Rainbow Tables: How Far Can CPU Go?

Gildas Avoine, Xavier Carpent, Diane Leblanc-Albarel

Rainbow Tables: How Far Can CPU Go?

Gildas Avoine[1], Xavier Carpent[2] and Diane Leblanc-Albarel[1]

[1]*CNRS, INSA Rennes, IRISA, FRANCE*
[2]*University of Nottingham, School of Computer Science, UK*
Email: diane.leblanc-albarel@irisa.fr

Rainbow tables are techniques commonly used in computer security to invert one-way functions, for instance to crack passwords, when the domain of definition is reasonably-sized. This article explores the limit on the problem size that can be treated by rainbow tables when the precomputation and the attack phases are both CPU-driven. We conclude that the bottleneck is no longer the memory as it may have been and the precomputation phase seems to have been underestimated so far. In contrast to the usual articles on rainbow tables, we offer a comparison of what can be done on different environments depending on the needs and available computing power of the users.

Keywords: Cryptography; Time-Memory Trade-Off (TMTO); Rainbow Table

1. INTRODUCTION

Context A rainbow table is a data structure used to invert one-way functions – for example hash or encryption functions – when the domain of definition is reasonably-sized. It is the most used variant of cryptanalytic Time-Memory Trade-Off (TMTO). A TMTO has characteristics between that of brute-force search (which has high computation cost) and dictionary attack (which has high storage cost). Rainbow tables are particularly useful (1) when the inversion (or *attack*) is repeated many times, (2) when the precomputation is delegated (e.g., a powerful entity precomputes, but a simple laptop carries out the attack), or (3) when there is a small pre-determined window of opportunity for the attack but a possibly long time to prepare for it (*lunchtime attack*).

The use of rainbow tables is divided in two phases: precomputation phase (performed once) and attack phase (performed for each inversion). Given a problem of size N (N is the number of possible solutions) and a memory of M_T, the complexity of the precomputation phase is linear in N (with a factor varying, depending on parameters and optimizations – see discussion in [1]) and the complexity of the attack phase is $O(N^2/M_T^2)$. This means that a memory anywhere from $O(\sqrt{N})$ (below which the attack becomes *slower* than brute force) to $O(N)$ (above which the memory becomes larger than in a dictionary attack) can be used to accelerate the search with respect to brute force. Various papers discuss algorithmic and implementation improvements to both phases: variants [2, 3], attack phase optimizations [4, 5, 6], storage optimizations [7], and precomputation improvements [1].

Motivation This article aims to characterize the efficiency of rainbow tables from a practical point of view, and determine which largest N can be realistically addressed by rainbow tables, with these optimizations in mind. This is explored with a CPU-driven computation model, with reasonable assumptions about the computing environment and parameters. We further aim to gain some insight from this work into the current practical impact of rainbow tables in computer security, and the future research for rainbow tables and associated concepts.

Contribution We define three potential types of environments for the precomputation phase. Each environment corresponds to typical needs and budget with today's technology. For each environment, we evaluate the largest N that can be undertaken and we identify the technological bottleneck that prevents going further. Experiments are provided to illustrate certain theoretical evaluations.

Two environments are super-computers or rather large computers, with at least 100 cores. The one that we call `supercomputer` is a computer typically belonging to the top-100 worldwide list[3]. The other one, which we call `computer` corresponds to a computer available for medium sized companies or academic research teams. The third environment, called the `cloud` environment corresponds to what could be expected on rented machines available in the

[3]`https://top500.org/`

cloud. The latter environment could be economically interesting if precomputations are performed only once or occasionally. Using these three environments, we conclude what the three main entity types can achieve in TMTOs today. Section 6 provides approaches to improve TMTOs in the future. Knowing the space size that can be easily attacked by TMTOs today also allows the whole community to take measures to protect themselves.

Organization Section 2 summarizes basics on rainbow tables, especially the way the two phases are carried out. Section 3 briefly presents other TMTOs variants and improvements made. Section 4 introduces the environments and scenarios considered in this article. The performed evaluations and experiments are described in Section 5. Discussions about the technological limits of TMTOs then conclude the article in Section 6.

2. BACKGROUND

2.1. Overview

The purpose of TMTOs is to retrieve a preimage of a given value obtained through a (one-way) function. Usually, this one-way function is denoted h with $h : A \to B$. The searched preimage is denoted by x with $x \in A$, and the given value is denoted by y with $y \in B$. The aim is therefore to retrieve x from y with $y = h(x)$, using precomputed tables.

Various TMTOs algorithms exist. One of the most efficient and commonly used variant is the *rainbow tables* variant, which is consequently the variant addressed in this paper. Rainbow tables without any improvement, called classic rainbow tables, are considered in this section. Other variants and improvements are briefly described in Section 3, and more extensive descriptions can be found in, e.g., [1, 8, 9, 10].

2.2. Precomputation Phase

2.2.1. Matrix Construction

To construct a rainbow table, a rainbow matrix of m rows and $t+1$ columns is firstly generated. Each matrix element is denoted $x_{i,j}$, with $x_{i,j} \in A$, $0 \leqslant i \leqslant t$ and $0 < j \leqslant m$. Each element $x_{i+1,j}$ of a row j, is obtained from the previous element $x_{i,j}$ using a function denoted f_i and named *hash-reduction* function, where:

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

Given the hash function $h : A \to B$, and a *reduction function* family[4] $r_i : B \to A$, the hash-reduction function f_i is defined by:

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

The successive application of f_i forms a *chain*. To form a *rainbow matrix* of m *rows* and $t+1$ *columns*, m chains are computed by iterating t times the function f_i on m arbitrary chosen elements of A.

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

FIGURE 1: Rainbow matrix

Figure 1 illustrates the construction of a matrix with m rows ans $t+1$ columns. The elements in the first column of the matrix are called *start points* and elements in the last column are called *end points*. Only start points and end points are kept to form the rainbow table. All the other columns are discarded.

2.2.2. Clean Rainbow Table

During the matrix generation, chains may *merge*. A merge between 2 chains occurs when all their elements are equal after a given column, i.e., when $x_{i,j} \neq x_{k,j}$ and $x_{i,j+1} = x_{k,j+1}$, for $0 \leqslant j < t$. Chain merging is possible because $|B| > |A|$.

To make the attack phase as efficient as possible, the table is trimmed to only keep rows that have not merged, i.e., chains with unique end points – this is called a *clean* or *perfect* table.

As the attack phase should be as fast as possible, clean rainbow tables are usually used. In what follows, tables will be considered clean, m_0 will denote the number of start points used to generate the matrix, and m_t will denote the number of points remaining in column t after the cleaning process.

2.2.3. Non-Maximal Table

The number of start points, m_0, is chosen to be as high as practically possible for a given t. When $m_0 = N$ the table is said to be *maximal*. In that case $m_t^{\max}$ points remain in the last column after cleaning the table. $m_t^{\max}$ is defined in Theorem 2.1 from [4].

Theorem 2.1. *Given t and a sufficiently large N, the expected maximum number of chains per clean rainbow table is:*

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

The higher the desired number of points remaining in column t, the higher the proportion of chains that need to be thrown out during the cleaning due to duplicate

[4]The purpose of a reduction function is to map a hash to an arbitrary element of the input space A. A notable example is $r_i(y) \mapsto (y + i) \bmod N$ for $A = \{0, 1, \ldots, N-1\}$.

end points. Therefore, besides being very costly, taking $m_0 = N$ is not worth the cost compared to the number of remaining chains at the end of the generation. Thus, in practice, *non-maximal* tables are generated with a number of start points much lower than N.

Given a number of start points $m_0 < N$, the *maximality* factor α with $0 < \alpha < 1$, is defined such that:

$$m_t = \alpha\, m_t^{\max}$$

It is then possible to define a ratio r of chains to be generated in order to obtain a targeted $m_t = \alpha m_t^{\max}$ number of chains in the final clean table. Proposition 2.1 introduced in [1] defines this ratio r.

PROPOSITION 2.1. *With a target of $m_t = \alpha\, m_t^{\max}$ unique end points, $m_0 = r m_t^{\max}$ chains need to be generated to construct a rainbow table, with:*

$$r \approx \frac{\alpha}{1-\alpha}$$

A table created with a typical ratio of $r = 20$ chains computed per chain kept for instance, results in m_t being at about 95% of its maximal value.

2.2.4. Success Probability

A single clean table covers much of A, but not all of it (very close to 86.47% for maximal tables). Therefore, several independently-computed tables are used in the attack phase, reaching a success probability arbitrarily close to 1. For a single table with m_t end points, the success probability $P(t)$ is given by Equation (1), adapted from [8].

$$P(t) = 1 - (1 - \frac{m_t}{N})^t. \tag{1}$$

Using ℓ tables the success probability $P(t, \ell)$ is given by Equation (2) from [8].

$$P(t, \ell) = 1 - (1 - P_c(t))^\ell. \tag{2}$$

2.3. Attack phase

Once the precomputation phase is completed, the attack phase can be performed. The latter phase uses the precomputed rainbow tables to find a preimage of a given $y \in B$. The following process is iterated until an answer is found or after t iterations (in which case the process *fails*): at iteration i, a chain of length i is computed: $f_t(\dots f_{t-i+1}(r_{t-i}(y))\dots)$. Its end point is then compared to the end points in the table. If a match is found, the corresponding chain of the matrix is rebuilt from its start point, up to the column $t-i-1$: $x_{t-i-1,j} = f_{t-i-1}(\dots f_1(x_{1,j})\dots)$. If $h(x_{t-i-1}) = y$ then the attack is completed and successful. If the two hashes are not equal, or if no match was found, the process proceeds to the next iteration. If the attack reaches t iterations, it fails.

Proposition 2.2 adapted from [4], defines the number of operations needed to perform a search in the column c, i.e., after $t - c$ iterations. The number of unique points remaining in column i is denoted m_i and can be easily computed as demonstrated in [1].

PROPOSITION 2.2. *After $t - c$ iterations, 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 [4]. The result can be extrapolated to non-maximal rainbow tables. □

The average attack time using rainbow tables can be deduced from Proposition 2.2 and is given by Theorem 2.2

THEOREM 2.2. *The average number of hash operations T required to perform an attack using rainbow tables, given a search space of size N and ℓ tables is:*

$$T = \ell\sum_{i=1}^{t}\left(\frac{m_t}{N}\left(1 - \frac{m_t}{N}\right)^{\ell(i-1)}\sum_{j=1}^{c} C_{t-j+1}\right) + e^{-2\ell}\ell\sum_{i=1}^{t} C_i.$$

Proof. As Proposition 2.2, this theorem is derived from formulas given in [4]. □

During the attack phase, tables can be loaded in RAM or on disks (preferably SSDs). In the latter case, several approaches can be taken (see [11, 12]).

In [11], the authors show that the attack time is not significantly impacted by the use of secondary memory (SSD, HDD etc.) rather than RAM. Furthermore, as discussed in Section 5 and 6, performing the attack phase on secondary memory rather than on RAM would not change the conclusions of this paper.

However, an attack on secondary memory is more complex. Thus, for the sake of clarity, in this paper, we consider attack phase on RAM.

The attack phase benefited from various algorithmic improvements that significantly improved its performances. All told, a fully-fledged and well-parameterized implementation can be from 5 to 10 times faster[5] than a naive implementation. While important in practice, they do not change the order of magnitude of the attack phase cost. A brief summary is nonetheless presented in Section 3, but for clarity, rainbow tables without improvement on the attack will be used in this paper.

3. RELATED WORK

This section provides an overview of TMTO variants and improvements published so far, along with references that detail each of them.

[5]The exact speed-up depends on many factors and parameters. This window is a conservative estimate, based on typical scenarios and configurations.

3.1. TMTOs Variants

Hellman Table. The earliest TMTO was introduced by Martin Hellman in 1980 [13]. This work differs from the rainbow variant on the choice of the reduction function: only one reduction function is used in Hellman's case instead of a reduction function per column in the rainbow variant. At each iteration performed during the attack phase, only one additional application of the reduction function is needed, which speeds up considerably this step. However, the coverage of Hellman's tables is very poor due to the large number of collisions it generates, which implies using a very large number of tables (typically t tables) to obtain a suitable coverage. The greater the number of tables used, the longer the attack time. In addition, cleaning Hellman's table is very difficult and not necessarily worth it. In the end, Hellman's tables are less efficient than rainbow tables and, therefore, are non longer used.

Distinguished Point (DP) Tables. DP variant is based on Hellman's tables, but instead of having chains of length t, chains are computed until reaching a so-called *distinguished point* (typically, points that have at least d bits at 0). DP tables have thus chains with variable lengths. End points are all distinguished points which make the table easy to clean. As chains of DP tables do not have predictable sizes, it alters both the precomputation phase and the attack phase. In the end, it has been established in [3, 10], that DP tables are less efficient than rainbow tables.

Fuzzy Rainbow Tables. This variant is a trade-off between rainbow tables and DP tables. Fuzzy rainbow tables are extensively studied in [2]. In this variant, each chain is built by concatenating DP chains. As shown in [9], except for very low success probabilities and, therefore, uninteresting cases in practice, fuzzy rainbow tables are less efficient in precomputation and attack than rainbow tables.

3.2. Precomputation Phase Improvement

Until recently, most of the improvements on TMTOs focused on the memory or time needed for the attack. In 2021, [1] shows that the precomputation time can also be improved to cover larger spaces. The *filtration* method have thus been introduced in [1], the method consists in cleaning the matrix as the generation progresses rather than at the end and allows dividing the precomputation time by 6 without increasing the attack time.

3.3. Attack Phase Improvements

Checkpoints. Checkpoints have been introduced in [4]. The concept consists in storing additional information – so call checkpoints – on each chain, such that this information allows reducing the attack phase time. The checkpoints, are stored along with the start points and end points. During the attack phase, when the attack chain matches an end point of the table, the checkpoints of the built attack chain are compared to the checkpoints of the matching chain. It at least a checkpoint differs, then the match is a false alarm and it is consequently useless to rebuild the chain of the table from its starting point. With optimal parameters, rainbow tables with checkpoints are faster than the classical case but require slightly more memory.

Fingerprints. This improvement has been introduced in [5]. Its consists of applying in an efficient way the checkpoints improvement with a memory improvement called *truncated end points.* the principle is to store end points that have been truncated to reduce the memory needs for the attacks phase. Truncated end points alone increase the attack time significantly but used combined with checkpoints in the fingerprints method, they allow a speedup of about 2 compared to the classical rainbow table.

Heterogeneous Tables. This variant has been introduced in [6]. It consists in using rainbow tables with different lengths. This implies that searching in the shorter tables takes less time than searching in classic tables but searching in the longer ones takes more time. This variant does not require more memory than the classical case. By wisely performing the attack phase (favoring searches in the shortest tables), this variant reduces the average attack time by up to 40% without increasing the memory. The counterpart is that the worst case is slightly longer than with classic tables.

4. ENVIRONMENTS AND SCENARIOS CONSIDERED

4.1. Context

In practice, the entities that perform TMTOs have different needs, purposes, and resources (e.g., available memory, price, time available, etc.). Resources are, in addition, not the same for the attack and precomputation phases.

It is therefore necessary to define the context in which each phase is performed. A company, for instance, does not have the same resources than a nation state. The memory available for the precomputation must be defined as well as the one for the attack. Other variables such as the available time should also be specified in advance.

To define the context in which a TMTO is performed, we will use the notion of *environments* and *scenarios.* An *environment* corresponds to the material resources available (RAM, number of cores, CPU performances etc.). A *scenario* corresponds to the non-material resources available (Time or money).

In this paper, we consider different environments, each of them described in 4.2 (precomputation) and

in 4.3 (attack phase). These environments aim to represent different entities. For each environment, we have considered different scenarios depending on available time or money.

4.2. Precomputation phase

Environment	#Cores	Hashes/Sec/Core	RAM (TB)
`supercomputer`	86 344	25 000 000	16 182
`computer`	127	11 000 000	1

TABLE 1: Precomputing `supercomputer` and `computer` environments

Three different types of precomputation environments are considered in this paper: (1) `supercomputer`, which is representative of a supercomputer among the top 100 worldwide; (2) `computer`, which corresponds to a 128-core computer; and (3) `cloud`, which consists of rented computing units from on the main cloud platforms, e.g., AWS, Azure, or GCP. For each environment, the number of cores, the number of hashes/second/core, and the available memory (RAM) are provided for `supercomputer` and `computer` in Table 1 and in Table 2 for `cloud`.

Typically, `supercomputer` represents the computing power of a governmental agency, `computer` might be a computer owned by a university or a small to medium-sized business, and `cloud` illustrates an entity that rents commercial computing units in order to precompute rainbow tables.

For each environment, three scenarios are considered. As shown in Table 1, for `supercomputer` and `computer`, the scenarios depends on the available time for the precomputation phase: 1 year, 1 month, or 1 week. For `cloud` however, the precomputation phase is bounded to 1 month, and as shown in Table 2, the environments are defined by the budget assigned to the precomputation phase according to the different scenarios: 1 000 000 USD, 100 000 USD, or 10 000 USD[6].

Scenarios	`cloud` Environments		
	#Cores	Hashes/Sec/Core	RAM (TB)
1M USD	13 055	11 000 000	256
100K USD	1 279	11 000 000	20
10K USD	127	11 000 000	2

TABLE 2: Precomputing cloud environments according to the scenarios

For each precomputing environment and scenario, $\ell = 4$ tables are used, and parameters $\alpha = 0.95$ and thus $r = 20$ are chosen to reach a success probability larger than 99.95%. The aim is to maximize N while keeping the attack phase affordable.

[6]The computing characteristics shown in Table 2 have been obtained by simulating on these costs on AWS (cluster EC2).

	Memory available for the attack (TB)			
Scenarios	`supercomputer`	`computer`	Scenarios	`cloud`
1 year	32	0.8	1M USD	16
1 month	16	0.4	100K USD	8
1 week	8	0.2	10K USD	4

TABLE 3: Memory available for the attack according to the scenarios and environments

4.3. Attack phase

For the attack phase, we consider environments with a single core[7]. This makes the analysis and description of environments a little simpler, and as discussed in Section 6, does not change the overal situation.

The number of hashes per second n_t, on the attack core is fixed for each environment $n_t = 11\,000\,000$[8]. The idea is that the attacker usually has limited resources for the attack phase and cannot benefit from a supercomputer for this phase.

The only attribute that has a bearing on the efficiency of the attack is the size of the memory available. The different values for the attack memory M_T are chosen to correspond to realistic, practical cases and influence the parameter t (given our assumption of $\ell = 4$ tables). Memory available (but not necessarily used) are presented in Table 3.

The memory available for this phase must remain relatively small. To correspond to cases encountered in practice, it must be small compared to the memory available for the precomputation phase. The memory available for the attack is therefore adapted to the scenarios considered and to the environments used for the precomputation phase.

For each environment, scenarios depend on the time or money invested in the precomputation phase. The aim is to perform the attack as quickly as possible given the tables generated during the corresponding precomputation scenarios: 1 year, 1 month, 1 week of precomputation or 1M, 100K, 10K, USD invested.

5. EVALUATION OF THE MAXIMUM PROBLEM SIZE

5.1. Methodology

The maximum problem size N that can be addressed by a CPU-based TMTO in a given time-frame or budget can be accurately evaluated from the analytical formulas provided in [1] (precomputation time) and [11, 12] (attack time).

5.1.1. Precomputation Phase

The minimum precomputation time P_{min} in seconds is given by Equation (3) derived from Theorem 2 in [1].

[7]Note that a single core is considered to provide a reference value, but the attack phase can be easily parallelized to operate on several cores.

[8]This corresponds to the number of SHA256 hashes per second measured on a AMD EPYC 7742 3.2 GHz processor.

The total precomputation time P in seconds is given by Equation (4) derived from Equations (5) to (9) in [1], with the same speed values (called v_o and v_c) used in the first environment of [1].

$$P_{\min} = \frac{2N}{n_h v_h} \ln(1+r) \tag{3}$$

$$P = Max\left(\frac{1}{n_h v_h}\sum_{i=1}^{a+1} m_{c_{i-1}}(c_i - c_{i-1}); \frac{1}{v_f n_f}\sum_{i=1}^{a+1} m_{c_{i-1}}\right) + v_o \sum_{i=1}^{a+1} m_{c_{i-1}} + \frac{v_c}{n_h}\sum_{i=1}^{a+1} m_{c_{i-1}} \tag{4}$$

The following parameters are used in these equations: a is the number of filters used during the precomputation phase; each c_i is the columns of the i-th filter, with $c_0 = 0$ and $c_a = t$; finally, the values n_h, v_h, n_f and v_f denote the number of computation nodes, the number of hashes per second per core, the number of filtration nodes and the number of filtration per second per core respectively. As in [1], a single filtration core is sufficient, thus $n_f = 1$. Values of n_h and v_h according to the environment and scenario are given in Table 1 and 2. Values for v_f are given in Table 4.

For `computer` and `cloud` environments the number of hashes per second corresponds to typical hashes value of an environment similar to `computer`. The filtration speed corresponds to filtration speed measured on an environment similar to `computer`. For `supercomputer` environment hashes and filtrations per second have been computed from typical FLOPS numbers on this kind of environment.

Environment	Hashes/Sec/Core	Filtration/Sec/Core
`supercomputer`	25 000 000	51 270 585
`computer`	11 000 000	15 949 709
`cloud`	11 000 000	15 949 709

TABLE 4: Hashes and filtrations per second per core

Several fixed parameters are defined as follows: the number of tables generated is $\ell = 4$, the factor $r = 20$ is used (so $m_0 = 20m_t^{\max}$ is considered at the beginning of precomputation). Since environments considered in this paper are very close to the first environment of [1], identical values of v_c and v_o are used for the estimations, i.e., $v_c = 0$ and $v_o = 1.37.10^{-10}$.

5.1.2. Memory Used

The memory M_T used by the $\ell = 4$ tables is given by Equation (5), the maximum RAM needed for the precomputation phase M_P is given by Equation (6) with m_{c_1} the number of unique points in the column of the first filter. The factor 3 is explained by the fact that the hash table used for filtration has a load factor $\lambda = 1.5$ and that both start points and end points are stored. As presented in [1], filters are placed to minimize P.

$$M_T = 2\ell m_t \log_2(N) \tag{5}$$

$$M_P = 3m_{c_1} \log_2(N) \tag{6}$$

The parameter t is chosen for each time frame (year, month, or week), determined from M_T, and according to the budget (1M, 100K, or 10K USD) for the precomputation phase. As presented, in Proposition 1 from [1], t impacts M_P. Therefore, t influences the memory of the two phases. If t is too high, the attack phase will be too slow. On the other hand, if t is too small, the memory needed for precomputation and especially for storing tables may be too large.

5.1.3. Attack Phase

As only one core is used for the attack phase regardless of the environment used for the precomputation, the attack time can be well estimated by dividing result of Theorem 2.2 by the number of hashes per second n_t performed by the CPU used for the attack. Therefore, the attack time T_{RAM}, is given by Equation (7)

$$T_{RAM} = \frac{T}{n_t}, \tag{7}$$

with T the number of hashes needed to perform the attack given in Theorem 2.2. As the attack should be performed by an average computer, we have chosen to use $n_t = 11\ 000\ 000$ as the number of hash per second used for the attack phase.

5.2. Results

The results of the evaluation are provided in Table 5, where 9 configurations are presented according to the environments and scenarios considered.

6. DISCUSSION

6.1. Noteworthy observations

(1) The *cost P and the theoretical lower bound $P_{\min}$ are very close* in most cases. In the cases where the gap between P_{min} and P is more significant (e.g., `supercomputer` environment), converting some computation nodes to filtering nodes would reduce the gap significantly, as it is mainly explained by a lack of filtration speed.

Therefore, unless a significant algorithmic change is introduced, improving the efficiency of the precomputation phase would have no noticeable impact on the domain size that can be realistically attacked.

(2) For all environments considered, the largest domain sizes on which tables can be precomputed in each scenario lead to tables that are *practical* for the attack phase (i.e., reasonably small in memory, and reasonably fast to execute a search). In other words, within the context established in this paper on the environments, scenarios, technology, and algorithms,

			Precomputation phase								
			supercomputer			computer			cloud		
			1 year	1 month	1 week	1 year	1 month	1 week	1M USD	100K USD	10K USD
N	Problem size		$\mathbf{2^{61}}$	$\mathbf{2^{56.83}}$	$\mathbf{2^{54.2}}$	$\mathbf{2^{50.68}}$	$\mathbf{2^{47.05}}$	$\mathbf{2^{44.9}}$	$\mathbf{2^{53}}$	$\mathbf{2^{50.13}}$	$\mathbf{2^{47.05}}$
t	Number of columns	($\times 10^3$)	7 611	788	242.9	1 200	89	17.1	75.5	37	41.5
M_P	Mem.(TB) prcmp.	Eq.(6)	182.23	59.82	14.1	0.98	0.96	0.98	25.05	15.29	1.99
M_T	Mem.(TB) attack	Eq.(5)	32.0	16.0	8.0	0.13	0.14	0.15	11.02	2.89	0.29
P	Prcmp. time (days)	Eq.(4)	366.05	30.06	7.73	364.35	30.0	7.11	31.23	30.16	30.04
$P_{\min}$	Prcmp. low. bnd.(days)	Eq.(3)	301.13	16.73	2.7	354.34	28.62	6.45	17.21	24.01	28.4

			Attack phase								
			supercomputer			computer			cloud		
T_{RAM}	Attack time	Eq.(7)	9.62 d	2.47 h	14.12 m	5.74 h	1.9 m	4.15 s	1.35 m	19.6 s	24.73 s

The abbreviations "d", "h", "m", and "s" respectively denote "days", "hours", "minutes", and "seconds".

TABLE 5: Evaluation of the maximum problem size

the *precomputation phase is the bottleneck* to using rainbow tables on large domains.

The slight overhead due to performing the attack phase on secondary memory (HDD or SSD) would thus not change this conclusion.

(3) The *memory* M_P needed to perform the precomputation is *not a bottleneck* in the environments and scenarios analyzed in this paper.

For instance, in the case of `computer` environment, the RAM available for the precomputation phase is limited compared to the ones available in the other environments. This limited memory available for precomputation does not significantly impact the attack times of the different scenarios as they remain fairly fast.

Depending on the variants and improvements considered, it might be expected that the memory for precomputation would be limiting, although in theory, this limitation would not necessarily exist.

Therefore, the cases considered in this paper demonstrate that the memory available for precomputation is not a limiting factor. Indeed, the attack times are fast when using realistic environments and scenarios with limited memory for precomputation.

6.2. Conclusion

Our main observation from Table 5 is that the precomputation cost P is, today, the bottleneck of the TMTO. Indeed, the attack time remains reasonably low, even for the bigger space sizes considered. The memory needed for the precomputation and the attack is affordable (for the corresponding entity that performs it). Therefore, it is the precomputation time P, and more precisely, the computing power of the adversary (number of computation nodes and/or number of hashes per second) dedicated to the precomputation that limits an increase in N.

Using the state-of-the-art precomputation algorithm, there is little room for improvements on the precomputation cost P. Indeed, the P is very close to the theoretical lower bound $P_{\min}$.

Going further may require forgoing the CPU technology for a more efficient one, which could currently be GPU or FPGA. While these technologies have constraints in how they are put to use, they do operate several orders of magnitude faster than typical CPUs. Several articles, e.g., [14] treat this problem and websites propose implementations or programs[9] to purchase or generate rainbow tables on GPU. Other contributions, e.g., [15, 16] focus on FPGA-based TMTOs. However, these papers address the problem with relatively small domains N (in the 2^{40} to 2^{50} range), deprecated one-way functions, or older GPU/FPGA models. Furthermore, they do not use recent improvements in TMTOs, e.g., filtration. Therefore, it is not possible to compare these results obtained on GPU/FGPA with the results we obtained in this paper. A deeper look into efficient GPU- or FPGA-based precomputation represents interesting future work that could evaluate these technologies capabilities to deal with larger TMTOs.

In summary, our work helps quantify the vulnerable spaces to TMTOs performed on CPUs. In addition, although the precomputation phase has not been as widely studied as the attack phase, it nevertheless seems to be the main bottleneck preventing larger TMTO-based attacks today.

Our conclusions are of course dependent on the assumptions made in the premise of this paper, i.e. specific to CPU-based attacks, under some reasonable computing environments and precomputation scenarios, with specific attack phase context in mind, etc. These assumptions were established with care, and a large

[9] For instance, `https://www.cryptohaze.com/` offers a GPU-based rainbow cracker, and `http://project-rainbowcrack.com/` has implementations of rainbow tables on GPU.

spectrum of conditions were considered. Nevertheless, a significant change in hardware technology, or a profound algorithmic modification (especially targeting the precomputation), or a notable deviation of the assumptions, could yet increase the domain size vulnerable to attacks.

The precomputation phase being a bottleneck in our observations may likewise be challenged by such changes. Indeed, the precomputation cost is by nature linear in N, whereas the attack phase is by nature quadratic in N. A situation where the attack phase is the bottleneck is therefore not far-fetched, depending on the development of technologies and research in this field.

ACKNOWLEDGMENTS

A significant portion of this work was done while Xavier Carpent worked at COSIC/KULeuven (BELGIUM).

REFERENCES

[1] Gildas Avoine, Xavier Carpent, and Diane Leblanc-Albarel. Precomputation for rainbow tables has never been so fast. In Elisa Bertino, Haya Shulman, and Michael Waidner, editors, *Computer Security – ESORICS 2021*, pages 215–234, Cham, 2021. Springer International Publishing.

[2] Byoung-Il Kim and Jin Hong. Analysis of the non-perfect table fuzzy rainbow tradeoff. In Colin Boyd and Leonie Simpson, editors, *Information Security and Privacy - 18th Australasian Conference, ACISP 2013, Brisbane, Australia, July 1-3, 2013. Proceedings*, volume 7959 of *Lecture Notes in Computer Science*, pages 347–362. Springer, 2013.

[3] Jin Hong, Kyung Chul Jeong, Eun Young Kwon, In-Sok Lee, and Daegun Ma. Variants of the distinguished point method for cryptanalytic time memory trade-offs. In Liqun Chen, Yi Mu, and Willy Susilo, editors, *Information Security Practice and Experience, 4th International Conference, ISPEC 2008, Sydney, Australia, April 21-23, 2008, Proceedings*, volume 4991 of *Lecture Notes in Computer Science*, pages 131–145. Springer, 2008.

[4] 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(4):17:1–17:22, 2008.

[5] Gildas Avoine, Adrien Bourgeois, and Xavier Carpent. Analysis of rainbow tables with fingerprints. In Ernest Foo and Douglas Stebila, editors, *Information Security and Privacy - 20th Australasian Conference, ACISP 2015, Brisbane, QLD, Australia, June 29 - July 1, 2015, Proceedings*, volume 9144 of *Lecture Notes in Computer Science*, pages 356–374. Springer, 2015.

[6] Gildas Avoine and Xavier Carpent. Heterogeneous rainbow table widths provide faster cryptanalyses. In Ramesh Karri, Ozgur Sinanoglu, Ahmad-Reza Sadeghi, and Xun Yi, editors, *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2017, Abu Dhabi, United Arab Emirates, April 2-6, 2017*, pages 815–822. ACM, 2017.

[7] Gildas Avoine and Xavier Carpent. Optimal storage for rainbow tables. In Hyang-Sook Lee and Dong-Guk Han, editors, *Information Security and Cryptology - ICISC 2013 - 16th International Conference, Seoul, Korea, November 27-29, 2013, Revised Selected Papers*, volume 8565 of *Lecture Notes in Computer Science*, pages 144–157. Springer, 2013.

[8] Philippe Oechslin. Making a faster cryptanalytic time-memory trade-off. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 617–630. Springer, 2003.

[9] Jin Hong and Sunghwan Moon. A comparison of cryptanalytic tradeoff algorithms. *J. Cryptol.*, 26(4):559–637, 2013.

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

[11] Gildas Avoine, Xavier Carpent, Barbara Kordy, and Florent Tardif. How to handle rainbow tables with external memory. In Josef Pieprzyk and Suriadi Suriadi, editors, *Information Security and Privacy - 22nd Australasian Conference, ACISP 2017, Auckland, New Zealand, July 3-5, 2017, Proceedings, Part I*, volume 10342 of *Lecture Notes in Computer Science*, pages 306–323. Springer, 2017.

[12] Jung Woo Kim, Jin Hong, and Kunsoo Park. Analysis of the rainbow tradeoff algorithm used in practice. *IACR Cryptol. ePrint Arch.*, page 591, 2013.

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

[14] Jiqiang Lu, Zhen Li, and Matt Henricksen. Time-memory trade-off attack on the GSM A5/1 stream cipher using commodity GPGPU. In Tal Malkin, Vladimir Kolesnikov, Allison Bishop

Lewko, and Michalis Polychronakis, editors, *Applied Cryptography and Network Security - 13th International Conference, ACNS 2015, New York, USA, June 2-5, 2015, Revised Selected Papers*, volume 9092 of *Lecture Notes in Computer Science*, pages 350–369. Springer, 2015.

[15] François-Xavier Standaert, Gaël Rouvroy, Jean-Jacques Quisquater, and Jean-Didier Legat. A time-memory tradeoff using distinguished points: New analysis & FPGA results. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, volume 2523 of *Lecture Notes in Computer Science*, pages 593–609. Springer, 2002.

[16] Nele Mentens, Lejla Batina, Bart Preneel, and Ingrid Verbauwhede. Time-memory trade-off attack on FPGA platforms: UNIX password cracking. In Koen Bertels, João M. P. Cardoso, and Stamatis Vassiliadis, editors, *Reconfigurable Computing: Architectures and Applications, Second International Workshop, ARC 2006, Delft, The Netherlands, March 1-3, 2006, Revised Selected Papers*, volume 3985 of *Lecture Notes in Computer Science*, pages 323–334. Springer, 2006.