optimal configuration and rainbow tables in several settings. The row "Positions" corresponds to the set of columns associated with a one-bit ridge. In the optimal configurations found, all ridges but the one in the last column consist of at most one bit. We assumed the use of the following ridge functions:

$$\begin{aligned} \Phi_i : A &\to \{0,1\}^{\sigma_i} \\ x &\mapsto \begin{cases} \mathrm{lsb}_{\sigma_i}(x) & \text{if } \sigma_i > 0 \\ \epsilon & \text{otherwise,} \end{cases} \end{aligned}$$

where $\mathrm{lsb}_n(x)$ is a function that outputs the n least significant bits of x.

One can observe that when additional memory is available, the optimal solutions tend to use some extra memory per chain, and vice versa. Additionally, we can note that the cost of a false alarm for fingerprint tables is around one third of the one for rainbow tables.

5 Experimental Results and Comparison

A time-memory trade-off with fingerprint tables has been implemented in order to illustrate the theory with experimental results. Our implementation has been used to crack Windows LM Hash passwords and Windows NTLM Hash passwords. We also practically compared the performance of the fingerprint tables with the rainbow tables. Note that, since rainbow tables are a special case of fingerprint tables, our softare may also be used to build them.

5.1 Windows LM Hash Passwords

The LM Hash authentication is limited by nature to passwords (uppercase only) of length 1 to 7 characters[4]. In order to compare the performance of the fingerprint tables with Ophcrack[5], we considered alphanumeric passwords only, which amounts to $N = \sum_{i=1}^{7} 36^i \approx 2^{36.23}$ passwords.

Also, Ophcrack uses an ad-hoc compression technique which consists in a decomposition of the endpoints in prefix and suffix tables (see [1] for more details). The size of the chains when using prefix-suffix decomposition depends on the parameters of the problem and is implementation-dependent; its determination is out of the scope of this article. Therefore, we chose the size of the fingerprints to be 37 bits, in order to match the size of the endpoints in the Ophcrack tables. In this way the prefix-suffix compression has exactly the same impact on the size of the tables and produces tables of 379MB in both cases, allowing for a fair comparison. The parameters

[4] LM Hash passwords longer than 7 characters are automatically splitted by the operating system into two passwords of at most 7 characters.

[5] Ophcrack is a Windows LM Hash password cracker considered to be the fastest one ever implemented.

Table 3: Analytical performance for the best configurations of fingerprint tables.

(a) $N = 2^{48}$, $M = 2$GB

	rainbow tables	fingerprint tables
m	4.47×10^7	4.88×10^7
$\lvert E_j\rvert, \lvert F_j\rvert$	48	40
t	1.26×10^7	1.15×10^7
T	2.32×10^{13}	1.24×10^{13} (-46.35%)
Average FA cost	1.33×10^{13}	0.41×10^{13} (-68.88%)
Positions	8476358, 9172110, 9663530, 10050060, 10371170, 10647046, 10889558, 11106326, 11302572, 11482032	

(b) $N = 2^{48}$, $M = 4$GB

	rainbow tables	fingerprint tables
m	8.95×10^7	9.65×10^7
$\lvert E_j\rvert, \lvert F_j\rvert$	48	41
t	6.29×10^6	5.83×10^6
T	5.79×10^{12}	3.18×10^{12} (-45.14%)
Average FA cost	3.33×10^{12},	1.06×10^{12} (-68.20%)
Positions	4285001, 4636802, 4885286, 5080733, 5243100, 5382597, 5505222, 5614831, 5714062, 5804807	

(c) $N = 2^{48}$, $M = 8$GB

	rainbow tables	fingerprint tables
m	1.79×10^8	1.89×10^8
$\lvert E_j\rvert, \lvert F_j\rvert$	48	43
t	3.15×10^6	2.98×10^6
T	1.45×10^{12}	0.81×10^{12} (-43.93%)
Average FA cost	8.31×10^{11}	2.58×10^{11} (-68.99%)
Positions	2155147, 2334199, 2460731, 2560281, 2642997, 2714070, 2776554, 2832410, 2882981, 2929230, 2971872	

of the tables are $m = 1.54 \times 10^7$, $t = 10000$, $\ell = 4$ and the optimal ridge functions[6] for these parameters are defined by:

$$\Phi_i(x) = \begin{cases} \mathrm{lsb}_1(x) & \text{if } i \in \{7540, 8168, 8612, 8961, 9251, 9501, 9720, 9917\} \\ \mathrm{lsb}_{29}(x) & \text{if } i = 10000 \\ \epsilon & \text{otherwise.} \end{cases}$$

Tab. 4 shows that the results achieved by our implementation of fingerprint tables are in accord with the theoretical expectations calculated from Theorem 3.

Tab. 4 also shows that fingerprint tables behave much better than the rainbow tables: their time-gain over the rainbow tables is about 30% in spite of a suboptimal configuration for the fingerprint tables. We can emphasize that the time saved by fingerprint tables compared to rainbow tables is due to the reduced cost due to false alarms. Indeed, as m and t stay the same, the average number of operations for building the online chains stays the same too.

Table 4: Theoretical and experimental results for LM Hash passwords.

	Fingerprint Tables		Rainbow Tables
	(theoretical)	(experimental)	(experimental)
# operations total ($\times 10^6$)	10.75	10.73	15.25
# operations for false alarms ($\times 10^6$)	3.995	4.09	8.62
# false alarms	508.3	533.7	1120

5.2 Windows NTLM Hash Passwords

We considered in this second experiment NTLM Hash alphanumeric (both lowercase and uppercase) passwords of length 1 to 7, which represents a search space of $N = \sum_{i=1}^{7} 62^i \approx 2^{41.70}$. Considering longer passwords would better emphasize the performance of the fingerprint tables, but we were time-constrained for the precomputation of the tables.

The parameters we used are $m = 10.5 \times 10^9$, $t = 13554$, $\ell = 4$. We also used prefix/suffix decomposition [1] in order to save some extra memory, but this has no influence on the online performance. The four tables take up about 14.8GB in total. Again, using the methodology described in section 4.2, we found the following optimal configuration:

$$\Phi_i(x) = \begin{cases} \mathrm{lsb}_1(x) & \text{if } i \in \{10077, 10928, 11530, 12004, 12398, 12736, 13034, 13301\} \\ \mathrm{lsb}_{34}(x) & \text{if } i = 13554 \\ \epsilon & \text{otherwise.} \end{cases}$$

The results of our experiment are presented in Table 5. Again, we observe that the practice matches with the theoretical estimations.

The precomputing phase for these tables took roughly a month on about a hundred machines. The online phase takes place on a machine with a i7-3770 CPU and 16GB of RAM. Recovering any alphanumeric NTLM Hash password (whose length is 1 to 7 characters) in this setting takes 3.5 seconds on average.

[6] This configuration was found using the approach described in section 4.2.

Table 5: Theoretical and experimental results for the fingerprint tables (NTLM Hash).

	Theoretical	Experimental
# operations total ($\times 10^6$)	19.44	19.29
# operations for false alarms ($\times 10^6$)	6.79	7.15
# false alarms	655.58	697.15

6 Conclusion

Cryptanalytic time-memory trade-offs constitute the keystone of many practical cryptanalyses, for example against A5/1 (GSM) in 2000, LILI-128 in 2002, Windows LM Hash in 2003, Unix passwords in 2005, and Texas Instruments DST in 2005. In spite of the wide use of time-memory trade-offs, few advances have been done since Oechslin introduced the rainbow tables.

The method we introduce in this article brings a major breakthrough in TMTO, discarding the endpoints from the tables. Instead, we propose a new time-memory trade-off construction that we call fingerprint tables, which provides a significant decrease in the time required for the search, compared to the rainbow tables. Specifically, the cost of false alarms is cut down to one third with respect to rainbow tables, which makes the search process up to twice as fast. We think that this new technique to construct tables should lead to further progress in the TMTO research field.

After introducing the fingerprint tables, we provided an analysis of the average search time of fingerprint tables and proposed a method to search for optimal parameters. We highlighted that the rainbow tables are a special case of fingerprint tables with non optimal parameters.

We also implemented our method and corroborated our theoretical analysis. We investigated the particular case of the Windows LM Hash password cracking, and compared our results with Ophcrack, a well-known implementation of rainbow tables for this problem. We finally implemented a Windows NTLM Hash password cracker based on fingerprint tables and made it freely available online.

Bibliography

[1] Gildas Avoine, Pascal Junod, and Philippe Oechslin. Time-memory trade-offs: False alarm detection using checkpoints. In *Progress in Cryptology – Indocrypt 2005*, volume 3797 of *Lecture Notes in Computer Science*, pages 183–196, Bangalore, India, December 2005. Cryptology Research Society of India, Springer-Verlag.

[2] Gildas Avoine, Pascal Junod, and Philippe Oechslin. Characterization and improvement of time-memory trade-off based on perfect tables. *ACM Trans. Inf. Syst. Secur.*, 11:17:1–17:22, July 2008. ISSN 1094-9224.

[3] Steve Babbage. A space/time tradeoff in exhaustive search attacks on stream ciphers. In *European Convention on Security and Detection*, volume 408, 1995.

[4] Elad Barkan, Eli Biham, and Adi Shamir. Rigorous bounds on cryptanalytic time/memory tradeoffs. In Cynthia Dwork, editor, *Advances in Cryptology – CRYPTO'06*, Lecture Notes in Computer Science, Santa Barbara, California, USA, August 2006. IACR, Springer-Verlag.

[5] Alex Biryukov and Adi Shamir. Cryptanalytic time/memory/data tradeoffs for stream ciphers. In Colin Boyd, editor, *Advances in Cryptology – ASIACRYPT'01*, volume 2248 of *Lecture Notes in Computer Science*, pages 1–13, Gold Coast, Australia, December 2001. IACR, Springer-Verlag.

[6] Alex Biryukov, Adi Shamir, and David Wagner. Real time cryptanalysis of A5/1 on a PC. In Bruce Schneier, editor, *Fast Software Encryption – FSE'00*, volume 1978 of *Lecture Notes in Computer Science*, pages 1–18, New York, USA, April 2000. Springer-Verlag.

[7] Alex Biryukov, Sourav Mukhopadhyay, and Palash Sarkar. Improved time-memory trade-offs with multiple data. In Bart Preneel and Stafford Tavares, editors, *Selected Areas in Cryptography – SAC 2005*, volume 3897 of *Lecture Notes in Computer Science*, pages 110–127, Kingston, Canada, August 2005. Springer-Verlag.

[8] Steve Bono, Matthew Green, Adam Stubblefield, Ari Juels, Avi Rubin, and Michael Szydlo. Security Analysis of a Cryptographically-Enabled RFID Device. In *14th USENIX Security Symposium – USENIX'05*, pages 1–16, Baltimore, Maryland, USA, July–August 2005. USENIX.

[9] Johan Borst, Bart Preneel, and Joos Vandewalle. On the time-memory tradeoff between exhaustive key search and table precomputation. In Peter de With and Mihaela van der Schaar-Mitrea, editors, *Symposium on Information Theory in the Benelux*, pages 111–118, Veldhoven, The Netherlands, May 1998.

[10] Anindya De, Luca Trevisan, and Madhur Tulsiani. Time space tradeoffs for attacks against one-way functions and PRGs. In Tal Rabin, editor, *Advances in Cryptology – CRYPTO'10*, volume 6223 of *Lecture Notes in Computer Science*, pages 649–665, Santa Barbara, California, USA, August 2010. IACR, Springer-Verlag.

[11] Dorothy Denning. *Cryptography and Data Security*, page 100. Addison-Wesley, Boston, Massachusetts, USA, 1982.

[12] Amos Fiat and Moni Naor. Rigorous time/space tradeoffs for inverting functions. In *ACM Symposium on Theory of Computing – STOC'91*, pages 534–541, New Orleans, Louisiana, USA, May 1991. ACM, ACM Press.

[13] Jovan Dj. Golić. Cryptanalysis of alleged a5 stream cipher. In *Advances in Cryptology – EUROCRYPT'97*, volume 1233 of *Lecture Notes in Computer Science*, pages 239–255, Konstanz, Germany, May 1997. IACR, Springer-Verlag.

[14] Martin Hellman. A cryptanalytic time-memory trade off. *IEEE Transactions on Information Theory*, IT-26(4):401–406, July 1980.

[15] Jin Hong, Kyung Jeong, Eun Kwon, In-Sok Lee, and Daegun Ma. Variants of the distinguished point method for cryptanalytic time memory trade-offs. In Liqun Chen, Yi Mu, and Willy Susilo, editors, *Information Security Practice and Experience*, volume 4991 of *Lecture Notes in Computer Science*, pages 131–145, Sydney, Australia, April 2008. Springer Berlin / Heidelberg.

[16] Iljun Kim and Tsutomu Matsumoto. Achieving higher success probability in time-memory trade-off cryptanalysis without increasing memory size. *IEICE Transactions on Communications/Electronics/Information and Systems*, E82-A(1):123–, January 1999.

[17] Koji Kusuda and Tsutomu Matsumoto. Optimization of time-memory trade-off cryptanalysis and its application to DES, FEAL-32, and Skipjack. *IEICE Transactions on Fundamentals*, E79-A(1):35–48, January 1996.

[18] Daegun Ma and Jin Hong. Success probability of the hellman trade-off. *Information Processing Letters*, 109(7):347–351, December 2008.

[19] Nele Mentens, Lejla Batina, Bart Preneel, and Ingrid Verbauwhede. Cracking Unix passwords using FPGA platforms. SHARCS - Special Purpose Hardware for Attacking Cryptographic Systems, February 2005.

[20] Karsten Nohl. Attacking phone privacy. Blackhat - White Paper, 2010.

[21] Philippe Oechslin. Making a faster cryptanalytic time-memory trade-off. In Dan Boneh, editor, *Advances in Cryptology – CRYPTO'03*, volume 2729 of *Lecture Notes in Computer Science*, pages 617–630, Santa Barbara, California, USA, August 2003. IACR, Springer-Verlag.

[22] Stuart J. Russell and Peter Norvig. *Artificial intelligence: a modern approach*, volume 2. Pearson Education, 2003.

[23] Markku-Juhani Olavi Saarinen. A time-memory tradeoff attack against LILI-128. In *Fast Software Encryption*, volume 2365, pages 231–236, Leuven, Belgium, February 2001.

[24] François-Xavier Standaert, Gael Rouvroy, Jean-Jacques Quisquater, and Jean-Didier Legat. A time-memory tradeoff using distinguished points: New analysis & FPGA results. In Burton Kaliski, Çetin Kaya Koç, and Christof Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems – CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 593–609, Redwood Shores, California, USA, August 2002. Springer-Verlag.