Heterogeneous Rainbow Table Widths Provide Faster Cryptanalyses

Gildas Avoine
INSA Rennes / IRISA
gildas.avoine@irisa.fr

Xavier Carpent
Computer Science Department
University of California, Irvine
xcarpent@uci.edu

ABSTRACT

Cryptanalytic time-memory trade-offs are techniques introduced by Hellman in 1980 to speed up exhaustive searches. Oechslin improved the original version with the introduction of rainbow tables in 2003. It is worth noting that this variant is nowadays used world-wide by security experts, notably to break passwords, and a key assumption is that rainbow tables are of equal width.

We demonstrate in this paper that rainbow tables are underexploited due to this assumption never being challenged. We stress that the optimal width of each rainbow table should be individually – although not independently – calculated. So it goes for the memory allocated to each table. We also stress that visiting sequentially the rainbow tables is no longer optimal when considering tables with heterogeneous widths.

We provide an algorithm to calculate the optimal configuration and a decision function to visit the tables. Our technique performs very well: it makes any TMTO based on rainbow tables 40% faster than its classical version.

Keywords

time-memory tradeoff; rainbow tables

1. INTRODUCTION

A cryptanalytic time-memory trade-off is a technique to find preimages of given outputs of a one-way function. They were first introduced by Hellman in 1980 [9] and they have been used in many practical attacks such as against A5/1 (used for GSM communications) in 2000 [7], or other stream ciphers like LILI-128 in 2002 [15]. The rainbow tables technique [14], a variant on Hellman's, has been illustrated by the very efficient cracking of Windows LM Hash passwords in 2003 [14] and Unix passwords (using FPGA) in 2005 [13].

Hellman's technique has been improved upon in various ways, mostly targeting the efficiency of the online phase. The most impactful of these improvements arguably was the aforementioned *rainbow tables* variant [14] introduced by Oechslin. Another notable variant is the *distinguished points* [8] by Rivest in 1982. A recent analysis [12] shows that the rainbow tables are the fastest variant known today. Several improvements on the rainbow tables were published during the last decade, including the checkpoints [4] and the fingerprints[1], and techniques to optimize the ending points storage [2] and to address non-uniform distributions [3].

Whatever the considered variant based on rainbow tables, time-memory trade-offs consist of tables sharing the same width. This assumption has never been challenged since the original publication of rainbow tables.

We demonstrate in this paper that rainbow tables are underexploited because considering tables of equal width is far from being the optimal configuration. We show that the width of each table – and so the memory allocated to each of these tables – should be individually (but not independently) calculated for each table. This approach lead to create so-called "heterogeneous tables", by opposition to "homogeneous tables". We also show that the widely-used rule that consists in visiting the tables sequentially is not the optimal one when considering heterogeneous tables. The paper thus shows that heterogeneous tables are about 40% faster than their homogeneous counterparts.

Section 2 provides the technical background that is needed to understand our technique. Section 3 describes the technique, which includes the description of the heterogenous tables and the interleaving exploring rule. Section 4 introduces an algorithm to identify the optinal configuration, and Section 5 finally evaluates the technique.

2. TECHNICAL BACKGROUND

2.1 Concept

A fundamental problem in cryptanalysis is finding the preimage of a given output of a one-way function. A simple method is applying the function to all possible inputs until finding the expected value. Such an exhaustive search requires N operations in the worst case to find a preimage, where N is the size of the input space. This becomes impractical when N is very large. The other extreme is to first construct a look-up table including all the preimage values. Afterwards, finding a preimage is done via a table look-up operation which requires a negligible amount of time. The precomputation process however requires an effort equal to an exhaustive search, but is to be performed only once. Although this method is quite fast during the online search phase, it may require prohibitively large amounts of memory for large problems.

ASIA CCS '17, April 02-06, 2017, Abu Dhabi, United Arab Emirates

DOI: http://dx.doi.org/10.1145/3052973.3053030

Time-memory trade-offs are an intermediate solution to this problem. They consist in an offline precomputation phase, and an online search phase, and require some memory. The efficiency of the online phase is proportional to N^2/M^2 where M is the memory associated to the trade-off. Typically, this translates into both time and memory being $O(N^{2/3})$, but ultimately the more memory is dedicated to the trade-off, the faster the search phase goes. The memory required is typically much smaller than for exhaustive storage, and the online phase is on average typically much faster than for exhaustive search. The precomputation phase however is more expensive than for the exhaustive storage solution.

We now introduce the notation used in this paper. Let $h : A \to B$ be a one-way function. Let $r_i : B \to A$ be the reduction function used in column i. The goal of a reduction function is to map a point in B to an arbitrary point in A in an efficient and uniformly distributed fashion. A typical reduction function is $r_i : y \mapsto (y + i) \bmod N$, with $N = |A|$. The rainbow tables method is divided into two phases: the offline (or precomputation) and the online phases.

2.2 Offline phase

During this step, the rainbow tables are computed and stored in memory. A table consists in a series of chains built by iterating alternatively h and r_i. The first points of the chains (called the *starting points*) are chosen arbitrarily (usually incremental values, see e.g. [2]). Chains are of fixed length t and once all chains are completed, only the starting points and the ending points (the last point of each chain) are saved. Tables are then usually filtered so as to only keep one chain per different ending point (*clean* tables[1]). The computation of chains stops when the number of chains with different ending points m is deemed satisfactory. See Figure 1 for a depiction of the structure of a rainbow table. Multiple clean rainbow tables are usually built for a given problem (see Section 3.1).

A table of *maximal size* is obtained when all (or almost all) the possible ending points are reached, which happens when the number of chains computed is sufficiently large (i.e. when any new chain would have a negligible probability of having a new ending point). Clean tables of maximal size are the most memory-efficient version of the rainbow tables [12]. See for instance [5, 12] for an analysis of clean tables and tables of maximal size (the results relevant for the analysis of heterogeneous tables are also presented in this paper).

2.3 Online phase

During this step, the rainbow tables are loaded in memory and searched through to find the preimage of a given hash. Given a hash $y = h(x)$, the goal is to find x. The first step consists in computing $r_{t-1}(y)$ and searching through the ending points list for a j such that $E_j = r_{t-1}(y)$. If such an ending point exists, a chain is rebuilt from the corresponding starting point S_j in order to compute $X_{j,t-1}$ and verify whether $h(X_{j,t-1}) = y$. If it is the case, the attack succeeds with $x = X_{j,t-1}$, and if not, this match was a false alarm. In the case of a false alarm, or when no matching endpoint is found, the attack proceeds to the next table. Once all tables are cycled, the attack proceeds to the next column, computing $r_{t-1}(h(r_{t-2}(y)))$, and again checking for a matching ending point. Then computing $r_{t-1}(h(r_{t-2}(h(r_{t-3}(y)))))$, and so on until either the search succeeds or all columns are have been searched through.

The search procedure for rainbow tables works in parallel. That is, all tables are searched through for each column rather than sequentially. The reason for this is that the search is increasingly more expensive towards the left of the tables. Result 1 (from [14]) quantifies that cost, in terms of number of cryptographic operations.

Result 1. *The average number of h evaluations during a search in column k (column 0 being the rightmost one) of a clean rainbow table of maximal size with chains of size t is:*

$$C_k = k + (t - k + 1)q_{t-k+1},$$

with

$$q_i = 1 - \frac{i(i-1)}{t(t+1)}.$$

It is relatively easy to observe numerically that the quantity written in Result 1 is increasing, but one can be convinced by observing that the negative term in k is multiplied by q_{t-k+1}, which is both smaller than 1 and decreasing.

3. OUR TECHNIQUE

3.1 Heterogeneous Tables

In order to obtain a clean table, many chains need to be thrown out, which reduces the coverage and thus the probability of success during the online phase. Even tables of maximal size have a bounded probability of success, provided in Result 2 and proved in [14].

Result 2. *The probability of success of a set of ℓ clean rainbow tables of maximal size is:*

$$P^* \approx 1 - e^{-2\ell}.$$

This implies that in order to obtain a higher probability of success while using maximal size clean tables, one must use ℓ independent tables, i.e., tables that use different reduction function families. A typical number for ℓ is 4, which achieves a total probability of success of about 99.97%. As explained in Section 2, these tables are built separately to one another, and explored in parallel during the online phase.

In the original paper [14] and subsequently, these ℓ tables are considered to have the same width and number of chains. What is suggested in this paper is to instead allow tables of different configurations, or *heterogeneous* tables, by opposition to *homogeneous* tables. Let[2] $[m]_k$ and $[t]_k$ be respectively the number of chains and the length of chains in table k. Since each table is considered to be clean and of maximal size, these quantities are related to one another through Result 3, proved in [14].

Result 3. *The average number m of chains of length t in a clean table of maximal size built for an input space of size N is:*

$$m = \frac{2N}{t+1}.$$

---

[1] Tables are said to be *perfect* [14, 5] or *clean* [2, 3] when they do not contain any merge.

[2] The brackets are used to specify quantities specific to tables, and also in order to avoid confusion with quantities such as m_i that appear for instance in [14].

$$\begin{array}{|c|cccccccc|c|}
\cline{1-1}\cline{10-10}
S_1 = X_{1,1} & \xrightarrow{r_1 \circ h} & X_{1,2} & \xrightarrow{r_2 \circ h} & \dots & \xrightarrow{r_{t-2} \circ h} & X_{1,t-1} & \xrightarrow{r_{t-1} \circ h} & & X_{1,t} = E_1 \\
S_2 = X_{2,1} & \xrightarrow{r_1 \circ h} & X_{2,2} & \xrightarrow{r_2 \circ h} & \dots & \xrightarrow{r_{t-2} \circ h} & X_{2,t-1} & \xrightarrow{r_{t-1} \circ h} & & X_{2,t} = E_2 \\
\vdots & & \vdots & & \ddots & & \vdots & & & \vdots \\
S_j = X_{j,1} & \xrightarrow{r_1 \circ h} & X_{j,2} & \xrightarrow{r_2 \circ h} & \dots & \xrightarrow{r_{t-2} \circ h} & X_{j,t-1} & \xrightarrow{r_{t-1} \circ h} & & X_{j,t} = E_j \\
\vdots & & \vdots & & \ddots & & \vdots & & & \vdots \\
S_m = X_{m,1} & \xrightarrow{r_1 \circ h} & X_{m,2} & \xrightarrow{r_2 \circ h} & \dots & \xrightarrow{r_{t-2} \circ h} & X_{m,t-1} & \xrightarrow{r_{t-1} \circ h} & & X_{m,t} = E_m \\
\cline{1-1}\cline{10-10}
\end{array}$$

Figure 1: Structure of a rainbow table. The framed columns, respectively the starting points and the ending points, are the parts stored in memory.

3.2 Interleaving

3.2.1 Rationale

When visiting homogeneous tables, the optimal order of search is to go through each table at the same pace. The reason for this is that searching through a table gets increasingly costly the further one goes (see Result 1).

For heterogeneous tables however, the optimal order of search might be different. Intuitively, it is better on average to start with the table that has the most/shortest chains. It is indeed in this table that the probability of success per step is highest, and the cost of verifying false alarms is lowest. However, after a while, the increasing cost in this table might overcome the benefit and it might be better to switch to the second table, possibly to return to the first table afterwards, or switch to the third and so on.

This idea of interleaving the order of search in multiple rainbow tables was explored for non-uniformly distributed input [3]. The premise of [3] is that rainbow tables are designed to work with uniformly distributed input, yet in some cases (notably for password cracking), the input is not uniformly distributed. The idea is to partition the input space into several subsets that each are considered to be individually uniformly distributed, and to build a set of ℓ rainbow tables for each of them. During the online phase, the ℓ tables of each subset are explored in parallel as one step of the search, but the order in which the subsets are explored is interleaved and tailored such that the average searching time is minimized.

The idea discussed in this paper works on the level of one rainbow table set, with input considered uniformly distributed. The technique is thus completely independent and compatible with the one discussed in [3] for non-uniformly distributed input.

3.2.2 Order of visit

This section discusses the order in which the columns in the ℓ tables should be visited. As mentionned in Section 3.2.1, this order of visit is not necessarily straightforward. The approach suggested here is the same as the one in [3], namely that at each step, a decision is made as to in which table the next search should be made, in order to minimize the average time.

This decision has a negligible impact on the overal computation. It is based on measuring a metric for each table, and picking the table in which this metric is highest. This metric for each table is defined as the ratio of the probability to find a solution divided by the average amount of work (Definition 1).

Definition 1. *The metric associated to the i-th step in the k-th table is defined as:*

$$\eta(i,k) = \frac{\Pr(x \textit{ found at the i-th step in table } k)}{\mathbb{E}[\textit{work for the i-th step in table } k]},$$

with x, an answer in the online phase.

In the case of clean rainbow tables, this metric is quantified in Theorem 1.

Theorem 1. *The metric associated to the i-th step in the k-th table is:*

$$\eta(i,k) = \frac{[m]_k}{N[C_i]_k}.$$

Proof. *The probability of finding x in any given column of the k-th table is* $\frac{[m]_k}{N}$*. In the denominator, the expected work required at step i in table k is denoted* $[C_i]_k$*, and is computed as indicated in Result 1.* □

When using this metric to determine the order of search, the average time is minimized. This has been proved in details in [3] for interleaving in the case of different subsets, but can be easily adapted to the case of interleaving different tables. Essentially, it boils down to picking, at each step, the table in which the probability to find an answer per cryptographic operation is highest. Intuitively, this strategy is optimal because the cost may only increase in further steps, while the probability stays the same.

4. OPTIMAL CONFIGURATION

For a given set of heterogeneous tables, the optimal order of search is determined in Section 3.2.2. The question remains how to determine how much of the total memory to allocate for each table. This essentially translates into how many chains each table contains. This also determines the length of chains because the tables are clean and of maximal size (see Result 3).

4.1 Average Searching Time

The approach used in this paper is to use an optimization algorithm to search for a configuration of table sizes that minimizes the average search time of the online phase. In that end, Result 4 (from [14]) presents the expression of the average time in the case of homogeneous tables and Theorem 2 presents a generalization of the average time in the case of heterogeneous tables.

RESULT 4. *The average number of h evaluations during the online phase of a set of ℓ clean rainbow tables of maximal size with m chains of size t on a problem set of size N is:*

$$\begin{aligned} T \;=\; & \sum_{k=1}^{t}\sum_{i=1}^{\ell}\frac{m}{N}\left(1-\frac{m}{N}\right)^{(k-1)\ell+i-1}\left(\ell\sum_{j=1}^{k-1}C_j+iC_k\right) \\ & + e^{-2\ell}\ell\sum_{k=1}^{t}C_k. \end{aligned} \tag{1}$$

THEOREM 2. *The average number of hash operations required in the online phase of a set of heterogeneous clean rainbow tables of maximal size, given an input subset of size N and ℓ tables with numbers of chains $\{[m]_1, ..., [m]_\ell\}$ and chain lengths $\{[t]_1, ..., [t]_\ell\}$, and with a vector $V = (V_1, ..., V_{\hat{t}})$ representing the order of visits (i.e. V_k is the table chosen at step k) is:*

$$\begin{aligned} T \;=\; & \sum_{k=1}^{\hat{t}}\frac{[m]_{V_k}}{N}\prod_{j=1}^{k-1}\left(1-\frac{[m]_{V_j}}{N}\right)\sum_{j=1}^{k}[C_{S_j}]_{V_j} \\ & + e^{-2\ell}\sum_{i=1}^{\ell}\sum_{s=1}^{[t]_i}[C_s]_i, \end{aligned} \tag{2}$$

with $\hat{t} = \sum_{b=1}^{n}[t]_b$, and S_k the number of steps for the table V_k after k steps in total, that is:

$$S_k = \#\{i \le k | V_i = V_k\}.$$

PROOF. *This expression is a generalization of Result 4 for heterogeneous tables and with a given order of visit. The result may be constructed using the same approach as in Result 4 (from [14]). Below is a developement showing that, in particular,* (2) *gives Result 4 when instantiated to homogeneous tables. That is, when:*

$$\begin{aligned} [t]_i &= t, & V_j &= (j-1) \bmod \ell + 1, \\ [m]_i &= m, & S_j &= \lfloor (j-1)/\ell \rfloor + 1, \\ [C_k]_i &= C_k, \end{aligned}$$

for all tables $1 \le i \le \ell$ and columns $1 \le j \le t$.

Replacing these values in Eq. (2) *gives:*

$$\begin{aligned} T \;=\; & \sum_{k=1}^{\hat{t}}\frac{[m]_{V_k}}{N}\prod_{j=1}^{k-1}\left(1-\frac{[m]_{V_j}}{N}\right)\sum_{j=1}^{k}[C_{S_j}]_{V_j} \\ & + e^{-2\ell}\sum_{i=1}^{\ell}\sum_{s=1}^{[t]_i}[C_s]_i \\ =\; & \sum_{k=1}^{\ell t}\frac{m}{N}\prod_{j=1}^{k-1}\left(1-\frac{m}{N}\right)\sum_{j=1}^{k}C_{S_j} + e^{-2\ell}\sum_{i=1}^{\ell}\sum_{s=1}^{t}C_s. \end{aligned}$$

The first summation can be split in two (one for tables and one for columns). Each index k becomes $(a-1)\ell + b$ in the next equation:

$$\begin{aligned} T \;=\; & \sum_{a=1}^{t}\sum_{b=1}^{\ell}\frac{m}{N}\prod_{j=1}^{(a-1)\ell+b-1}\left(1-\frac{m}{N}\right)\sum_{j=1}^{(a-1)\ell+b}C_{S_j} \\ & + e^{-2\ell}\sum_{i=1}^{\ell}\sum_{s=1}^{t}C_s. \end{aligned}$$

Constant sums and products can be simplified and the sum $\sum_{j=1}^{(a-1)\ell+b}C_{S_j}$ can be split in two, then again simplified, giving the expected Eq. (1)*:*

$$\begin{aligned} T \;=\; & \sum_{a=1}^{t}\sum_{b=1}^{\ell}\frac{m}{N}\left(1-\frac{m}{N}\right)^{(a-1)\ell+b-1}\left(\sum_{j=1}^{(a-1)\ell}C_{S_j}+\sum_{j=(a-1)\ell+1}^{(a-1)\ell+b}C_{S_j}\right) \\ & + e^{-2\ell}\ell\sum_{s=1}^{t}C_s \\ =\; & \sum_{a=1}^{t}\sum_{b=1}^{\ell}\frac{m}{N}\left(1-\frac{m}{N}\right)^{(a-1)\ell+b-1}\left(\ell\sum_{c=1}^{(a-1)}C_c+C_a b\right) \\ & + e^{-2\ell}\ell\sum_{s=1}^{t}C_s. \end{aligned}$$

□

4.2 Optimization Algorithm

The minimization problem can be expressed as follows:

$$\begin{aligned} \min_{[t]_1,\ldots,[t]_\ell} \quad & T([t]_1, \ldots, [t]_\ell) \\ \text{s.t.} \quad & \textstyle\sum_{i=1}^{\ell} M_i \le M, \end{aligned} \tag{3}$$

with M_i the memory associated with table i (of length $[t]_i$) and M the total memory. Note that, of Eq. (2), $([t]_1, \ldots, [t]_\ell)$ are the only variables of the problem, because $([m]_1, \ldots, [m]_\ell)$ are fixed by the respective chain lengths (see Result 3), and the order of visit V is decided as described in Section 3.2.2.

This type of constrained minimization problem can be solved efficiently by using methods such as Sequential Quadratic Programming [10, 11]. This technique is implemented in the `scipy python` library, that was used in the context of this work. The algorithm used is presented in Appendix A.

5. EVALUATION

5.1 Comparison

It is worth noting that both Result 4 and Theorem 2 only depend on t and ℓ (or $\{[t]_1, ..., [t]_\ell\}$). This is because the cost[3] from Result 1 only depends on t, and $\frac{m}{N} = \frac{2}{t+1}$ in clean tables of maximal size. Therefore, only comparing values of the average time for varying values of t and ℓ is relevant.

To evaluate our technique, we chose $N = 2^{40}$ because this is a typical input space size that is considered to evaluate TMTOs [14]. Figure 2 depicts the speedup of the average searching time of heterogeneous tables (as computed by Theorem 2) compared to homogeneous tables (as computed by Result 4) for various values of ℓ and t. Choosing t for a set of homogeneous tables defines a total memory, and the same memory is used for the corresponding set of heterogeneous tables. This in turns defines the minimization problem (3). We observe in Figure 2 that the speedup is independent of t, and increases for larger values of ℓ. The consequence is that the heterogeneous tables improvement is independent of the

---

[3] Note that this is not strictly true because the expression of the cost (and specificly q_i) is an approximation (see for instance [5] for details), which is not valid for extremely small values of t or N. For problems of reasonable size however, such as in typical instances of TMTO's, the approximation is very good and the expected average search time from Result 4 very close to the observed average cost.

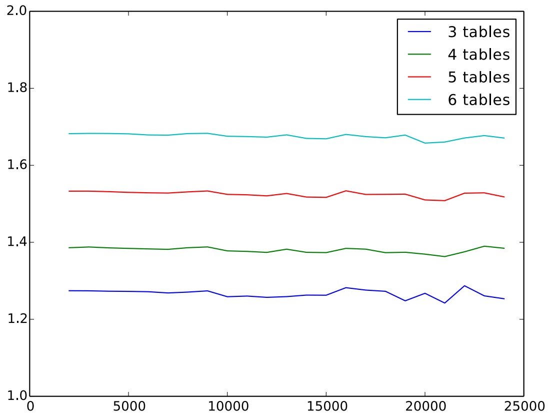


Figure 2: Speedup of the average searching time of heterogeneous tables compared to that of homogeneous tables as a function of t, for varying values of ℓ, and $N = 2^{40}$.

size of the problem. It does depend on ℓ (which sets the probability of success), but the number of tables will rarely be very high in typical applications.

Figure 3 shows a visual comparison of homogeneous and heterogeneous tables, along with a typical distribution of chain lengths in a set of heterogeneous tables. The space is of size $N = 2^{40}$ and $t = 10000$ is used for the homogeneous tables (which, together with the fact that tables of maximal size are used, fixes the number of chains per table to be $m = \frac{2N}{t+1} \approx 2.19 \times 10^8$). The values for $[t]_i$ for the heterogeneous tables are found through the optimization problem of Section 4, and are the ones computed by the script given in Appendix A. Note that each of the 8 tables in Figure 3 have the same area because they are all of maximal size, they just happen to have different shape. Like all setups using 4 tables, in this example heterogeneous tables perform about 40% faster than the homogeneous tables.

5.2 Precomputation

The cost of precomputation is roughly equivalent for heterogeneous tables and homogeneous tables. The reason is that the precomputation cost for each table is proportional to mt (see e.g. [12]). In clean tables of maximal size, this value is $2N$ and thus independent of the shape of the table.

Note that tables of "almost maximal size" are usually preferred in practice because they significantly reduce the precomputation load. The number of chains in a table of almost maximal size is typically 98% of what is found in a maximal table. It is worth noting that the impact of using tables of almost maximal size with our technique is negligible.

5.3 Worst Case

The procedure described in Section 4 aims to minimize the average searching time. However, a drawback of heterogeneous tables is that the worst case is worse than for homogeneous tables. The worst case arises when the value searched for is not covered by any of the ℓ tables. This occurs with probability $1 - e^{-2\ell}$ (Result 2), which is very rare for reasonnable values of ℓ. Nevertheless, it is a factor to consider in applications where the worst case is a concern.

To put this in perspective, Figure 4 shows the (cumulative) distribution probability of the average searching time for heterogeneous tables and homogeneous tables, with $N = 2^{40}$, $t = 10000$ and $\ell = 4$. Only on about 1% of cases are the heterogeneous tables worse than homogeneous ones.

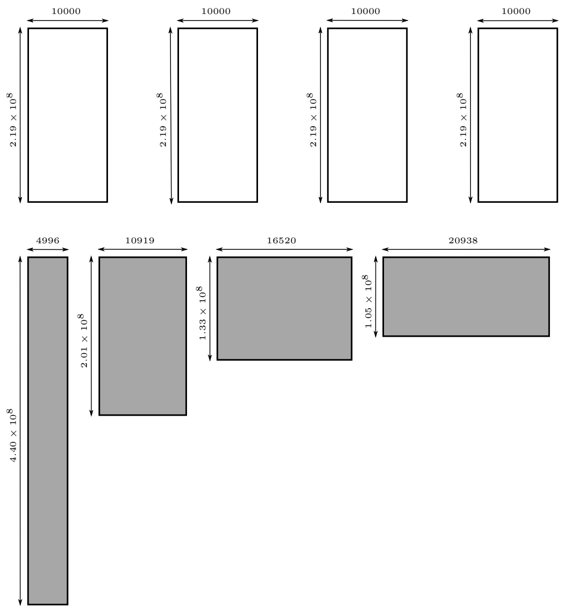


Figure 3: Shapes of a set of 4 homogeneous tables (top row) compared to optimal heterogeneous tables (bottom row) using the same memory on a set of size $N = 2^{40}$. The vertical scale (number of chains) is 10000 smaller than the horizontal scale (chain lengths).

6. OTHER TMTO VARIANTS

Table heterogeneity might be beneficial to other tradeoff variants than rainbow tables. This was not thoroughly explored as part of this work, mostly because rainbow tables seem to perform better [12][4]. A further argument is that it may have a less significant impact on these other variants.

In Hellman's technique, the cost of a search at each column (i.e. the analogue of Result 1) is roughly constant, so longer searches are less penalized than in rainbow tables. A more practical concern is the relative difficulty to find an optimal distribution of table sizes for a large number of tables. Nevertheless, it seems at least intuitively that Hellman's technique could perhaps benefit from using heterogeneous tables as well.

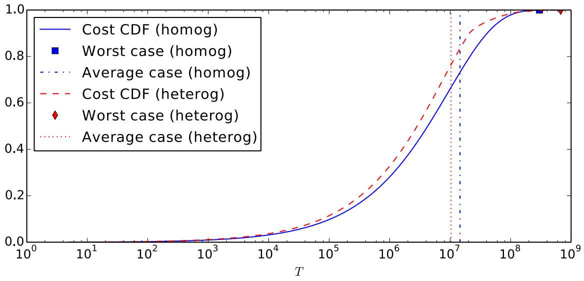
In the distinguished points variant however, tables are intrinsically heterogeneous themselves in the sense that chains *within* each table have different sizes, so the technique studied in this paper is not applicable.

7. CONCLUSION

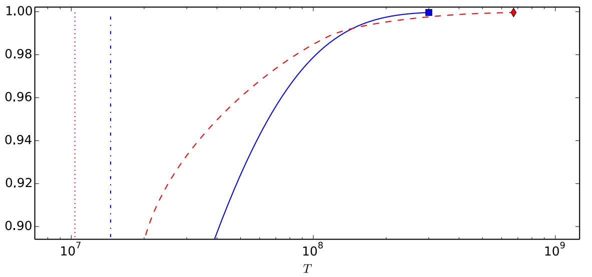
This paper explores the effects of allowing heterogeneous sizes in a set of rainbow tables, and gives the optimal exploration order. This results in a speedup in average time that is independent of problem size or memory, but depends on the number of tables (driven by the desired probabil-

---

[4] Note that some older analyses have different conclusions, e.g. [6]



(a) Cumulative distribution of probability of the search time.



(b) Detail of the above plot highlighting the area where heterogeneous tables are slower than homogeneous tables.

Figure 4: Comparison of the distribution of the searching time for homogeneous and heterogeneous tables. Parameters are $N = 2^{40}$, $\ell = 4$ and $t = 10000$.

ity of success). In typical applications (e.g. $\ell = 4$, that is $P^* = 99.97\%$), the heterogeneous tables are about 40% faster than their homogeneous counterparts. The worst-case time is negatively impacted: ≈ 2.13 slowdown with the same parameters, where less than 1% of cases are the heterogeneous tables worse than homogeneous ones. This downside is of limited consequences as typical applications of time-memory trade-offs favor average time.

The improvement stems from the relaxation of the arbitrary (albeit natural) constraint of having tables of equal dimensions, which allows to focus on efficiency at the start of the search. The precomputation cost and the probability of success are not impacted. An algorithm to determine the optimal memory distribution for a given number of tables is given, along with a procedure to determine the order of search.

Acknowledgments

Xavier Carpent was supported, in part, by a fellowship of the Belgian American Educational Foundation.

8. REFERENCES

[1] Gildas Avoine, Adrien Bourgeois, and Xavier Carpent. Analysis of rainbow tables with fingerprints. In *Australasian Conference on Information Security and Privacy – ACISP 2015*, Lecture Notes in Computer Science, Brisbane, Australia, June 2015. Springer.

[2] Gildas Avoine and Xavier Carpent. Optimal storage for rainbow tables. In Sukwoo Kim and Seok-Yeol Kang, editors, *International Conference on Information Security and Cryptology – ICISC 2013*, volume 8565 of *Lecture Notes in Computer Science*, pages 144–157, Seoul, South Korea, November 2013. Springer.

[3] Gildas Avoine, Xavier Carpent, and Cédric Lauradoux. Interleaving cryptanalytic time-memory trade-offs on non-uniform distributions. In Günther Pernul, Peter Y. A. Ryan, and Edgar Weippl, editors, *European Symposium on Research in Computer Security – ESORICS 2015*, volume 9326 of *Lecture Notes in Computer Science*, pages 165–184, Vienna, Austria, September 2015. Springer.

[4] 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.

[5] 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.

[6] Elad Barkan, Eli Biham, and Adi Shamir. Rigorous bounds on cryptanalytic time/memory tradeoffs. In *Annual International Cryptology Conference*, pages 1–21. Springer, 2006.

[7] 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.

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

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

[10] Dieter Kraft. A software package for sequential quadratic programming. Technical Report DFVLR-FB 88-28, DLR German Aerospace Center – Institute for Flight Mechanics, Koln, Germany, 1988.

[11] Dieter Kraft. Algorithm 733: TOMP-Fortran modules for optimal control calculations. *ACM Transactions on Mathematical Software*, 20(3):262–281, 1994.

[12] Ga Won Lee and Jin Hong. A comparison of perfect table cryptanalytic tradeoff algorithms. Cryptology ePrint Archive, Report 2012/540, 2012.

[13] 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.

[14] 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.

[15] 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.

APPENDIX

A. OPTIMIZATION ALGORITHM

```
import pylab
from scipy import optimize

################################################################################

def T(t, ell):
  T = 0
  Csum = 0
  pf = 2.0/(t+1) # = m/N
  for k in xrange(1, t+1):
    q = 1 - (t-k-1)*(t-k)/float((t)*(t+1))
    C = k + (t-k+1)*q
    for j in xrange(ell):
      Csum += C
      p = pf * (1-pf)**(ell*(k-1)+j)
      T += p*Csum
  return T

def T_heterog_intlv(ts):
  ell = len(ts)
  T = 0
  pnot = 1
  Csum = 0
  ks = [1]*ell
  stops = [False]*ell
  while not all(stops):
    metrics = [0]*ell
    ps = [0]*ell
    Cs = [0]*ell
    for j in xrange(ell):
      if stops[j]:
        continue
      q = 1 - (ts[j]-ks[j]-1)*(ts[j]-ks[j])/float((ts[j])*(ts[j]+1))
      Cs[j] = ks[j] + (ts[j]-ks[j]+1)*q
      ps[j] = 2.0/(ts[j]+1) # = [m]j/N
      metrics[j] = ps[j]/Cs[j]
    best = metrics.index(max(metrics))
    Csum += Cs[best]
    T += Csum * ps[best]*pnot
    pnot *= 1-ps[best]
    ks[best] += 1
    if ks[best] > ts[best]:
      stops[best] = True
  return T

def W_heterog(ts):
  W = 0
  for j in xrange(ell):
    for k in xrange(1, ts[j]+1):
      q = 1 - (ts[j]-k-1)*(ts[j]-k)/float((ts[j])*(ts[j]+1))
      C = k + (ts[j]-k+1)*q
      W += C
  return W

################################################################################

N = 2**40
t = 10000
m = 2*N/float(t+1)
ell = 4

Thomog = T(t, ell)

# The optimal order of search for homogeneous tables is parallel
assert abs(Thomog - T_heterog_intlv([t]*ell))/Thomog < 0.0001

# Initial guess for faster convergence (doesn't have to meet the constraints)
init_ts = pylab.linspace(t/2, 2*t, ell)

# Minimization
```

```
cons = ({'type':'ineq', 'fun':lambda ts:((m*ell)-sum([2*N/float(tt+1) for tt in ts]))},)
bnds = [(1, 10*t)]*ell
res = optimize.minimize(T_heterog_intlv, init_ts, method='SLSQP', bounds=bnds,
  constraints=cons, options={'maxiter':100, 'ftol':Thomog/1000.})
ts = [int(tt) for tt in res.x]
ms = [int(2*N/float(tt+1)) for tt in ts]
Theterog = T_heterog_intlv(ts)

# The total memory used in both cases is virtually equal
assert abs(m*ell - sum(ms))/(m*ell) < 0.0001

# Results
print 'homogeneous:'
print 't:', t
print 'M:', m*ell
print 'T:', Thomog
print 'W:', W_heterog([t]*ell)
print
print 'heterogeneous:'
print 'ts:', ts
print 'M:', sum(ms)
print 'T:', Theterog, '(␣speedup:', Thomog/Theterog, ')'
print 'W:', W_heterog(ts), '(␣slowdown:', W_heterog(ts)/W_heterog([t]*ell), ')'
```