Time-Memory Trade-Offs: False Alarm Detection Using Checkpoints

Gildas Avoine[1], Pascal Junod[2], and Philippe Oechslin[1,3]

[1] EPFL, Lausanne, Switzerland

[2] Nagravision SA (Kudelski Group), Switzerland

[3] Objectif Sécurité, Gland, Switzerland

Abstract. Since the original publication of Martin Hellman's cryptanalytic time-memory trade-off, a few improvements on the method have been suggested. In all these variants, the cryptanalysis time decreases with the square of the available memory. However, a large amount of work is wasted during the cryptanalysis process due to so-called "false alarms". In this paper we present a method of detection of false alarms which significantly reduces the cryptanalysis time while using a minute amount of memory. Our method, based on "checkpoints", reduces the time by much more than the square of the additional memory used, e.g., an increase of 0.89% of memory yields a 10.99% increase in performance. Beyond this practical improvement, checkpoints constitute a novel approach which has not yet been exploited and may lead to other interesting results. In this paper, we also present theoretical analysis of time-memory trade-offs, and give a complete characterization of the variant based on rainbow tables.

Key words: time-memory trade-off, cryptanalysis, precomputation

1 Introduction

Many cryptanalytic problems can be solved in theory using an exhaustive search in the key space, but are still hard to solve in practice because each new instance of the problem requires to restart the process from scratch. The basic idea of a time-memory trade-off is to carry out an exhaustive search once for all such that following instances of the problem become easier to solve. Thus, if there are N possible solutions to a given problem, a time-memory trade-off can solve it with T units of time and M units of memory. In the methods we are looking at T is proportional to N^2/M^2 and a typical setting is $T = M = N^{2/3}$.

The cryptanalytic time-memory trade-off has been introduced in 1980 by Hellman [8] and applied to DES. Given a plaintext P and a ciphertext C, the problem consists in recovering the key K such that $C = \mathsf{S}_K(P)$ where S is an encryption function assumed to follow the behavior of a random function. Encrypting P under all possible keys and storing each corresponding ciphertext allows for immediate cryptanalysis but needs N elements of memory. The idea of a trade-off is to use chains of keys. It is achieved thanks to a reduction function R which generates a key from a ciphertext. Using S and R, chains of alternating ciphertexts and keys can thus be generated. The key point is that only the first and the last element of each chain are stored. In order to retrieve K, a chain is generated from C. If at some point it yields a stored end of chain, then the entire chain is regenerated from its starting point. However, finding a matching end of chain does not necessarily imply that the key will be found in the regenerated chain. There exist situations where the chain that has been generated from C merges with a chain that is stored in the memory which does not contains K. This situation is called a *false alarm*. Matsumoto, with Kusuda [10] in 1996 and with Kim [9] in 1999, gave a more precise analysis of the parameters of the trade-off. In 1991, Fiat and Naor [6, 7] showed that there exist cryptographically sound one-way functions that cannot be inverted with such a trade-off.

Since the original work of Hellman, several improvements have been proposed. In 1982, Rivest [5] suggested an optimization based on *distinguished points* (DP) which greatly reduces the amount of look-up operations which are needed to detect a matching end point in the table. Distinguished points are keys (or ciphertexts) that satisfy a given criterion, e.g., the last n bits are all zero. In this variant, chains are not generated with a given length but they stop at the first occurrence of a distinguished point. This greatly simplifies the cryptanalysis. Indeed, instead of looking up in the table each time a key is generated on the chain from C, keys are generated until a distinguished point is found and only then a look-up is carried out in the table. If the average length of the chains is t, this optimization reduces the amount of look-ups by a factor t. Because merging chains significantly degrades the efficiency of the trade-off, Borst, Preneel, and Vandewalle [4] suggested in 1998 to clean the tables by discarding the merging and cycling chains. This new kind of tables, called *perfect table*, substantially decreases the required memory. Later, Standaert, Rouvroy, Quisquater, and Legat [14] dealt with a more realistic analysis of distinguished points and also proposed an FPGA implementation applied to DES with 40-bit keys. Distinguished points can also be used to detect collisions when a function is iterated, as proposed by Quisquater and Delescaille [13], and van Oorschot and Wiener [15].

In 2003, Oechslin [12] introduced the trade-off based on *rainbow tables* and demonstrated the efficiency of his technique by recovering Windows passwords. A rainbow table uses a different reduction function for each column of the table. Thus two different chains can merge only if they have the same key at the same position of the chain. This makes it possible to generate much larger tables. Actually, a rainbow table acts almost as if each column of the table was a separate single classic[4] table. Indeed, collisions within a classic table (or a column of a rainbow table) lead to merges whereas collisions between different classic tables (or different columns of a rainbow table) do not lead to a merge. This analogy can be used to demonstrate that a rainbow table of mt chains of length t has

[4] By *classic* we mean the tables as described in the original Hellman paper.

the same success rate as t single classic tables of m chains of length t. As the trade-off based on distinguished point, rainbow tables reduce the amount of look-ups by a factor of t, compared to the classic trade-off. Up until now, trade-off techniques based on rainbow tables are the most efficient ones. Recently, an FPGA implementation of rainbow tables has been proposed by Mentens, Batina, Preneel, and Verbauwhede [11] in order to retrieve Unix passwords.

Whether it is the classic Hellman trade-off, the distinguished points or the rainbow tables, they all suffer from a significant quantity of false alarms. Contrarily to what is claimed in the original Hellman paper, false alarms may increase the time complexity of the cryptanalysis by more than 50%. We will explain this point below. In this paper, we propose a technique whose goal is to reduce the time spent to detect false alarms. It works with the classic trade-off, with distinguished points, and with rainbow tables. Such an improvement is especially pertinent in practical cryptanalysis, where time-memory trade-offs are generally used to avoid to repeat an exhaustive search many times. For example, when several passwords must be cracked [12], each of them should not take more than a few seconds. In [1], the rainbow tables are used to speed up the search process in a special database. In such a commercial application, time is money, and therefore any improvement of time-memory trade-off also.

In Section 2, we give a rough idea of our technique based on checkpoints. We provide in Section 3 a detailed and formal analysis of the rainbow tables. These new results allow to formally compute the probability of success, the computation time, and the optimal size of the tables. Based on this analysis we can describe and evaluate our checkpoint technique in detail. We illustrate our method by cracking Windows passwords based on DES. In Section 4, we show how a trade-off can be characterized in general. This leads to the comparison of the three existing variants of trade-off. Finally, we give in Section 5 several implementation tips which significantly improve the trade-off in practice.

2 Checkpoint Primer

2.1 False Alarms

When the precalculation phase is achieved, a table containing m starting points $S_1, \ldots, S_m$ and m end points $E_1, \ldots, E_m$ is stored in memory. This table can be regenerated by iterating the function f, defined by $f(K) := \mathsf{R}(\mathsf{S}_K(P))$, on the starting points. Given a row j, let $X_{j,i+1} := f(X_{j,i})$ be the i-th iteration of f on S_j and $E_j := X_{j,t}$. We have:

$$\begin{array}{ccccccccccc}
S_1 = & X_{1,1} & \xrightarrow{f} & X_{1,2} & \xrightarrow{f} & X_{1,3} & \xrightarrow{f} & \ldots & \xrightarrow{f} & X_{1,t} & = E_1 \\
S_2 = & X_{2,1} & \xrightarrow{f} & X_{2,2} & \xrightarrow{f} & X_{2,3} & \xrightarrow{f} & \ldots & \xrightarrow{f} & X_{2,t} & = E_2 \\
\vdots & & & & & & & & & & \vdots \\
S_m = & X_{m,1} & \xrightarrow{f} & X_{m,2} & \xrightarrow{f} & X_{m,3} & \xrightarrow{f} & \ldots & \xrightarrow{f} & X_{m,t} & = E_m
\end{array}$$

In order to increase the probability of success, i.e., the probability that K appears in the stored values, several tables with different reduction functions are generated.

Given a ciphertext $C = \mathsf{S}_K(P)$, the on-line phase of the cryptanalysis works as follows: R is applied on C in order to obtain a key Y_1, and then the function f is iterated on Y_1 until matching any E_j. Let s be the length of the generated chain from Y_1:

$$C \quad \stackrel{\mathsf{R}}{\rightarrow} \quad Y_1 \quad \stackrel{f}{\rightarrow} \quad Y_2 \quad \stackrel{f}{\rightarrow} \quad \dots \quad \stackrel{f}{\rightarrow} \quad Y_s$$

Then the chain ending with E_j is regenerated from S_j until yielding the expected key K. Unfortunately K is not in the explored chain in most of the cases. Such a case occurs when R collides: the chain generated from Y_1 merged with the chain regenerated from S_j after the column where Y_1 is. That is a false alarm, which requires $(t-s)$ encryptions to be detected.

Hellman [8] points out that the expected computation due to false alarms increases the expected computation by at most 50 percent. This reasoning relies on the fact that, for any i, $f^i(Y_1)$ is computed by iterating f i times. However $f^i(Y_1)$ should be computed from Y_i because $f^i(Y_1) = f(Y_i)$. In this case, the computation time required to reach a chain's end is significantly reduced on average while the computation time required to rule out false alarms stays the same. Therefore, false alarms can increase by more than 50 percent the expected computation. For example, formulas given in Section 3 allow to determine the computation wasted during the recovering of Windows passwords [12]: false alarms increase by 125% the expected computation.

2.2 Ruling Out False Alarms Using Checkpoints

Our idea consists in defining a set of positions α_i in the chains to be checkpoints. We calculate the value of a given function G for each checkpoint of each chain j and store these $G(X_{j,\alpha_i})$ with the end of each chain $X_{j,t}$. During the on-line phase, when we generate $Y_1, Y_2, \dots, Y_s$, we also calculate the values for G at each checkpoint, yielding the values $G(Y_{\alpha_i+s-t})$. If Y_s matches the end of a chain that we have stored, we compare the values of G for each checkpoint that the chain Y has gone through with the values stored in the table. If they differ at least for one checkpoint we know that this is a false alarm. If they are identical, we cannot determine if a false alarm will occur without regenerating the chain.

In order to be efficient, G should be easily computable and the storage of its output should require few bits. Below, we consider the function G such that $G(X)$ simply outputs the less significant bit of X. Thus we have:

$$\Pr\{G(X_{j,\alpha}) \neq G(Y_{\alpha+s-t}) \mid X_{j,\alpha} \neq Y_{\alpha+s-t}\} = \frac{1}{2}\left(1 - \frac{1}{2^{|K|}}\right) \approx \frac{1}{2}.$$

The case $X_{j,\alpha} \neq Y_{\alpha+s-t}$ occurs when the merge appears after the column α (Fig 1). The case $X_{j,\alpha} = Y_{\alpha+s-t}$ occurs when either K appears in the regenerated chain or the merge occurs before the column α (Fig. 2).

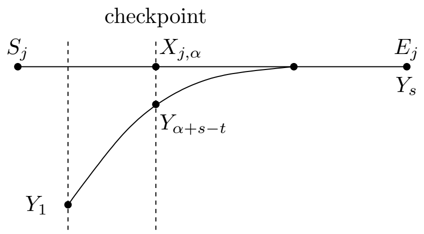

Fig. 1. False alarm detected with probability 1/2

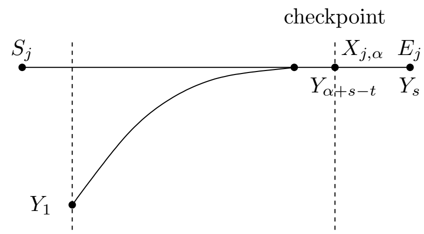

Fig. 2. False alarm not detected

In the next section we will analyze the performances of perfect rainbow tables in detail. Then, we will introduce the checkpoint concept in rainbow tables and analyze both theoretical and practical results.

3 Perfect Rainbow Tables and Checkpoints

3.1 Perfect Tables

The key to an efficient trade-off is to ensure that the available memory is used most efficiently. Thus we want to avoid the use of memory to store chains that contain elements which are already part of other chains. To do so, we first generate more chains than we actually need. Then we search for merges and remove chains until there are no merges. The resulting tables are called perfect tables. They have been introduced by [4] and analyzed by [14]. Creating perfect rainbow and DP tables is easy since merging chains can be recognized by their identical end points. Since end points need to be sorted to facilitate the look-ups, identifying the merges comes for free. Classic chains do not have this advantage. Every single element of every classic chain that is generated has to be looked up in all elements of all chains of the same table. This requires $mt\ell$ look-ups in total where ℓ is the number of stored tables. A more efficient method of generating perfect classic tables is described in [2]

Perfect classic and DP tables are made of unique elements. In perfect rainbow tables, no element appears twice in any given column, but it may appear more than once across different columns. This is consistent with the view that each column of a rainbow table acts like a single classic table. In all variants of the trade-off, there is a limit to the size of the perfect tables that can be generated. The brute-force way of finding the maximum number of chains of given length t that will not merge is to generate a chain from each of the N possible keys and remove the merges.

In the following sections, we will consider perfect tables only.

3.2 Optimal Configuration

From [12], we know that the success rate of a single un-perfect rainbow table is $1-\prod_{i=1}^{t}\left(1-\frac{m_i}{N}\right)$ where m_i is the number of different keys in column i. With perfect rainbow tables, we have $m_i = m$ for all i s.t. $1 \leq i \leq t$. The success rate of a single perfect rainbow table is therefore

$$P_{\text{rainbow}} = 1 - \left(1 - \frac{m}{N}\right)^t. \qquad (1)$$

The fastest cryptanalysis time is reached by using the largest possible perfect tables. This reduces the amount of duplicate information stored in the table and reduces the number of tables that have to be searched. For a given chain length t, the maximum number $m_{\max}(t)$ of rainbow chains that can be generated without merges is obtained (see [12]) by calculating the number of independent elements at column t if we start with N elements in the first column. Thus we have

$$m_{\max}(t) = m_t \;\text{ where }\; m_1 = N \;\text{ and }\; m_{n+1} = N\left(1 - e^{-\frac{m_n}{N}}\right) \text{ where } 0 < n < t.$$

For non small t we can find a closed form for $m_{\max}$ (see [2]):

$$m_{\max}(t) \approx \frac{2N}{t+2}.$$

From (1), we deduce the probability of success of a single perfect rainbow table having $m_{\max}$ chains:

$$P_{\text{rainbow}}^{\max} = 1 - \left(1 - \frac{m_{\max}}{N}\right)^t \approx 1 - e^{-t\frac{m_{\max}}{N}} \approx 1 - e^{-2} \approx 86\%.$$

Interestingly, for any N and for t not small, this probability tends toward a constant value. Thus the smallest number of tables needed for a trade-off only depends on the desired success rate P. This makes the selection of optimal parameters very easy (see [2] for more details):

$$\ell = \left\lceil \frac{-\ln(1-P)}{2} \right\rceil, \quad m = \frac{M}{\ell}, \text{ and } t = \frac{\ln(1-P)}{\ln(1 - \frac{M}{\ell N})\ell} \approx \frac{-N}{M}\ln(1-P).$$

3.3 Performance of the Trade-Off

Having defined the optimal configuration of the trade-off, we now calculate the exact amount of work required during the on-line phase. The simplicity of rainbow tables makes it possible to include the work due to false alarms both for the average and the worst case.

Cryptanalysis with a set of rainbow tables is done by searching for the key in the last column of each table and then searching sequentially through previous columns of all tables. There are thus a maximum of ℓt searches. We calculate the expectation of the cryptanalysis effort by calculating the probability of success and the amount of work for each search k. When searching a key at position c of a table, the amount of work to generate a chain that goes to the end of the table is $t - c$. The additional amount of work due to a possible false alarm is c

since the chain has to be regenerated from the start to c in order to rule out the false alarm. The probability of success in the search k is given below:

$$p_k = \frac{m}{N}\left(1 - \frac{m}{N}\right)^{k-1}. \tag{2}$$

We now compute the probability of a false alarm during the search k. When we generate a chain from a given ciphertext and look-up the end of the chain in the table, we can either not find a matching end, find the end of the correct chain or find an end that leads to a false alarm. Thus we can write that the probability of a false alarm is equal to one minus the probability of actually finding the key minus the probability of finding no end point. The probability of not finding an end point is the probability that all points that we generate are not part of the chains that lead into the end points. At column i, these are the m_i chains that we used to build the table. The probability of a false alarm at search k (i.e., in column $c = t - \lfloor \frac{k}{\ell} \rfloor$) is thus the following:

$$q_c = 1 - \frac{m}{N} - \prod_{i=c}^{i=t}\left(1 - \frac{m_i}{N}\right) \tag{3}$$

where $c = t - \left\lfloor \frac{k}{\ell} \right\rfloor$, $m_t = m$, and $m_{i-1} = -N \ln(1 - \frac{m_i}{N})$. When the tables have exactly the maximum number of chains $m_{\max}$ we find a short closed form for q_c (see [2] for more details):

$$q_c = 1 - \frac{m}{N} - \frac{c(c+1)}{(t+1)(t+2)}. \tag{4}$$

The average cryptanalysis time is thus:

$$T = \sum_{\substack{k=1 \\ c=t-\lfloor \frac{k}{\ell} \rfloor}}^{k=\ell t} p_k \left(W(t-c-1) + Q(c)\right)\ell + (1 - \frac{m}{N})^{\ell t}\left(W(t) + Q(1)\right)\ell \tag{5}$$

where $$W(x) = \sum_{i=1}^{i=x} i \quad \text{and} \quad Q(x) = \sum_{i=x}^{i=t} q_i i.$$

The second term of (5) is the work that is being carried out every time no key is found in the table while the first term corresponds to the work that is being carried out during the search k. W represents the work needed to generate a chain until matching a end point. Q represents the work to rule out a false alarm. We can rewrite (5) as follows:

$$\begin{aligned} T &= \sum_{\substack{k=1 \\ c=t-\lfloor \frac{k}{\ell} \rfloor}}^{k=\ell t} p_k \left(\sum_{i=1}^{i=t-c-1} i + \sum_{i=c}^{i=t} q_i i\right)\ell + (1 - \frac{m}{N})^{\ell t}\left(\sum_{i=1}^{i=t} i + \sum_{i=1}^{i=t} q_i i\right)\ell \\ &= \sum_{\substack{k=1 \\ c=t-\lfloor \frac{k}{\ell} \rfloor}}^{k=\ell t} p_k \left(\frac{(t-c)(t-c-1)}{2} + \sum_{i=c}^{i=t} q_i i\right)\ell + (1 - \frac{m}{N})^{\ell t}\left(\frac{t(t-1)}{2} + \sum_{i=1}^{i=t} q_i i\right)\ell \end{aligned}$$

We have run a few experiments to illustrate T. The results are given in Table 1.

Table 1. Calculated and measured performance of rainbow tables

$N = 8.06 \times 10^{10}$, $t = 10000$, $m = 15408697$, $\ell = 4$	theory	measured over 1000 experiments
encryptions (average)	1.55×10^7	1.66×10^7
encryptions (worst case)	2.97×10^7	2.96×10^8
number of false alarms (average)	1140	1233
number of false alarms (worst case)	26048	26026

3.4 Checkpoints in Rainbow Tables

From results of Section 3.3, we establish below the gain brought by the checkpoints. We firstly consider only one checkpoint α. Let $Y_1 \dots Y_s$ be a chain generated from a given ciphertext C. From (3), we know that the probability that $Y_1 \dots Y_s$ merges with a stored chain is q_{t-s}. The expected work due to a false alarm is therefore $q_{t-s}(t-s)$.

We now compute the probability that the checkpoint detects the false alarm. If the merge occurs before the checkpoint (Fig. 2) then the false alarm cannot be detected. If the chain is long enough, i.e., $\alpha + s > t$, the merge occurs after the checkpoint (Fig. 1) with probability q_α. In this case, the false alarm is detected with probability $\Pr\{G(X_{j,\alpha}) \neq G(Y_{\alpha+s-t}) \mid X_{j,\alpha} \neq Y_{\alpha+s-t}\}$.

We define $g_\alpha(s)$ as follows:

$$g_\alpha(s) = \begin{cases} 0 \text{ if there is no checkpoint in column } \alpha, \\ 0 \text{ if } (\alpha + s) \leq t, \text{ i.e. the generated chain does not reach column } \alpha, \\ \Pr\{G(X_{j,\alpha}) \neq G(Y_{\alpha+s-t}) \mid X_{j,\alpha} \neq Y_{\alpha+s-t}\} \text{ otherwise.} \end{cases}$$

We can now rewrite $Q(x) = \displaystyle\sum_{i=x}^{i=t} i \left(q_i - q_\alpha \cdot g_\alpha(t-i)\right)$.

We applied our checkpoint technique with $N = 8.06 \times 10^{10}$, $t = 10000$, $m = 15408697$, $\ell = 4$ and G as defined in Section 2.2. Both theoretical and experimental results are plotted on Fig. 3.

We can generalize to t checkpoints. We can rewrite $Q(x)$ as follows:

$$Q(x) = \sum_{i=x}^{i=t} i \left(q_i - q_i \cdot g_i(t-i) - \sum_{j=i+1}^{j=t} \left(q_j \cdot g_j(t-j) \prod_{k=i}^{k=j-1} (1 - g_k(t-k)) \right) \right).$$

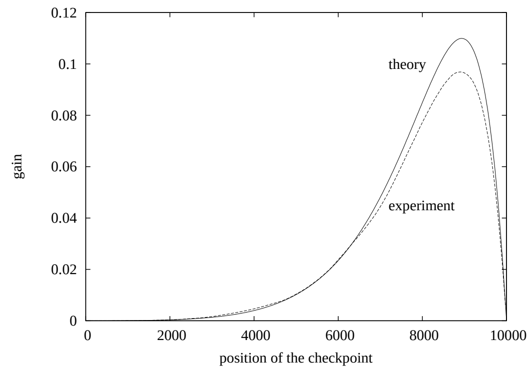

Fig. 3. Theoretical and experimental gain when one checkpoint is used

We now define memory cost and time gain. Let M, T, N and M', T', N' be the parameters of two trade-offs respectively. We define σ_M and σ_T as follows:

$$M' = \sigma_M \cdot M \quad \text{and} \quad T' = \sigma_T \cdot T.$$

The memory cost of the second trade-off over the first one is straightforwardly defined by $(\sigma_M - 1) = M'/M - 1$ and the time gain is $(1 - \sigma_T) = 1 - T'/T$. When a trade-off stores more chains, it implies a memory cost. Given that $T \propto N^2/M^2$ the time gain is:

$$(1 - \frac{T'}{T}) = 1 - \frac{1}{\sigma_M^2}.$$

Instead of storing additional chains, the memory cost can be used to store checkpoints. Thus, given a memory cost, we can compare the time gains when the additional memory is used to store chains and when it is used to store checkpoints. Numerical results are given in Table 2.

The numerical results are amazing. An additional 0.89% of memory saves about 10.99% of cryptanalysis time. This is six times more than the 1.76% of gain that would be obtained by using the same amount of memory to store additional chains. Our checkpoints thus perform much better than the basic trade-off. As we add more and more checkpoints, the gain per checkpoint decreases. In our example it is well worth to use 6 bits of checkpoint values (5.35% of additional memory) per chain to obtain a gain of 32.04%. The 0.89% of memory per checkpoint are calculated by assuming that the start and the end of the chains are stored in 56 bits each, as our example uses DES keys. As we explain in Section 5 the amount of bits used to store chain can be optimized and reduced to 49 bits in our example. In this case a bit of checkpoint data adds 2% of memory and it is still well worth using three checkpoints of one bit each to save 23% of work.

Table 2. Cost and gain of using checkpoint in password cracking, with $N = 8.06 \times 10^{10}$, $t = 10000$, $m = 15408697$, and $\ell = 4$

Number of checkpoints	1	2	3	4	5	6
Cost (memory)	0.89%	1.78%	2.67%	3.57%	4.46%	5.35%
Gain (time) storing chains	1.76%	3.47%	5.14%	6.77%	8.36%	9.91%
Gain (time) storing checkpoints	10.99%	18.03%	23.01%	26.76%	29.70%	32.04%
Optimal checkpoints	8935	8565 9220	8265 8915 9370	8015 8655 9115 9470	7800 8450 8900 9250 9550	7600 8200 8700 9000 9300 9600
	± 5	± 5	± 5	± 5	± 50	± 100

4 Characterization and Comparison of Trade-Offs

In this section we give a generic way of characterizing the different variants of the trade-off. We calculate the characteristic of rainbow tables exactly and compare it to measured characteristics of other variants.

4.1 Time-Memory Graphs

Knowing how to calculate the success rate and the number of operations needed to invert a function, we can now set out to plot the time-memory graphs. In order to do so, we fix a given success rate and for each memory size we find the table configuration that yields the fastest trade-off and plot the time that it takes. The graphs show that cryptanalysis time decreases with the square of the memory size, independently of the success rate. We can thus write the time-memory relation as

$$T = \frac{N^2}{M^2}\gamma(P) \tag{6}$$

where $\gamma(P)$ is a factor that depends only on the success probability. It is interesting to note that for $P = 86\%$ which is the the maximum success probability of a single rainbow table, the factor is equal to 1. In that case we find the typical trade-off which was already described by Hellman, namely that $M = T = N^{\frac{2}{3}}$.

Note that this simple expression of the trade-off performance was not possible for the previous variants. In those cases, calculations were always based on non-perfect tables, on the worst case (the key is not found in any table) and ignoring the amount of work due to false alarms. Optimizations have been proposed with these limitations, but to our knowledge the actual average amount of work, including false alarms has never been used to find optimal parameter. Our simple formula allows for a very simple calculation of the optimal parameters when any two of the success rate, the inversion time or the memory are given.

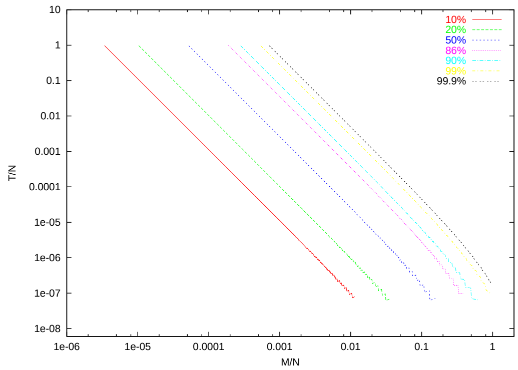

Fig. 4. Time-Memory graphs for rainbow tables, with various success rates.For $P_{\text{rainbow}} = 86\%$ the graph follows exactly $T = N^2/M^2$

4.2 The Time-Memory Characteristic

The previous section confirms that rainbow tables follow the same $T \propto N^2/M^2$ relation as other variants of the trade-off. Still, they seem to perform better. We thus need a criterion to compare the trade-offs. We propose to use $\gamma(P)$ as the trade-off characteristic. The evolution of γ over a range of P shows how a variant is better than another. Figure 5 shows a plot of $\gamma(P)$ for rainbow tables:

In the following sections, we compare the performance of rainbow tables with the performance of classic tables and DP tables. DP tables are much harder to analyze because of the variable length of the chains. We will thus concentrate on classic tables first.

4.3 Classic and DP Tables

The trade-off using classic or DP tables can also be characterized using the γ factor. Indeed both trade-offs follow the $T \propto N^2/M^2$ relation in a large part of the parameter space up to a factor which depends of the success rate and the type of trade-off. We have first devised a strategy to generate the largest possible perfect tables for each variant of the trade-off and have then used as many tables as necessary to reach a given success rate. The details of this work and the resulting time-memory graphs are available in [2]. In Figure 6 we show the evolution of the trade-off characteristic of classic tables and of DP tables.

The experiments and analysis show that rainbow tables outperform classic tables and DP tables for success rates above 80%. Below this limit, perfect classic tables are slightly better than perfect rainbow tables in terms of hash operations needed for cryptanalysis. However, the price of using classic tables is that they need t times more table look-ups. Since these do not come for free in most architectures (content addressable memory could be an exception), rainbow tables seem to be the best option in any case.

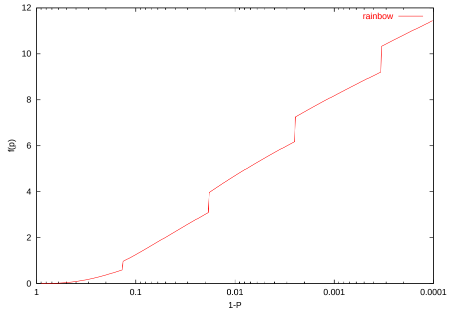

Fig. 5. The Time-Memory characteristic of rainbow tables.Steps happen every time an additional table has to be used to achieve the given success probability.

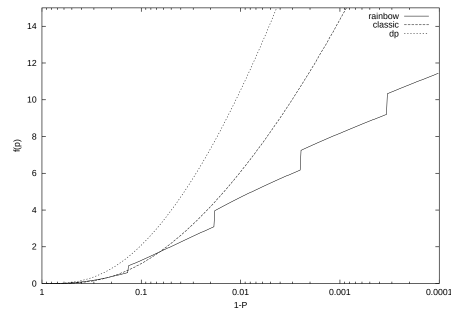

Fig. 6. The Time-Memory characteristics of rainbow, classic and DP tables compared.

5 Implementation Tips

For the sake of completeness we want to add some short remarks on the optimized implementation of the trade-offs. Indeed, an optimized implementation can yield performance gains almost as important as the algorithmic optimizations. We limit our tips to the implementation of rainbow tables.

5.1 Storing the Chain End Points

The number of operations of the trade-off decreases with the square of the available memory. Since available memory is measured in bytes and not in number of chains, it is important to choose an efficient format for storing the chains. A first issue is whether to use inputs or outputs of the function to be inverted (keys or ciphertexts) as beginning and end of chains. In practice the keys are usually smaller than the ciphertexts. It is thus more efficient to store keys (the key at the end of the chain has no real function but the extra reduction needed to generate it from the last ciphertext is well worth the saved memory). A second and more important issue is that we can take advantage of the way the tables are organized. Indeed a table consists of pairs of beginnings and ends of chains. To facilitate look-ups the chains are sorted by increasing values of the chain ends. Since the ends are sorted, successive ends often have an identical prefix. As suggested in [3] we can thus remove a certain length of prefix and replace it by an index table that indicates where every prefix starts in the table.

In our Windows password example, there are about 2^{37} keys of 56 bits. Instead of storing the 56 bits, we store a 37 bit index. From this index we take 21 bits as prefix and store only the last 16 bits in memory. We also store a table with 2^{21} entries that point to the corresponding suffixes for each possible prefix.

5.2 Storing the Chain Starting Points

The set of keys used for generating all the chains is usually smaller that the total set of keys. Since rainbow tables allow us to choose the starting points, we can use keys with increasing value of their index. In our example we used about 300 million starting points. This value can be expressed in 29 bits, so we only need to store the 29 lower bits of the index. The total amount of memory needed to store a chain is thus $29+16$ bits for the start and the end. The table that relates the prefixes to the suffixes incurs about 3.5 bits per chain. Altogether we thus need 49 bits per chain. A simple implementation that stores the full 56 bits of the start and end chain would need 2.25 times more memory and be 5 times slower.

5.3 Storing the Checkpoints

For reasons of efficiency of memory access it may in some implementations be more efficient to store the start and the end of a chain (that is, its suffix) in multiples of 8 bits. If the size of some parameters does not exactly match the size of the memory units, the spare bits can be used to store checkpoints for free. In our case, the 29 bits of the chain start are stored in a 32 bit word, leaving 3 bits available for checkpoints.

6 Conclusion

We have introduced a new optimization for cryptanalytic time-memory trade-offs which performs much better than the usual $T \propto N^2/M^2$. Our method works by reducing the work due to false alarms. Since this work is only a part of the total work our method can not reduce the work indefinitely. Besides having better performance, checkpoints can be generated almost for free while generating the trade-off tables. There is thus no indication for not using checkpoints and we conjecture that they will be used in many future implementations of the trade-off. Also, checkpoints are a new concept in time-memory trade-offs and they may lead to further optimizations and applications. In order to analyze the gain due to checkpoints we have presented a complete analysis of the rainbow tables. Using this analysis we are able to predict the gain that can be achieved with checkpoints. Finally we have also presented a simple way of comparing the existing variants of the trade-off with a so-called trade-off characteristic. We have calculated this characteristic for rainbow tables and measured it for the other variants. The results show that rainbow tables outperform the other variants in all cases except when table look-ups are free and the success probability is below 80%. The fact that the cryptanalysis time decreases with the square of the number of elements stored in memory indicates that it is very important to reduce the memory usage. This is why we have shared our tips on how this can be achieved in practice.

References

1. Gildas Avoine, Etienne Dysli, and Philippe Oechslin. Reducing time complexity in RFID systems. In *Selected Areas in Cryptography – SAC 2005*, Lecture Notes in Computer Science, Kingston, Canada, August 2005. Springer-Verlag.
2. Gildas Avoine, Pascal Junod, and Philippe Oechslin. Time-memory trade-offs: False alarms detection using checkpoints. Technical Report LASEC-REPORT-2005-002, Swiss Federal Institute of Technology (EPFL), Security and Cryptography Laboratory (LASEC), Lausanne, Switzerland, September 2005.
3. Alex Biryukov, Adi Shamir, and David Wagner. Real time cryptanalysis of A5/1 on a PC. In *Fast Software Encryption – FSE'00*, volume 1978 of *Lecture Notes in Computer Science*, pages 1–18, New York, USA, April 2000. Springer-Verlag.
4. Johan Borst, Bart Preneel, and Joos Vandewalle. On the time-memory tradeoff between exhaustive key search and table precomputation. In *Symposium on Information Theory in the Benelux*, pages 111–118, Veldhoven, The Netherlands, May 1998.
5. Dorothy Denning. *Cryptography and Data Security*, page 100. Addison-Wesley, Boston, Massachusetts, USA, June 1982.
6. 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.
7. Amos Fiat and Moni Naor. Rigorous time/space tradeoffs for inverting functions. *SIAM Journal on Computing*, 29(3):790–803, December 1999.
8. Martin Hellman. A cryptanalytic time-memory trade off. *IEEE Transactions on Information Theory*, IT-26(4):401–406, July 1980.
9. 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.
10. 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.
11. 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.
12. Philippe Oechslin. Making a faster cryptanalytic time-memory trade-off. In *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.
13. Jean-Jacques Quisquater and Jean-Paul Delescaille. How easy is collision search? Application to DES (extended summary). In *Advances in Cryptology – EUROCRYPT'89*, volume 434 of *Lecture Notes in Computer Science*, pages 429–434, Houthalen, Belgium, April 1989. IACR, Springer-Verlag.
14. François-Xavier Standaert, Gael Rouvroy, Jean-Jacques Quisquater, and Jean-Didier Legat. A time-memory tradeoff using distinguished points: New analysis & FPGA results. In *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.
15. Michael Wiener and Paul van Oorschot. Parallel collision search with cryptanalytic applications. *Journal of Cryptology*, 12(1):1–28, March 1999.