Stairway To Rainbow

Gildas Avoine, Xavier Carpent, Diane Leblanc-Albarel

Stairway To Rainbow

Gildas Avoine
INSA, CNRS, IRISA
France

Xavier Carpent
University of Nottingham
United Kingdom

Diane Leblanc-Albarel
CNRS, INSA, IRISA
France

ABSTRACT

A cryptanalytic time-memory trade-off is a technique introduced by M. Hellman in 1980 to perform brute-force attacks. It consists of a time-consuming precomputation phase performed and stored once and for all, which is then used to reduce the computation time of brute-force attacks. A variant, known as *rainbow tables*, introduced by Oechslin in 2003 is used by most of today's off-the-shelf password-guessing tools. Precomputation of such tables is highly inefficient however, because much of the values computed during this task are eventually discarded. This paper revisits rainbow tables precomputation, challenging what has so far been regarded as an immutable foundation. The key idea consists in recycling values discarded during the precomputation phase, and adapting the brute force phase to make use of these recycled values. For a given memory and probability of success, the *stepped rainbow tables* thus created significantly reduce the workload induced by both the precomputation phase and the attack phase. The speedup obtained by using such tables is provided, and backed up by practical experiments.

CCS CONCEPTS

• **Security and privacy → Cryptanalysis and other attacks**; *Authentication.*

KEYWORDS

Applied Cryptography, Time-Memory Trade-Offs, Password Cracking, Rainbow Tables

ACM Reference Format:
Gildas Avoine, Xavier Carpent, and Diane Leblanc-Albarel. 2023. Stairway To Rainbow. In *ACM ASIA Conference on Computer and Communications Security (ASIA CCS '23), July 10–14, 2023, Melbourne, VIC, Australia.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3579856.3582825

1 INTRODUCTION

A cryptanalytic Time-Memory Trade-Off (TMTO) is a technique used to find preimages of outputs of a one-way function. TMTOs were initially developed by Hellman in 1980 [21] to attack block ciphers. TMTO has been at the heart of many real life attacks. For example, A5/1, a stream cipher used in GSM communications, was the target of a TMTO based attack [16, 19, 29]. More recently, in 2022, WPA3, a widely used WiFi protocol, has been attacked using TMTO [33]. TMTOs have notoriously been used to demonstrate the weakness of Windows passwords [31] and played a significant role in the popularization of salting in password hashing.

The *preimage problem* is defined as "finding $x^* \in A$ such that $h(x^*) = y$, given a one way (e.g., hash) function $h : A \to B$, and an image $y \in h(A) \subseteq B$". It has two extreme solutions: (1) the *brute force attack*, where all preimages in A are tested sequentially; and (2) the *dictionary* or *precomputed attack*, where a database of preimage-image pairs is computed and saved, and answering an instance of the preimage problem consists in a simple lookup. Denoting $N = |A|$, the former costs $O(N)$ hash operations during the attack phase but requires no setup or storage. The latter costs $O(N)$ hash operations during the precomputation phase as well as $O(N)$ in storage, but costs nothing in the attack phase (barring the cost of the lookup). A TMTO presents a compromise between the two: with $O(M)$ memory, the attack phase costs $O(N^2/M^2)$ and the precomputation phase is $O(N)$ [15].

Variants on Hellman's original construction have been developed over time. Some were designed to be applied to stream ciphers [14, 18, 20], or generalized to work with multiple data (so-called Time-Memory-Data Trade-Off [17]). Others apply TMTOs in specific circumstances or environments, such as when the input space is not uniformly distributed [9, 22], using external memory [8, 26], or FPGA [30, 32] and GPU [27, 29]. Finally, a number of techniques were designed to improve the trade-off characteristics, most notably *distinguished points* [23], *rainbow tables* [31], and *fuzzy rainbow tables* [25]. All of these have been analyzed [13, 32], improved upon [5–7, 12], and compared extensively [24, 28]. The rainbow tables technique in particular stands out in terms of efficiency as evidenced in [28], and is also popular in practice partly due to the visibility of the Windows password attack [31], and a number of hacking tools that implement rainbow tables [1–4]. We consequently focus on rainbow tables in this work.

A TMTO is usually thought of as a trade-off between attack time and attack memory (as its name suggests). However, other characteristics are important, namely: probability of success and precomputation time. The precomputation time in particular should not be underestimated, as for large N, it is the most likely practical bottleneck [10].

In this paper, we identify that these four characteristics are dictated by only three parameters: *length of chains* t, *number of tables* ℓ, and *coefficient of maximality* α (see Sect. 2). We then propose a new construction of rainbow tables that takes this framework into consideration. Using the *same memory* and the *same probability of success*, this construction, named *stepped rainbow tables*, can reach new points in the trade-off space that were not reachable with classic rainbow tables, and performs better everywhere else. Using stepped rainbow tables for instance allows (all other characteristics being equal) for 1.2 times smaller attack time, or 2.56 times smaller precomputation time (See Sect. 6.4).

ASIA CCS '23, July 10–14, 2023, Melbourne, VIC, Australia

ACM ISBN 979-8-4007-0098-9/23/07…$15.00
https://doi.org/10.1145/3579856.3582825

$$\begin{array}{ccccccc} & f_0 & & f_1 & & & \\ \boldsymbol{x_{0,1}} & \longrightarrow & x_{1,1} & \longrightarrow & x_{2,1} & \cdots & \boldsymbol{x_{t,1}} \\ & f_0 & & f_1 & & & \\ \boldsymbol{x_{0,2}} & \longrightarrow & x_{1,2} & \longrightarrow & x_{2,2} & \cdots & \boldsymbol{x_{t,2}} \\ \vdots & & \vdots & & \vdots & & \vdots \\ & f_0 & & f_1 & & & \\ \boldsymbol{x_{0,m}} & \longrightarrow & x_{1,m} & \longrightarrow & x_{2,m} & \cdots & \boldsymbol{x_{t,m}} \end{array}$$

Figure 1: Rainbow Matrix.

The paper is organized as follows. Some background on the construction and properties of rainbow tables is given in Sect. 2. The concept and construction of stepped rainbow tables is given in Sect. 3. A theoretical analysis of their properties is given in Sect. 4. A comparison between theoretical analysis and experiments is given in Sect. 5. A comparison between rainbow tables and stepped rainbow tables is given in Sect. 6. Finally, Sect. 7 concludes on the benefits of using stepped rainbow tables.

2 BACKGROUND ON RAINBOW TABLES

2.1 Matrix Generation

The first and main step of the precomputation is the construction of a so-called *rainbow matrix* (see Fig. 1), which consists of $t+1$ columns and m rows, where each element is denoted $x_{i,j}$, such that $x_{i,j} \in A$ with $0 \leqslant i \leqslant t$ and $0 < j \leqslant m$. Given $x_{i,j}$, $x_{i+1,j}$ is defined as:

$$x_{i+1,j} = f_i(x_{i,j}).$$

The functions f_i are called the *hash-reduction functions* and they are different in each column of the matrix with:

$$\begin{array}{rccc} f_i & : & A & \to & A \\ & & x_{i,j} & \mapsto & R_i(h(x_{i,j})) = x_{i+1,j}, \end{array}$$

where $R_i : B \to A$ are called *reduction functions* and h is the one-way function intrinsic to the preimage problem at hand. The choice of reduction functions is discussed in [31]. In a nutshell, they are chosen to (1) (nearly) uniformly map elements from B to elements in A,; (2) all be different; and (3) take negligible time compared to hash functions. A common construction is $R_i : y \mapsto y + i \bmod N$.

A sequence $(x_{i,j})_{i=0}^{t}$ is called a *chain*. The *length* of a chain is the number of hash-reduction functions applied to build a chain. A matrix of $t+1$ columns has chains of length t.

Fig. 1 depicts a rainbow matrix. $x_{0,j}$ is called the *start point (SP) of row j* and it can be arbitrarily chosen. $x_{t,j}$ is called the *end point (EP) of row j*.

After the matrix is computed, only the start points and end points, i.e., the first and the last columns (in bold in Fig. 1) are stored, while intermediary values are discarded. This is the key point of the time-memory trade-off technique. The set of pairs resulting from the first and last columns is called a *rainbow table*.

2.2 Clean Rainbow Tables

During the matrix generation, collisions may (and do) occur. When a collision between two values occurs in a given column, the corresponding chains *merge*, meaning that all their elements are equal after that column. More formally, a merge between two chains j and k occurs when $x_{i,j} \neq x_{i,k}$ and $x_{i+1,j} = x_{i+1,k}$ for a given column i. Merges are a source of inefficiency for rainbow tables. It is thus recommended (see e.g., [24, 28, 31]) to keep a single chain in each set of merged chains. A table without merged chains is called a *clean* rainbow table[1]. The rest of the paper discusses clean tables only.

2.3 Maximality of Rainbow Tables

Since cleaning a table reduces the number of chains it contains, there is an upper-bound on the number of chains a clean table of a given length can have. A clean table is said to be *maximal* when it has been generated from $m_0 = N$ start points. Given a maximal table containing $t+1$ columns, the number of unique end points in column t is given by Thm. 1, introduced and proved in [13]. This approximation is very precise numerically, and its simple closed form allows for expressing other results in a clear and insightful way.

Theorem 1. *Given t and a sufficiently large N, the expected maximum number of chains per clean rainbow table is:*

$$m_t^{\max} \approx \frac{2N}{t+2}.$$

Computing maximal tables is not practical however, because N is typically very large. In practice, *non-maximal* rainbow tables are generated, with $m_0 \ll N$. The *maximality factor* α (with $0 < \alpha < 1$), introduced in [10] is defined in Eq. (1)

$$m_t = \alpha\, m_t^{\max}. \tag{1}$$

Eq. (2), introduced and proved in [10], generalizes Thm. 1 and provides m_i, the expected number of unique points in column i, given m_0:

$$m_i \approx \frac{2N}{i + \frac{2N}{m_0}}. \tag{2}$$

By setting $m_0 = r m_t^{\max}$, one can combine Eq. (1) and Eq. (2) to derive Eq. (3) (see proof and details in [10]):

$$r \approx \frac{\alpha}{1-\alpha}. \tag{3}$$

This helps determine the target number of chains m_0 to generate in order to create a clean table with m_t chains and a maximality factor α:

$$m_0 = \frac{m_t}{1-\alpha} \tag{4}$$

2.4 Success probability

The *coverage* of a given matrix is the ratio between the number of its unique values and N. The coverage of a matrix is trivially equivalent to the *success probability* of the attack phase: a value in the matrix will always be recovered, and conversely. The success probability of a table with m chains of length t is given in [31] for maximal rainbow tables and is adapted for non-maximal rainbow tables with m_t chains and $t+1$ columns in Eq. (5).

$$p = 1 - \left(1 - \frac{m_t}{N}\right)^t. \tag{5}$$

With $m_t = m_t^{\max}$, $p \approx 1 - e^{-2} \approx 0.86$, which is an upper bound [31]. In order to obtain a higher success probability, a set of ℓ independent

[1] Also known as *perfect* rainbow table.

tables should be used instead of a single one. For ℓ tables, the success probability is thus given by Eq. (6) [31]:

$$p_\ell = 1 - (1-p)^\ell. \tag{6}$$

2.5 Filtration

As explained in Sect. 2.2, among the m_0 chains used at the beginning of the precomputation phase, only $m_t \ll m_0$ chains remain after cleaning. Therefore $(m_0 - m_t)\cdot t$ hash operations – which represents most of the effort – are wasted.

The paper [10] addresses this issue by introducing a technique named *filtration*, which significantly reduces the number of unnecessary hash operations during the precomputation phase. The filtration method consists in progressively cleaning the rainbow matrix in well-chosen columns instead of once at the end. Pushed to its limit, filtration in every column leads to Thm. 2 [10].

Theorem 2. *Given the factor $r = \frac{m_0}{m_t^{\max}} \ll t$, the number of columns $t+1$, the minimum precomputation cost is:*

$$P_{\min} = \sum_{i=0}^{t-1} m_i \approx 2N\ln(1+r).$$

Filtering in every column is not advisable however, due to the overhead of the filtration process [10]. Cleaning the tables in a limited number of well-chosen columns is already sufficient to be very close to the theoretical lower bound without incurring a significant overhead. For instance, filtering in 25 optimally-placed columns leads to a precomputation cost of $P \approx 1.13 \times P_{\min}$[10].

2.6 Attack Phase

The attack phase is divided into t steps (per table). One first assumes that x^* belongs to column $t-1$ (of a given table) and tests the hypothesis as described below. If not, one proceeds similarly to column $t-2$, and so on, until either a valid preimage is found, or until reaching the beginning of the table.

Checking whether x^* is in column $t-1$ consists in checking whether $R_t(y)$ matches an end point $x_{t,j}$ (since the table is clean, there can be at most one such end point). If so, the chain j is rebuilt from the corresponding start point $x_{0,j}$ up to $x_{t-1,j}$. If $h(x_{t-1,j}) = y$ (a case called a *true alarm*), the attack succeeds and the preimage of y is $x_{t-1,j}$. If not (a case called a *false alarm*), the attack proceeds to the next step. In the next step, $f_t(R_{t-1}(y))$ is computed and, again, the attacker checks for a matching end point. In the i-th step (corresponding to column $t-i$), $f_t(f_{t-1}(\dots f_{t-i+1}(R_{t-i}(y))))$ is computed.

The average number of hash operations needed to perform a search in column c is given by Prop. 1.

Proposition 1. *For a given column c, the average number of hash operations C_c needed to perform a search is:*

$$C_c = t - c\prod_{i=c}^{t}\left(1 - \frac{m_i}{N}\right).$$

Proof. A proof for maximal tables can be found in [13]. The result is here generalized to non-maximal rainbow tables, as in the proof of Thm. 5. □

Once the search cost in a given column c is known, the average number of hash operations for an entire attack using ℓ tables can be deduced. Thm. 3 gives a general expression of the average attack phase cost.

Theorem 3. *The average number of hash operations T required to perform an attack using ℓ rainbow tables of length t, given a search space of size N is:*

$$T = \ell\sum_{c=1}^{t}\left(\frac{m_t}{N}\left(1-\frac{m_t}{N}\right)^{\ell(c-1)}\sum_{j=1}^{c}C_{t-j+1}\right) + \ell\left(1-\frac{m_t}{N}\right)^t\sum_{c=1}^{t}C_c.$$

Proof. See [13]. □

2.7 Memory Used

Due to the characteristic complexity of the attack phase, $O(N^2/M^2)$, optimizing the use of the memory is paramount. Different storage optimizations can be used to store rainbow tables [6, 12, 13]. In [6], the authors show that a very effective method to store rainbow tables, the *compressed delta encoding* method, allows to store tables with a memory very close to the theoretical lower bound (approx. 0.66% [6]). We use this lower bound in the rest of this paper, as it simplifies discussions, and is extremely close to the practical storage cost using compressed delta encoding.

Near-optimal storage of rainbow tables consists in *sorting* and *compressing* the end points (using ad-hoc compression), and storing the start points as-is on $\lceil\log_2(m_0)\rceil$ bits [6]. The total cost is given by Eq. (7):

$$\begin{aligned} M^{RT} &= M_{sp}^{RT} + M_{ep}^{RT} \\ &= \ell\left[m_t\lceil\log_2(m_0)\rceil + \log_2\binom{N}{m_t}\right]. \end{aligned} \tag{7}$$

3 STEPPED RAINBOW TABLES

During the cleaning process of rainbow tables, most computed chains are discarded due to merges. Our key idea consists in recycling some of these merged chains instead of discarding them. The recycled chains are shorter than regular chains because the merged parts are removed. Such tables may contain several such "steps", leading to so-called *stepped rainbow tables*. For the sake of clarity, they are first introduced with a single step in Sect. 3.1, then generalized to the multiple steps in Sect. 3.2.

3.1 Stepped Rainbow Tables with a Single Step

3.1.1 Precomputation phase. Precomputation is similar to that of classic rainbow tables with the following twist. Once column s is reached, a filtration is performed and *intermediary end points* are temporarily stored. At each subsequent filtration, when chains merge, one is kept for further extension while the rest are cut short. The chains that are cut short have length s and their intermediary end points are their final end point. The ones that still remain after the final filtration in column t have length t and their final end point is their value in column t. Fig. 2 illustrates the construction of a stepped rainbow table. The solid blue curve is m_c, the number of unique points remaining in column c. The area under the dashed red curve represents the amount of hash operations needed to generate the table using the filtration method (each landing of the curve

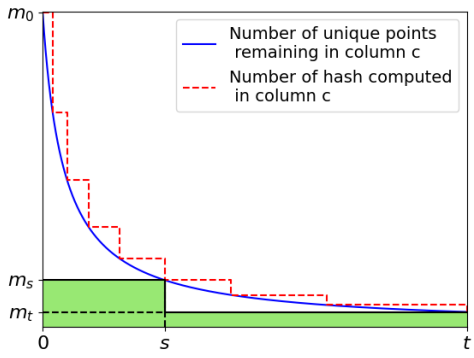


Figure 2: Construction of stepped rainbow matrix with single step.

corresponds to the application of a filter). The stepped rainbow matrix obtained at the end of precomputation phase is colored in green.

3.1.2 Attack Phase. In a classic rainbow table, all columns have the same probability of raising a true alarm, but the further to the right the column is, the lower is the cost of searching in that column (in other words, C_c is *decreasing*). Therefore, the most effective search order is from the last column t and in subsequent decreasing order.

For a 1-step stepped rainbow table, the probability of finding the preimage depends on the column index, because not all columns contain the same number of points: columns 0 to s contain m_s points and columns $s+1$ to t contain m_t points.

Moreover, when an alarm (true or false) occurs in a column c with $c < s$, it can either be detected in column s or in column t. It is detected in column s if a match occurs with one of the chains of length s, otherwise the alarm is detected in column t. The number of cryptographic operations needed to obtain an alarm is consequently not the same in all columns. Additionally, the number of cryptographic operations to check whether an alarm is true or false similarly depends whether the alarm was raised in column s or in column t.

Consequently, with 1-step stepped rainbow tables, instead of exploring the table from column $t-1$ to column 0, the search is performed in an order that minimizes the total expected time. This corresponds to searching in the non-explored column with the highest success probability over average cost ratio[2].

The procedure to perform the attack phase with 1-step stepped rainbow table is provided in Algo. 1. The definition of the function μ that appears in Algo. 1 is provided in Def. 3. In a nutshell, this function returns the index of the most promising column for the forthcoming search.

3.2 Stepped Rainbow Tables with τ Steps

The concept of stepped rainbow tables can be naturally generalized to $\tau > 1$ steps. Fig. 3 illustrates the resulting structure. In what follows, the columns that specify the steps are denoted s_i with $0 < i \leqslant \tau$, with $s_i < s_j$ for $i < j$.

In the stepped rainbow table, m_{s_1} end points and their corresponding start points are stored: for each step s_i with $0 < i < \tau$, $m_{s_i} - m_{s_{i+1}}$ chains are stored with a length s_i, and m_t chains of length t are also stored.

The attack phase using τ steps is similar to Algo. 1. Algo. 2, provided in Appendix B, depicts the attack phase of the generalized stepped rainbow tables.

Algorithm 1: Attack Phase (1-step)

Input : $y \in B : y = h(x^*) \in A$
Output: x^* (success) or $\perp$ (failure)

$stepsList \leftarrow [s, t]$
$EP \leftarrow [[EP_1 ... EP_{m_s - m_t}], [EP_{m_s - m_t + 1} ... EP_{m_s}]]$
$SP \leftarrow [[SP_1 ... SP_{m_s - m_t}], [SP_{m_s - m_t + 1} ... SP_{m_s}]]$
$c \leftarrow [s, t]$
$v \leftarrow [0 : \text{len}(c) - 1]$
$success \leftarrow$ False
while $c \neq [0, 0]$ **and** $success$ = False **do**
  $alarm \leftarrow$ False
  for $d = 0$ **to** $\text{len}(c) - 1$ **do**
    $v[d] \leftarrow \mu(c[d])$
  end
  $j \leftarrow c[v.\text{index}(\max(v))]$
  $i \leftarrow j$
  $x_i \leftarrow R_i(y)$
  while $i \leqslant t$ **and** $alarm$ = False **do**
    if $i \in stepsList$ **then**
      $index_i \leftarrow stepsList.\text{index}(i)$
      if $x_i \in EP[index_i]$ **then**
        $alarm \leftarrow$ True
        $x \leftarrow SP[EP[index_i].\text{index}(x_i)]$
        for $g = 1$ **to** $j - 1$ **do**
          $x \leftarrow f_g(x)$
        end
        if $h(x) = y$ **then**
          $success \leftarrow$ True
        end
      end
    end
    if $alarm$ = False **and** $success$ = False **then**
      $x_i \leftarrow f_i(x_i)$
      $i \leftarrow i + 1$
    end
  end
  if $success$ = False **then**
    $c[c.\text{index}(j)] \leftarrow c[c.\text{index}(j)] - 1$
  else
    return $(success, x)$
  end
end
return $(success, 0)$

[2]Another example of non-monotonic search order is discussed in the analysis of rainbow tables with heterogeneous widths [7]. Optimality of the "success probability over average cost ratio" as a decision metric is also discussed there.

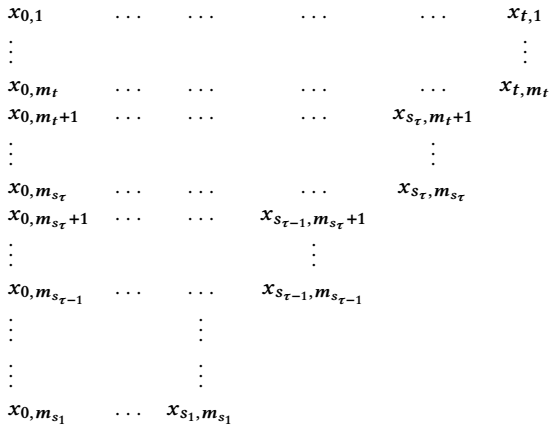


Figure 3: Stepped rainbow matrix with τ steps.

4 ANALYSIS OF STEPPED RAINBOW TABLES

This section presents an analysis of the characteristics of stepped rainbow tables.

4.1 Notations

Definition 1. *We note $k(c)$ the index of the leftmost step that is to the right of column c, i.e., $s_{k(c)-1} \leqslant c < s_{k(c)}$.*

Def. 1 is useful to express some quantities concisely, in particular: there are $m_{s_{k(c)}}$ points in the step immediately to the right of column c; similarly, there are $m_{s_{k(c)}} - m_{s_{k(c)+1}}$ chains of length $s_{k(c)}$ there.

Where appropriate, $s_{\tau+1}$ refers to column t and s_0 refers to column 0. This convention allows for some results to be presented more clearly and concisely.

The ratio ρ is also introduced in Def. 2 for the purpose of making subsequent results more concise.

Definition 2. *Given a column c and a step s_i of a stepped rainbow table, the proportion of chains with length s_i in column c is given by $\rho_{i,k(c)}$ with $\rho_{i,j}$ such that:*

$$\rho_{i,j} = \begin{cases} \frac{m_{s_i} - m_{s_{i+1}}}{m_{s_j}} & i \leqslant \tau \\ \frac{m_t}{m_{s_j}} & i = \tau + 1 \end{cases}$$

In what follows, the *attack chain* refers to the chain built from y. The attack chain starting in column c and ending in column t is thus the chain finishing by the value $f_t(f_{t-1}(\dots f_{c+1}(R_c(y))))$.

4.2 Success Probability

Theorem 4. *Given τ steps noted s_i with $0 < i \leqslant \tau$, $s_0 = 0$ and $s_{\tau+1} = t$, and considering m_{s_i} the number of unique elements in column s_i, the success probability p of a single clean stepped rainbow table is:*

$$p = 1 - \prod_{i=1}^{\tau+1} \left(1 - \frac{m_{s_i}}{N}\right)^{s_i - s_{i-1}}.$$

Proof. Each column c covers $m_{s_k(c)}$ different elements. Following a similar argument as for Eq. (5), we write:

$$p = 1 - \prod_{c=1}^{t} \left(1 - \frac{m_{s_{k(c)}}}{N}\right).$$

Using the fact that $s_{k(c)}$ (and therefore $m_{s_{k(c)}}$) is equal for all c between two steps, we can group these factors together, resulting in the expression in Thm. 4. □

The success probability for ℓ stepped rainbow tables p_ℓ is then directly obtained from Thm. 4, and is given by Eq. (8)

$$p_\ell = 1 - (1-p)^\ell. \tag{8}$$

4.3 Precomputation Time

Constructing stepped rainbow tables costs the same number of hash operations as classic rainbow tables (see Sect. 2.5). Whether intermediary steps are saved during the precomputation phase or not has no bearing on the number of hash computations (only the number m_0 of start points considered at the beginning of generation and the number and positions of filters matter). As shown in Sect. 6.4, the overhead due to filtration or the storage of intermediary points is insignificant.

A distinguishing feature of stepped rainbow tables however is that, for a given target memory and probability of success, m_0 is typically much smaller than with classic tables. All other things being equal, this significantly reduces the precomputation time.

The figure $P = 1.13 \times P_{\text{min}}$ is used to quantify the number of hash operations performed in this phase (for both stepped rainbow and classic tables, in accordance with [10]). As discussed in Sect. 5, our experimental results closely match this estimation.

4.4 Attack Time

Contrarily to rainbow tables, the search order is not monotonic. The cost in each column has an associated average cost, and a given probability that the search succeeds. The average attack time corresponds to the sum of the cost of the search in all columns, visited in optimal order, and weighted by the probability that the search stops there.

As in the classic case, a search in a given column either leads to a *true alarm* (successful search), a *false alarm* or to *no alarm*. An analysis of these probabilities and the associated costs is the focus of sections 4.4.1 to 4.4.6.

A useful intermediary result is the probability that no merge occurs between two given columns. This is one of the fundamental results in [31] and is generalized to any two columns c and c' in Lemma 1.

Lemma 1. *Given two columns c and c' with $c < c' \leqslant t$, the probability that the attack chain does not merge with any chain of the rainbow matrix by c', given it had not merged in or before c, is:*

$$p_{nomrg}(c, c') = \prod_{i=c+1}^{c'} \left(1 - \frac{m_i}{N}\right). \tag{9}$$

4.4.1 Probability of True Alarm. A true alarm occurs when the start point of the attack chain appears in the corresponding column of the stepped rainbow matrix.

Proposition 2. *The probability that a true alarm occurs when starting the attack chain in c is:*

$$p_{find}(c) = \frac{m_{s_{k(c)}}}{N}.$$

Proof. There are $m_{s_{k(c)}}$ different elements in column c of the stepped rainbow matrix. Given that elements are uniformly distributed and that the number of elements in the solutions space is N, the probability that the target x^* is an element of a column c is the stated quantity. □

4.4.2 Probability of False Alarm. In stepped rainbow tables, a false alarm occurs when the attack chain at column c merges with a chain of the matrix. This is similar to classic rainbow tables, with the exception that this merge can be observed at any of the steps $s_i > c$, including indeed $s_{\tau+1} = t$, but not exclusively.

There is a distinction to be made as to whether the merge occurs before $s_{k(c)}$ or after. The two cases are analyzed in Props. 3 and 4.

Proposition 3. *The probability to raise a false alarm due to a merge between columns c and $s_{k(c)}$ is*

$$p_{fa\text{-}pre}(c) = 1 - p_{find}(c) - p_{nomrg}(c, s_{k(c)}).$$

Proof. Straightforward, as these events (no alarm, true alarm, false alarm) are mutually exclusive. □

Proposition 4. *The probability to raise a false alarm due to a merge between columns s_i an s_j, with $c \leqslant s_{k(c)} < s_i < s_j$, is:*

$$p_{fa\text{-}post}(c, s_i, s_j) = p_{nomrg}(c, s_i) - p_{nomrg}(c, s_j).$$

Proof. A false alarm due to a merge between columns s_i and s_j means that no merge occurs between c and s_i but one occurs between s_i and s_j. Writing E_1 and E_2 respectively the events "no merge occurs between c and s_i" and "a merge occurs between s_i and s_j", we have:

$$\begin{aligned} p_{\text{fa-post}}(c, s_i, s_j) &= \Pr(E_1 \wedge E_2) \\ &= \Pr(E_1) \times \Pr(E_2 \mid E_1) \\ &= p_{\text{nomrg}}(c, s_i)\left(1 - p_{\text{nomrg}}(s_i, s_j)\right) \\ &= p_{\text{nomrg}}(c, s_i) - p_{\text{nomrg}}(c, s_i) p_{\text{nomrg}}(s_i, s_j) \\ &= p_{\text{nomrg}}(c, s_i) - p_{\text{nomrg}}(c, s_j) \end{aligned}$$

The last equality stems from the nature of Eq. (9). □

4.4.3 Probability of No Alarms. When no match occurs between the attack chain and end points of a table, no alarm is raised.

Proposition 5. *The probability that no alarm occurs between a column c and column t is*

$$p_{noalarm}(c) = p_{nomrg}(c, t).$$

Proof. For no alarm to happen, the attack chain must not merge with the stepped rainbow matrix at any point between its start in column c and its end in column t. □

4.4.4 Cost of Alarms. The cost of an alarm, i.e., the number of cryptographic operations performed when an alarm occurs, is the same for true and false alarms. In both cases, an entire chain has to be rebuilt: firstly from column c, where the search is performed, to a step s_i or to column t (depending where the merge is detected) and then from column 0 to column $c-1$. Therefore, the cost of an alarm does not depends on its nature (true or false alarm) but only on the column in which it is detected. The cost of an alarm is given in Prop. 6 using Def. 2.

Proposition 6. *Given a search performed in a column c and $k(c)$ the index of the leftmost step that is to the right of column c, the number of hash operations needed to rule out a false alarm is:*

$$\sum_{i=k(c)}^{\tau+1} \rho_{i,k(c)} s_i.$$

Proof. $\rho_{i,k(c)}$ is the proportion of chains with length s_i in column c. It corresponds to the number of chains with a length s_i in column s_i divided by the total number of chains in column s_i. Given c and $k(c)$ as $s_{k(c)-1} \leqslant c < s_{k(c)}$, alarms are not necessarily detected in step $s_{k(c)}$ but can be detected in each step $s_j > s_{k(c)}$. If a merge occurs in column j with $c < j \leqslant s_i$, the probability that this merge is detected in step s_i is $\rho_{i,k(c)}$.

If an alarm is detected in step s_i, a chain of length s_i has to be rebuilt, which costs s_i hash operations. Therefore, the average cost of an alarm in a column c is the sum of the probability to detect the alarm in each step s_i with $c \leqslant s_i$, multiplied by the number of hash operations needed to rule it out, depending on the step in which the alarm is detected. □

4.4.5 Cost of No Alarm. The cost of no alarm in a given column c is $(t-c)$. If no alarm occurs in any steps, no chain will be rebuilt, therefore only $(t-c)$ hash operations will be computed.

4.4.6 Total Time of a Search in a Given Column. The number of operations needed to perform a search in a column c is given by Thm. 5.

Theorem 5. *For a given column c and the index $k(c)$, the average number of cryptographic operations C_c needed to perform a search is: For $s_\tau < c \leqslant t$:*

$$C_c = t - c p_{noalarm}(c).$$

For $c \leqslant s_\tau$:

$$\begin{aligned} C_c = &\left(1 - p_{nomrg}(c, s_{k(c)})\right) \sum_{i=k(c)}^{\tau+1} \rho_{i,k(c)} s_i \\ &+ \sum_{j=k(c)}^{\tau} p_{fa\text{-}post}(c, s_j, s_{j+1}) \sum_{i=j+1}^{\tau+1} \rho_{i,j+1} s_i \\ &+ (t-c)\, p_{noalarm}(c). \end{aligned}$$

Proof. See Appendix A □

4.4.7 Average Attack Time. As explained in Sect. 3.1.2, a metric μ (corresponding to probability of success over average cost ratio) is used to determine the most efficient column in which to perform each iteration of the search. Using μ as defined in Def. 3, vector v is defined in Def. 4. Vector v is composed of all columns of a stepped rainbow table from the most to least efficient.

Definition 3. *Given N and a column c with $0 \leqslant c \leqslant t$:*

$$\mu(c) = \frac{m_{s_{k(c)}}}{NC_c}.$$

Definition 4. *Given a stepped rainbow table with t columns, the vector v with $v = [v_1, v_2, \ldots, v_t]$ is obtained by sorting the columns $[1, \ldots, t]$ in decreasing order of $\mu(c)$.*

The average attack time i.e., the average number of hash operations needed to perform an attack with a set of ℓ stepped rainbow tables, is given by Thm. 6.

Theorem 6. *Given N, ℓ stepped rainbow tables with τ steps, and considering its vector $v = [v_1, v_2, \ldots v_t]$ ordering the columns of tables, the average number of hash operations T required to perform an attack is:*

$$T = \ell \sum_{c=1}^{t} \left(\frac{m_{v_c}}{N} \prod_{i=1}^{c-1} \left(1 - \frac{m_{v_i}}{N}\right) \sum_{j=1}^{c} C_{v_{t-j+1}} \right) + \ell \prod_{i=1}^{t} \left(1 - \frac{m_{v_i}}{N}\right) \sum_{c=1}^{t} C_{v_c}.$$

with m_{v_c} the number of points in column c, and with $s_{v_{c-1}} \leqslant c < s_{v_c}$.

Proof. This expression is a generalization of Thm. 3. The proof is constructed using an approach similar to the one used in [31]. T is obtained by adding on the one hand, the success probability of the attack using ℓ tables, multiplied by its average cost, and on the other hand, the failure probability of the attack using ℓ tables, multiplied by the cost of a failed search.

The first term is obtained by multiplying for each column c, the probability of true alarm in the column, with the probability of no true alarm in all earlier iterations:

$$\frac{m_{v_c}}{N} \prod_{i=1}^{c-1} \left(1 - \frac{m_{v_i}}{N}\right)$$

This is multiplied by the cost of all searches performed until reaching this column: $\sum_{j=1}^{c} C_{v_{t-j+1}}$.

The second term is obtained by multiplying the failure probability using ℓ tables, namely $\ell \prod_{i=1}^{t} \left(1 - \frac{m_{v_i}}{N}\right)$, with the cost of performing a search in all columns of a table, namely $\sum_{c=1}^{t} C_{v_{t-j+1}}$. □

4.5 Memory Used

Formulas to evaluate the memory needed to store rainbow tables are presented in [6]. In this section, these are adapted to stepped rainbow tables.

4.5.1 Rationale. When using stepped rainbow tables, instead of considering one set of points per table, end points and start points are grouped according to their step. For the same number of chains and the same number of tables, storage of stepped rainbow tables end points thus takes more memory than rainbow tables. Indeed, the points to be stored are divided into independent collections as opposed to a single collection, resulting in less efficient compression.

On the other hand, in order to generate the same number of chains and the same number of tables, much fewer start points need to be considered at the beginning of precomputation. Thus for a given number of chains (regardless of their size), m_0 is smaller for stepped rainbow tables than for classic tables. As the memory needed for storing start points depends on m_0, the storage of start points ends up using less memory for stepped rainbow tables than for rainbow tables.

Overall, the memory for stepped rainbow tables compared to classic tables depends on specific parameters (See Sect. 6.5.3 for discussion). Nevertheless, in this paper we always choose parameters to compare with rainbow tables for the same coverage and memory.

4.5.2 Analysis. For each step s_i, there are $m_{s_i} - m_{s_{i+1}}$ points to be stored. These are in addition to the m_t chains of length t.

The memory needed to store the table end points is given by Eq. (10), adapted for stepped rainbow tables from [6].

$$M_{ep}^{SRT} = \ell \left(\log_2 \binom{N}{m_t} + \sum_{i=1}^{\tau} \log_2 \binom{N}{m_{s_i} - m_{s_{i+1}}} \right). \tag{10}$$

As for start points, m_{s_1} points are stored for each table instead of m_t. The storage of individual start points is the same as in the classic case. The memory needed to store stepped rainbow tables start points, M_{sp}^{SRT}, is given in Eq. (11)):

$$M_{sp}^{SRT} = \ell m_{s_1} \lceil \log_2(m_0) \rceil. \tag{11}$$

The total memory used to store ℓ stepped rainbow tables M^{SRT} is simply $M^{SRT} = M_{sp}^{SRT} + M_{ep}^{SRT}$.

5 EXPERIMENTS

This section illustrates the theoretical results obtained in Sect. 4 with practical experiments. Stepped rainbow tables with between 1 to 5 steps have been generated, and their precomputation time, success probability, and attack time have been compared to the respective theoretical results. In all cases, when a sufficient number of experiments have been done, the relative difference between theoretical and practical results is below 1%.

It is worth noting that the precomputation time (number of hash operations) is the same for both rainbow tables and stepped rainbow tables when considering equal parameters, namely number of start points, number and positions of filters, and value of t. The comparison that follows consequently focuses on attack time and success probability only.

In what follows, for the sake of clarity, we call *configuration* a list of parameters describing a set of rainbow tables: maximality factor of the tables, number of columns, and number of tables, respectively denoted α, t, and ℓ. When considering stepped rainbow tables, the configuration also contains the number of steps, denoted τ, and their positions $s_1, \ldots, s_\tau$.

5.1 Availability

The code used to generate stepped rainbow tables and perform attacks with them is publicly available on GitHub[3]. It provides a python script that launches stepped rainbow tables generation for parameters (i.e., N, ℓ, α, t and step positions) given in the script header, and performs attacks using these generated tables. The number of attacks to perform has to be specified. Log files are

[3] https://github.com/DianeLeblancAlbarel/Stairway-To-Rainbow

created for tracing success probability, precomputation time and attack time.

5.2 Environment

Experiments were conducted on a computer hosting two AMD EPYC 7H12 processors composed of 64 physical cores and 64 virtual cores each, for a total of 128 physical cores and 128 virtual cores. We used up to 128 of the physical cores in our experiments.

The precomputation phase has been distributed as proposed in [10] using the filtration method adapted on stepped rainbow tables. Both the precomputation and attack phases were written in C, using function SHA256 of OpenSSL for hashing. The Open MPI library was used to distribute the precomputation phase.

5.3 Success Probability

Theorem 4 is used to choose stepped rainbow tables configurations that reach a target success probability.

Experiments with different configurations were performed, containing up to 5 steps positioned between columns $0.3t$ and t. The positions of the steps are not critical in these experiments, given that there is no "optimal" configuration. Indeed, the position of the steps defines a trade-off between precomputation time and the attack time. Nevertheless, configurations with widely different step positions were tested to cover various cases: concentrated on the right, concentrated on the left, uniformly distributed, etc. Success probabilities were tested from 85% to 99.5%. All experiments lead to the same conclusions. For brevity, only the case using 2 tables and a 95% success probability is presented here. The probability of 95% have been chosen arbitrary but same results are obtained with other target success probabilities.

Fig. 4 depicts the success probabilities when considering experiments based on ℓ stepped rainbow tables with $\ell = 2$. The blue horizontal line in the figure is the theoretical success probability, chosen to be 95%.

A test consists in randomly choosing one of the N elements of the space A, hashing it, and trying to find its preimage using the generated stepped rainbow tables. In Fig. 4, 25 configurations are tested. A batch of 10 000 attacks was performed for each configuration. Each point of the figure represents a different tested configuration. The success probability obtained corresponds to the proportion of the attacks in the batch that succeeded.

For all configurations, the obtained success probabilities are all between 94.7% and 95.4%.

5.4 Attack Time

To test the compliance of the attack time with the model given in Sect. 4, various configurations have been tested. For each configuration tested, 25 batches of 20 000 attacks were performed, i.e., 500 000 attacks per configuration. For brevity and clarity of exposition, we display a random selection of 25 among the large sample of tested configurations. Tests were made for the same coverage and memory, though experiments performed for different memory and coverage lead to same conclusion. The 25 configurations presented here use 5 steps, 2 tables, and a 95% success probability. The differences between each configuration tested are various values of the parameters α and t, and steps positions. The target probability and

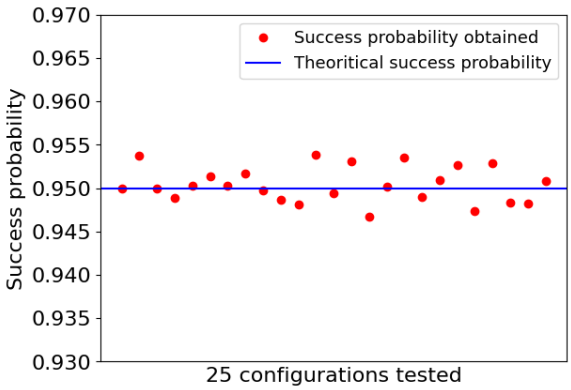


Figure 4: Experimental success probability according to the configuration used, with theoretical success probability equal to 95%.

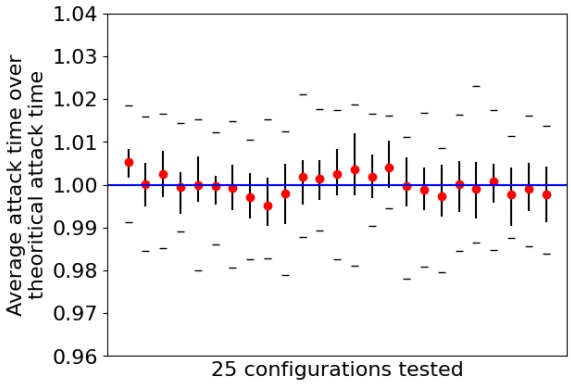


Figure 5: Average number of hash for an attack over theoretical number of hash for different configurations.

memory are the same. The memory M and N have been arbitrarily chosen to correspond to reasonable cases, namely $M = 63.9$ GB and $N = 2^{42}$. The position of the steps varies between $0.3t$ and t (See Sect. 6.3 for explanations about this bound).

Fig. 5 presents the results obtained. For each tested configuration, the average attack time of the configuration corresponds to the average number of hashes performed over the 500 000 attacks. Each point represents the ratio between the average attack time of each configuration and the theoretical counterpart given by Thm. 6.

The black line represents for each point the first and third quartile obtained on each of the 25 batches of 20 000 attacks. The black dashes represent quartiles 0 and 4.

The results are well distributed around 1.0. The large majority of experiments (between first and third quartile) are within ±1.5%. Fig. 5 leads to conclude that experimental results fit the theory very well. It is worth noting that significant variations in the attack time are intrinsic to rainbow tables and are noticeable for both stepped rainbow and classical rainbow tables.

6 EVALUATION

This section compares performances of rainbow tables and stepped rainbow tables. The comparison methodology is first introduced, then the configurations used for the comparison are established, the results are presented, and the compatibility of the stepped rainbow tables with existing improvements is finally discussed.

6.1 Comparison Methodology

The concept of *configuration* is introduced in Sect. 5. Defining such a configuration is needed prior to generating a set of tables. As concluded in [11], the precomputation time is usually the bottleneck that prevents the use of large-scale TMTOs. Nonetheless, the attack time is of primary importance as well. The comparison thus focuses on the trade-off between these two constraints.

Various trade-off curves are compared. For each comparison, a target success probability and an available memory for the attack phase are set. These parameters being fixed, the *best configuration* refers to the configuration for which there is no other configuration having *both* better precomputation time and better attack time, whether we consider rainbow tables or stepped rainbow tables. These configurations define the *Pareto frontier* for each comparison.

We describe costs (both precomputation and attack phases) in terms of number of hash operations. This allows for better accuracy, and can be transposed to any computing environment. A good approximation of the time in seconds for the precomputation or attack phase is indeed simply the number of hash operations to be performed divided by the number of hash operations computable per second on the chosen environment. As shown in [10], this approximation may vary slightly for the precomputation phase depending on the distributed environment used. For the attack phase, this approximation is very accurate.

6.2 Configurations

This section presents how the configurations for the comparisons were determined.

6.2.1 Rainbow Tables. As previously explained in Sect. 5, three parameters, namely α, t, and ℓ are sufficient to fully define the characteristics of a rainbow table. The success probability of a single table and the memory available for the attack phase, respectively given by Eq. (5) and Eq. (7), are fixed by α and t. Only the number of tables, ℓ, remains free. As a consequence, for any given value[4] for ℓ, there exist a unique configuration once α and t are chosen.

Typically, the number ℓ of tables chosen, is the smallest or second smallest value that allows reaching the expected success probability[5] given by Eq. (6). In practice, only a small range of values for ℓ is used, since, above a certain number of tables (e.g., 6 tables and up), adding a table does not reduce the precomputation time significantly but does increases the attack significantly. Therefore, in practice, for a given success probability and a given memory for the attack phase, only a few configurations of rainbow tables are meaningful.

[4]Barring those for which the target success probability or memory are unattainable.
[5]Certain success probabilities are only achievable by using a sufficient number of tables.

6.2.2 Stepped Rainbow Tables. By using stepped rainbow tables instead of rainbow tables, new parameters are available for determining possible configurations: the number of steps τ, and their positions. Considering various numbers of steps and their possible positions, a large number of stepped rainbow tables configurations can reach the expected success probability and memory. When considering τ steps, the number of possible configurations is technically bounded by $(t-1)^\tau$. Among these, many provide better performance than rainbow tables. However, some of them perform worse. This often arises for example when the steps are mostly located towards the left part of the chains. Indeed, in such a case, a lot of very short chains are stored, causing a significant increase in memory without significantly increasing the success probability.

6.3 Parameters

We considered configurations with 1 to 5 steps positioned between $0.3t$ and t. Placing steps further to the left of the table (in columns smaller than about $0.3t$) is possible and the formulas apply perfectly to these cases, but in practice these tables are very inefficient and are therefore never used. Using more than 5 steps is possible but we have observed that it does not improve the trade-off between precomputation time and attack time sufficiently to justify the effort in finding the optimal step positions. To determine the configurations that reach the success probability and the available memory, we performed a search according to Algo. 3 available in Appendix C.

We chose a set of size $N = 2^{42}$ and a memory of size 63.9 GB distributed over ℓ tables. $N = 2^{42}$ has been used to provide an easy comparison with articles[5, 10], also dealing with this space. The memory was arbitrarily chosen to enable use of our results in practical cases.

For each evaluation we performed, we identified the configurations that reach the expected success probability and available memory according to Algo. 3. For all configurations, the precomputation time and the attack time are plotted, both expressed in number of hash operations.

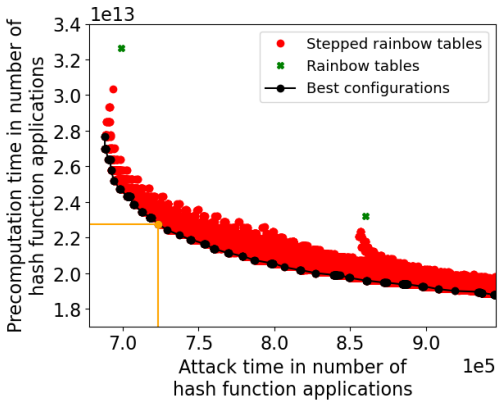
6.4 Results

Results are presented in Fig. 6a to 6c, which correspond to success probabilities of 96%, 98%, and 99.5%, respectively. The (green) crosses in the figures represent the results for rainbow tables and the (red) dots, the results for stepped rainbow tables. Black dots identify the best results, which indeed all correspond to stepped rainbow tables.

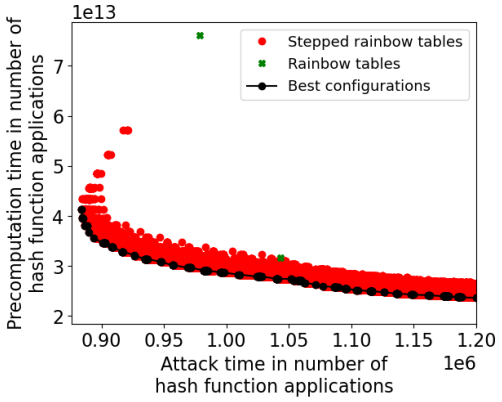
It is worth noting that the precomputation times differ between Fig. 6a to 6c depending on both the number of chains and the number of tables to generate to reach the target coverage given the target memory.

The main highlight is that, no matter the success probability, there is always a configuration where stepped rainbow tables behave better than rainbow tables. In other words, when comparing rainbow tables and stepped rainbow tables for a same memory and success probability, rainbow tables are not on the Pareto frontier of configurations.

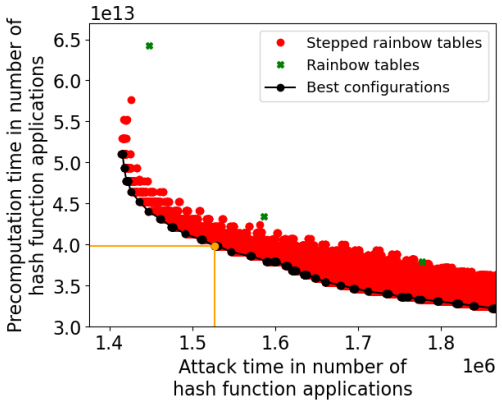
The relative gain depends on the success probability, which is mostly due to the maximality factor α. Indeed, whatever the success probability, using stepped rainbow tables reduces α and thus the



(a) Trade off between precomputation time and attack time 96% of success, $N = 2^{42}$ and a 63.9 GB memory. 2 and 3 tables used.



(b) Trade off between precomputation time and attack time 98% of success, $N = 2^{42}$ and a 63.9 GB memory. 2, 3 and 4 tables used.



(c) Trade off between precomputation time and attack time 99.5% of success, $N = 2^{42}$ and a 63.9 GB memory. 3 to 5 tables used.

Figure 6: Trade-off between precomputation and attack.

Table 1: Expected gain illustrated on several examples with SRT meaning Stepped Rainbow Tables, and RT meaning Rainbow Tables. Precomputation and attack phase numbers are quantity of cryptographic operations.

Success probability: 96%		
	Precomputation	Attack
2 SRT	2.46×10^{13}	6.99×10^{5}
2 RT	3.26×10^{13}	
Gain	25%	
2 SRT	2.32×10^{13}	7.17×10^{5}
3 RT		8.6×10^{5}
Gain		17%
Success probability: 98%		
	Precomputation	Attack
2 SRT	2.96×10^{13}	9.79×10^{5}
2 RT	7.59×10^{13}	
Gain	61%	
2 SRT	3.17×10^{13}	1.04×10^{6}
3 RT		9.26×10^{5}
Gain		11%
Success probability: 99.5%		
	Precomputation	Attack
3 SRT	4.4×10^{13}	1.45×10^{6}
3 RT	6.42×10^{13}	
Gain	31%	
3 SRT	4.35×10^{13}	1.59×10^{6}
4 RT		1.46×10^{5}
Gain		8%

number of chains to be computed during the precomputation phase. Therefore the gain in precomputation time is even more important when the success probability requires a larger maximality factor.

For example, Fig. 6b for success probability of 98%, using two stepped rainbow tables instead of two rainbow tables divides the precomputation time by 2.56 without increasing the attack time (as all of these comparisons, for the same coverage and the same memory used). On the environment described in Sect. 5.2, this correspond to generating a set of tables in approximately 5.6 hours, instead of 14.4 hours. If three classic rainbow tables are used instead of two, stepped rainbow tables perform 11% faster. On a single core, these attack times correspond to approximately 0.08 second per attack.

Table 1 provides the expected gains of several configuration examples with various success probabilities.

We compared so far stepped rainbow tables and rainbow tables either for the same attack time or for the same precomputation time. However, Fig. 6a to 6c shows that many other configurations ("best

configurations", represented by black dots) may also be interesting in practice.

For instance, for a success probability of 99.5%, in the configuration identified by an orange dot in Fig. 6c, the use of stepped rainbow tables results in a precomputation time of 3.98×10^{13} hash operations and an attack time of 1.53×10^{6} hash operations. Compared to the 3-tables rainbow table configuration, this stepped rainbow tables configuration allows a reduction of 38% in the precomputation time for an attack time phase only 5% slower.

In the same vein, in Fig. 6a, the stepped rainbow tables configuration identified by an orange dot has a much faster attack time than the 3-table rainbow table configuration (right green cross) and in the same time has a slightly faster precomputation time. Indeed, using this stepped rainbow tables configuration instead of 3 rainbow tables decreases by 2% the precomputation time, and decreases by 16% the attack time. Compared to the 2-tables rainbow table configuration (left green cross), this stepped rainbow tables configuration allows a reduction of more than 30% in the precomputation time for an attack time phase only 4% slower.

6.5 Conclusion on Efficiency Comparison

6.5.1 Coverage. As seen in Sect. 2.4, the maximum coverage of a rainbow table is approximately 86%. Stepped rainbow tables, however, have a maximum coverage of 100%, obtained when a step is placed in every column. Of course, placing a step in every column is not worth the cost in memory. Instead, a few, well-positioned steps are used. Nevertheless, the coverage obtained with a stepped rainbow table is better than those obtained with a rainbow table. This allows using one or two fewer tables to obtain the same coverage for the same memory. This decreases simultaneously the attack time, and the precomputation time (all other characteristics being equal).

6.5.2 Attack Time. Even for the same number of tables used, there are always better configurations of stepped rainbow tables with the same memory and coverage as rainbow tables, that are faster in both attack and precomputation. The gain in precomputation is obtained from the smaller number of chains that have to be computed to obtain the same coverage.

When using the same number of tables, the attack time is smaller for a less obvious reason. As mentioned in Sect. 4, when a false is detected in a step, the attack chain is not rebuilt until the last column of the table. This allows to decrease the average cost of false alarms and thus, the attack time.

6.5.3 Memory. Intuitively, stepped rainbow tables are more expensive in memory than rainbow tables. However, in this paper, rainbow tables and stepped rainbow tables are compared for the same memory and the same coverage.

The memory used to store end points is larger for stepped rainbow tables than for rainbow tables, but as fewer points are considered at the beginning of the generation, less memory is required to store start points. The storage of start points allows to keep the total storage cost of stepped rainbow tables close to those of classic tables and thus obtain more efficient tables in precomputation and attack for the same memory and coverage.

6.5.4 Memory Accesses. Depending on the memory used for the trade-off, the number of memory accesses may also be quite relevant to determine the real cost of an attack.

The number of memory accesses needed for an attack with stepped rainbow tables and rainbow tables are very close. In our experiments, stepped rainbow tables in their optimal configurations typically require fewer memory accesses. This is particularly the case when stepped rainbow tables have fewer tables than classic rainbow tables, which is often the case, as discussed in 6.5.1.

6.6 Future Work

Many improvements on the seminal rainbow tables introduced by Oechslin [31] have been published. Although some improvements and variants concern the precomputation phase, the vast majority of them focus on the attack phase.

Concerning the precomputation phase, the fuzzy rainbow variant [25] of TMTOs can have, for a same attack time, a slightly faster precomputation time than clean rainbow tables, but only for low success probabilities, which makes this variant uninteresting in many practical cases. The filtration method introduced in [10], designed for clean rainbow tables significantly reduces the precomputation time of rainbow tables and can be applied on stepped rainbow tables as well. The stepped rainbow tables and the rainbow tables configurations used in Sect. 5, have been generated using this method.

Concerning the attack phase, the major improvements on rainbow tables are checkpoints [12], truncation and fingerprints [5, 28], and the heterogeneous tables [7], which is arguably the most efficient variant suggested so far. For the sake of clarity, stepped rainbow tables are introduced in the general case, without any improvement on the attack phase. Combining the concept of heterogeneous tables with stepped rainbow tables seems very promising since this could drastically reduce both the precomputation and attack phases. We leave the evaluation of combining these two methods for future work.

7 CONCLUSION

This paper introduces stepped rainbow tables, which outperform classic rainbow tables. The key idea of stepped rainbow tables consists in recycling chains that merged during the precomputation phase instead of discarding them. These recycled chains are shorter than classic ones. This leads to the concept of *stepped* rainbow tables, a name inspired by their staircase-like appearance in a graphical representation. The attack phase is modified accordingly to take advantage of the new construction. Stepped rainbow tables allow configurations that are not reachable by classic rainbow tables. The impact is twofold.

Firstly, stepped rainbow tables always perform better than classic rainbow tables. Either precomputation is reduced without increasing the attack time, or the attack time is reduced without increasing precomputation. Both of those characteristics are reduced in certain scenarios. The gain depends on the problem parameters and other characteristics. In the practical examples we explored, stepped rainbow tables divide by 2.56 the precomputation time for the same attack time, same coverage and same memory used, or reduce the

attack time by up to 17% for the same precomputation time, same coverage and memory.

Secondly, stepped rainbow tables are able to slightly increase a parameter to significantly reduce another one. For instance for same coverage and same memory used, stepped rainbow tables can reduce the precomputation time by 38% while increasing by 5% the attack phase.

The consequence of the reduction in precomputation time in particular is not to be understated. The precomputation phase is the bottleneck of rainbow tables for large spaces today. And a reduction in the precomputation cost directly translates into a (admittedly modest, but significant) increase in the reachable attack space.

ACKNOWLEDGMENTS

A significant portion of this work was done while author Xavier Carpent worked at COSIC/KULeuven (Leuven, Belgium).

REFERENCES

[1] Last consulted January 2023. Cryptohaze, GPU Rainbow Cracker. https://www.cryptohaze.com/.

[2] Last consulted January 2023. L0phtCrack, L0phtCrack 6. http://www.l0phtcrack.com/.

[3] Last consulted January 2023. Objectif Sécurité, Ophcrack. http://ophcrack.sourceforge.net/.

[4] Last consulted January 2023. RainbowCrack Project, RainbowCrack and RainbowCrack for GPU. http://project-rainbowcrack.com/.

[5] Gildas Avoine, Adrien Bourgeois, and Xavier Carpent. 2015. Analysis of Rainbow Tables with Fingerprints. In *Information Security and Privacy - 20th Australasian Conference, ACISP 2015, Brisbane, QLD, Australia, June 29 - July 1, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9144)*, Ernest Foo and Douglas Stebila (Eds.). Springer, 356–374. https://doi.org/10.1007/978-3-319-19962-7_21

[6] Gildas Avoine and Xavier Carpent. 2013. Optimal Storage for Rainbow Tables. In *Information Security and Cryptology - ICISC 2013 - 16th International Conference, Seoul, Korea, November 27-29, 2013, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 8565)*, Hyang-Sook Lee and Dong-Guk Han (Eds.). Springer, 144–157.

[7] Gildas Avoine and Xavier Carpent. 2017. Heterogeneous Rainbow Table Widths Provide Faster Cryptanalyses. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2017, Abu Dhabi, United Arab Emirates, April 2-6, 2017*, Ramesh Karri, Ozgur Sinanoglu, Ahmad-Reza Sadeghi, and Xun Yi (Eds.). ACM, 815–822.

[8] Gildas Avoine, Xavier Carpent, Barbara Kordy, and Florent Tardif. 2017. How to Handle Rainbow Tables with External Memory. In *Information Security and Privacy - 22nd Australasian Conference, ACISP 2017, Auckland, New Zealand, July 3-5, 2017, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10342)*, Josef Pieprzyk and Suriadi Suriadi (Eds.). Springer, 306–323.

[9] Gildas Avoine, Xavier Carpent, and Cédric Lauradoux. 2015. Interleaving Cryptanalytic Time-Memory Trade-Offs on Non-uniform Distributions. In *Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9326)*, Günther Pernul, Peter Y. A. Ryan, and Edgar R. Weippl (Eds.). Springer, 165–184.

[10] Gildas Avoine, Xavier Carpent, and Diane Leblanc-Albarel. 2021. Precomputation for Rainbow Tables has Never Been so Fast. In *Computer Security – ESORICS 2021*, Elisa Bertino, Haya Shulman, and Michael Waidner (Eds.). Springer International Publishing, Cham, 215–234.

[11] Gildas Avoine, Xavier Carpent, and Diane Leblanc-Albarel. 2022. Rainbow Tables: How Far Can CPU Go? *Comput. J.* (10 2022). https://doi.org/10.1093/comjnl/bxac147 arXiv:https://academic.oup.com/comjnl/advance-article-pdf/doi/10.1093/comjnl/bxac147/46671690/bxac147.pdf bxac147.

[12] Gildas Avoine, Pascal Junod, and Philippe Oechslin. 2005. Time-Memory Trade-Offs: False Alarm Detection Using Checkpoints. In *Progress in Cryptology - INDOCRYPT 2005 (Lecture Notes in Computer Science, Vol. 3797)*, Subhamoy Maitra, C. E. Veni Madhavan, and Ramarathnam Venkatesan (Eds.). Springer, 183–196. https://doi.org/10.1007/11596219_15

[13] Gildas Avoine, Pascal Junod, and Philippe Oechslin. 2008. Characterization and Improvement of Time-Memory Trade-Off Based on Perfect Tables. *ACM Trans. Inf. Syst. Secur.* 11, 4 (2008), 17:1–17:22.

[14] S.H. Babbage. 1995. Improved "exhaustive search" attacks on stream ciphers. In *European Convention on Security and Detection, 1995.* 161–166. https://doi.org/10.1049/cp:19950490

[15] Elad Barkan, Eli Biham, and Adi Shamir. 2006. Rigorous Bounds on Cryptanalytic Time/Memory Tradeoffs. In *Advances in Cryptology - CRYPTO 2006, 26th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4117)*, Cynthia Dwork (Ed.). Springer, 1–21. https://doi.org/10.1007/11818175_1

[16] Eli Biham and Orr Dunkelman. 2000. Cryptanalysis of the A5/1 GSM Stream Cipher. In *Progress in Cryptology - INDOCRYPT 2000, First International Conference in Cryptology in India, Calcutta, India, December 10-13, 2000, Proceedings (Lecture Notes in Computer Science, Vol. 1977)*, Bimal K. Roy and Eiji Okamoto (Eds.). Springer, 43–51. https://doi.org/10.1007/3-540-44495-5_5

[17] Alex Biryukov, Sourav Mukhopadhyay, and Palash Sarkar. 2005. Improved Time-Memory Trade-Offs with Multiple Data. In *Selected Areas in Cryptography, 12th International Workshop, SAC 2005, Kingston, ON, Canada, August 11-12, 2005, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 3897)*, Bart Preneel and Stafford E. Tavares (Eds.). Springer, 110–127.

[18] Alex Biryukov and Adi Shamir. 2000. Cryptanalytic Time/Memory/Data Tradeoffs for Stream Ciphers. In *Advances in Cryptology - ASIACRYPT 2000, 6th International Conference on the Theory and Application of Cryptology and Information Security, Kyoto, Japan, December 3-7, 2000, Proceedings (Lecture Notes in Computer Science, Vol. 1976)*, Tatsuaki Okamoto (Ed.). Springer, 1–13. https://doi.org/10.1007/3-540-44448-3_1

[19] Alex Biryukov, Adi Shamir, and David Wagner. 2000. Real Time Cryptanalysis of A5/1 on a PC. In *International Workshop on Fast Software Encryption*. Springer, 1–18.

[20] Jovan Dj Golić. 1997. Cryptanalysis of alleged A5 stream cipher. In *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 239–255.

[21] Martin E. Hellman. 1980. A cryptanalytic time-memory trade-off. *IEEE Trans. Inf. Theory* 26, 4 (1980), 401–406.

[22] Yaacov Zvi Hoch. 2009. *Security analysis of generic iterated hash functions*. Ph. D. Dissertation.

[23] Jin Hong, Kyung Chul Jeong, Eun Young Kwon, In-Sok Lee, and Daegun Ma. 2008. Variants of the Distinguished Point Method for Cryptanalytic Time Memory Trade-Offs. In *Information Security Practice and Experience, 4th International Conference, ISPEC 2008, Sydney, Australia, April 21-23, 2008, Proceedings (Lecture Notes in Computer Science, Vol. 4991)*, Liqun Chen, Yi Mu, and Willy Susilo (Eds.). Springer, 131–145.

[24] Jin Hong and Sunghwan Moon. 2013. A Comparison of Cryptanalytic Tradeoff Algorithms. *J. Cryptol.* 26, 4 (2013), 559–637.

[25] Byoung-Il Kim and Jin Hong. 2013. Analysis of the Non-perfect Table Fuzzy Rainbow Tradeoff. In *Information Security and Privacy - 18th Australasian Conference, ACISP 2013, Brisbane, Australia, July 1-3, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7959)*, Colin Boyd and Leonie Simpson (Eds.). Springer, 347–362. https://doi.org/10.1007/978-3-642-39059-3_24

[26] Jung Woo Kim, Jin Hong, and Kunsoo Park. 2013. Analysis of the Rainbow Tradeoff Algorithm Used in Practice. *IACR Cryptol. ePrint Arch.* (2013), 591.

[27] Jung Woo Kim, Jungjoo Seo, Jin Hong, Kunsoo Park, and Sung-Ryul Kim. 2015. High-speed parallel implementations of the rainbow method based on perfect tables in a heterogeneous system. *Softw. Pract. Exp.* 45, 6 (2015), 837–855. https://doi.org/10.1002/spe.2257

[28] Ga-Won Lee and Jin Hong. 2016. Comparison of perfect table cryptanalytic tradeoff algorithms. *Des. Codes Cryptogr.* 80, 3 (2016), 473–523.

[29] Jiqiang Lu, Zhen Li, and Matt Henricksen. 2015. Time-Memory Trade-Off Attack on the GSM A5/1 Stream Cipher Using Commodity GPGPU. In *Applied Cryptography and Network Security - 13th International Conference, ACNS 2015, New York, USA, June 2-5, 2015, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 9092)*, Tal Malkin, Vladimir Kolesnikov, Allison Bishop Lewko, and Michalis Polychronakis (Eds.). Springer, 350–369.

[30] Nele Mentens, Lejla Batina, Bart Preneel, and Ingrid Verbauwhede. 2006. Time-Memory Trade-Off Attack on FPGA Platforms: UNIX Password Cracking. In *Reconfigurable Computing: Architectures and Applications, Second International Workshop, ARC 2006, Delft, The Netherlands, March 1-3, 2006, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 3985)*, Koen Bertels, João M. P. Cardoso, and Stamatis Vassiliadis (Eds.). Springer, 323–334.

[31] Philippe Oechslin. 2003. Making a Faster Cryptanalytic Time-Memory Trade-Off. In *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2729)*, Dan Boneh (Ed.). Springer, 617–630.

[32] François-Xavier Standaert, Gaël Rouvroy, Jean-Jacques Quisquater, and Jean-Didier Legat. 2002. A Time-Memory Tradeoff Using Distinguished Points: New Analysis & FPGA Results. In *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers (Lecture Notes in Computer Science, Vol. 2523)*, Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar (Eds.). Springer, 593–609.

[33] Mathy Vanhoef. 2022. A Time-Memory Trade-Off Attack on WPA3's SAE-PK. In *APKC '22: Proceedings of the 9th ACM on ASIA Public-Key Cryptography Workshop, APKC@AsiaCCS 2022, Nagasaki, Japan, 30 May 2022*, Jason Paul Cruz and Naoto Yanai (Eds.). ACM, 27–37. https://doi.org/10.1145/3494105.3526235

A PROOF OF THEOREM 5

Proof. To obtain the total cost of the search, the probability of the three events (true alarm, false alarm, no alarm) has to be multiplied by their cost and summed together.

Case 1: $s_\tau < c \leqslant t$.

(a) The probability of a true alarm is obtained from Prop. 2 and is $\frac{m_t}{N}$, its cost is obtained from Prop. 6 and is $\rho_{\tau+1,k(c)} s_{\tau+1} = \rho_{\tau+1,\tau+1} s_{\tau+1} = t$.

(b) The probability of false alarm is given by Prop. 3 and is: $1 - \frac{m_t}{N} - p_{\text{noalarm}}(c)$. The cost is the same as for true alarms.

(c) The probability of no alarm is given by Prop. 5 and its cost is $t - c$ (Sect. 4.4.5).

In total, we thus have:

$$\begin{aligned} C_c &= t\frac{m_t}{N} + t\left(1 - \frac{m_t}{N} - p_{\text{noalarm}}(c)\right) + (t-c)p_{\text{noalarm}}(c) \\ &= t - cp_{\text{noalarm}}(c). \end{aligned}$$

Case 2: $c \leqslant s_\tau$.

(a) The probability of a true alarm is $\frac{m_{s_{k(c)}}}{N}$, and its cost is $\sum_{i=k(c)}^{\tau+1} \rho_{i,k(c)} s_i$.

(b) The probability of a false alarm due to a merge between c and $s_{k(c)}$ is given by Prop. 3 and is $1 - \frac{m_{s_{k(c)}}}{N} - p_{\text{nomrg}}(c, k(c))$. Its cost is the same as for a true alarm.

(b') The probability of a false alarm due to a merge between s_j and s_{j+1} is $p_{\text{fa-post}}(c, s_j, s_{j+1})$ with $c \leqslant s_j < s_{j+1}$ and is given by Prop. 4. Therefore the probability of false alarm due to a merge between $s_{k(c)}$ and s_τ is $\sum_{j=k(c)}^{\tau} p_{\text{fa-post}}(c, s_j, s_{j+1})$. Its cost, given by Prop. 6, depends of step s_j in which the false alarm is detected and is $\sum_{i=j+1}^{\tau+1} \rho_{i,j+1} s_i$.

(c) The probability of no alarm is given by Prop. 5 and its cost is $t - c$ (Sect. 4.4.5).

In total, we thus have:

$$\begin{aligned} C_c = \frac{m_{s_{k(c)}}}{N} &\sum_{i=k(c)}^{\tau+1} \rho_{i,k(c)} s_i \\ &+ \left(1 - \frac{m_{s_{k(c)}}}{N} - p_{\text{nomrg}}(c, s_{k(c)})\right) \sum_{i=k(c)}^{\tau+1} \rho_{i,k(c)} s_i \\ &+ \sum_{j=k(c)}^{\tau} p_{\text{fa-post}}(c, s_j, s_{j+1}) \sum_{i=j+1}^{\tau+1} \rho_{i,j+1} s_i \\ &+ (t-c)\, p_{\text{noalarm}}(c). \end{aligned}$$

The conclusion follows from the $\frac{m_{s_{k(c)}}}{N} \sum_{i=k(c)}^{\tau+1} \rho_{i,k(c)} s_i$ terms canceling out. □

B ATTACK PHASE WITH τ STEPS

Algorithm 2: Attack Phase with τ-Steps stepped rainbow table

input : $y \in B$ s.t. $y = h(x^*) \in A$
output: x^*, if it belongs to the stepped rainbow matrix

$stepsList \leftarrow [s_1, s_2, ...s_\tau, t]$
$EP \leftarrow [[EP_1 ... EP_{m_{s_1} - m_{s_2}}],$
$[EP_{m_{s_1} - m_{s_2}+1} ... EP_{m_{s_1} - m_{s_3}}] ...$
$[EP_{m_{s_1} - m_t+1} ... EP_{m_{s_1}}]]$
$SP \leftarrow [SP_1 ... SP_{m_{s_1} - m_{s_2}}],$
$[SP_{m_{s_1} - m_{s_2}+1} ... SP_{m_{s_1} - m_{s_3}}] ...$
$[SP_{m_{s_1} - m_t+1} ... SP_{m_{s_1}}]]$
$c \leftarrow [s_1, s_2, ..s_\tau, t]$
$v \leftarrow [0 : len(c) - 1]$
$success \leftarrow False$
while $c \neq [0, 0, .., 0]$ *and* $success = False$ **do**
  $Alarm \leftarrow False$
  for $d = 0$ *to* $len(c) - 1$ **do**
    $v[d] \leftarrow \mu(c[d])$ *#See Def. 3*
  end
  $j \leftarrow c[v.index(max(v))]$
  $i \leftarrow j$
  $x_i \leftarrow R_i(y)$
  while $i \leqslant t$ *and* $Alarm = False$ **do**
    if i *in* $stepsList$ **then**
      $index_i \leftarrow stepsList.index(i)$
      if x_i *in* $EP[index_i]$ **then**
        $Alarm \leftarrow True$
        $x \leftarrow SP[EP[index_i].index(x_i)]$
        for $g = 1$ *to* $g = j - 1$ **do**
          $x \leftarrow f_g(x)$
        end
        if $h(x) = y$ **then**
          $success \leftarrow True$
        end
      end
    end
    if $Alarm = False$ *and* $success = False$ **then**
      $x_i \leftarrow f_i(x_i)$
      $i \leftarrow i + 1$
    end
  end
  if $success = False$ **then**
    $c[c.index(j)] \leftarrow c[c.index(j)] - 1$
  else
    return $(success, x)$
  end
end
return $(success, 0)$

C CONFIGURATIONS

Function *adjust* used in Algo. 3, takes a number of column t, and a list of *steps*, with a relative step position between 30% and 100% of t for each step. It return the positions of steps according to t.

Algorithm 3: Algorithm used to find valid configurations

input : The success probability *TargetProba*
The targeted memory
TargetMemory
output: A list *validConfig* of configurations that reaches the targeted success probability and memory

Procedure:
$t \leftarrow 1$
$\alpha \leftarrow 0.001$
for $ell = 1$ *to* $\ell = 6$ **do**
  for $nbSteps = 1$ *to* $nbSteps = 5$ **do**
    for *pos in all possible steps combinations* **do**
      $config \leftarrow [t, alpha, \ell, steps]$
      $end \leftarrow valid\_criteria(config)$
      while $end \neq 1$ **do**
        while $proba(config) < TargetProba$ **do**
          $\alpha \leftarrow \alpha + 0.001$
          $config \leftarrow [t, alpha, \ell, steps]$
        end
        while $memory(config) < 0.999 TargetMemory$ **do**
          $t \leftarrow t - 1$
          $steps \leftarrow adjust(t, pos)$
          $config \leftarrow [t, alpha, \ell, steps]$
        end
        while $memory(config) > 1.001 TargetMemory$ **do**
          $t \leftarrow t + 1$
          $steps \leftarrow adjust(t, pos)$
          $config \leftarrow [t, alpha, \ell, steps]$
        end
        $end \leftarrow valid\_criteria(config)$
      end
      $validConfig.append(config)$
    end
  end
end
return *validConfig*