Precomputation for Rainbow Tables has Never Been so Fast

Gildas Avoine, Xavier Carpent, Diane Leblanc-Albarel

Precomputation for Rainbow Tables Has Never Been so Fast

Gildas Avoine[1,3][0000−0001−9743−1779], Xavier Carpent[2][0000−0003−1697−6940], and Diane Leblanc-Albarel[3][0000−0001−5979−8457]

[1] CNRS, INSA Rennes, IRISA, France
{gildas.avoine, diane.leblanc-albarel}@irisa.fr
[2] KU Leuven, Belgium
xavier.carpent@kuleuven.be

Abstract. Cryptanalytic time-memory trade-offs (TMTOs) are techniques commonly used in computer security e.g., to crack passwords. However, TMTOs usually encounter in practice a bottleneck that is the time needed to perform the precomputation phase (preceding to the attack). We introduce in this paper a technique, called *distributed filtration-computation*, that significantly reduces the precomputation time without any negative impact the online phase. Experiments performed on large problems with a 128-core computer perfectly match the theoretical expectations. We construct a rainbow table for a space $N = 2^{42}$ in approximately 8 hours instead of 50 hours for the usual way to generate a table. We also show that the efficiency of our technique is very close from the theoretical time lower bound.

Keywords: Cryptography · Time-Memory Trade-Offs (TMTO) · Rainbow Table · Distributed Precomputation.

1 Introduction

Inverting a hash function (or equivalent cryptographic problem) can be addressed using an exhaustive search when the problem is reasonably sized. An illustrative case is password cracking, which consists in recovering a password from its hash stored by the targeted system. The computation cost may be prohibitive, though, when the attack is repeated. A time-memory trade-off (TMTO) is then an efficient alternative to an exhaustive search. It consists of a precomputation phase – or offline phase – performed once then stored, and an online phase performed each time the hash function should be inverted. The precomputation phase is therefore exploited to accelerate the online phase.

A TMTO, introduced by Martin Hellman [1], offers a significant speedup in practice. Given a problem of size N (i.e., $N = |A|$ where A is the considered set of possible solutions) and a memory M, the time complexity of the online phase is $O(N^2/M^2)$ instead of N for the exhaustive search. It is worth noting that the time complexity of the precomputation phase remains $O(N)$. This means that using a TMTO makes sense in specific scenarios: the attack has to be performed

several times, the attack itself has to last for a short period of time ("lunch time" attack), or the attacker is not powerful enough to perform an exhaustive search but he can download the result of a precomputed phase, stored in what is called *tables*.

Hellman's work has been improved over time, particularly with the rainbow tables [2] and the distinguished points [3]. These variants are faster [2,4,5] than the original time-memory trade-off. A few improvements [6,7] have been suggested on the distinguished points, and the rainbow tables also benefited from various optimizations concerning the online phase, notably the way of storing and using tables [8,9,10,11], checkpoints [12] and the use of data [13]. In 2016, Lee and Hong [4] demonstrated that, in the absence of excessive constraints such as a very limited memory, rainbow tables are the most efficient TMTOs for both online and precomputation phases. We consequently focus on rainbow tables in this paper.

The precomputation phase is very costly, though, typically of the order of $160\,N$ when considering practical scenarios, even for rainbow tables. The precomputation phase must consequently be distributed on many computers.

This paper introduces *distributed filtration-computation*, a technique that drastically decreases the cost of the precomputation phase and that is compliant with a distribution of that phase. The filtration process identifies computations that will eventually be useless, avoiding so to carry out computations that would be thrown out at the end precomputation phase. The distribution trivially consists in sharing the computing load among several computing units. In common scenarios, the technique we introduce divides by 6 the precomputation time, but the speedup can be much higher in extreme cases, namely when considering what are called *maximum* tables. As far as we know, this is the first time a technique is introduced to improve the precomputation phase of TMTOs.

After providing background on rainbow tables in Section 2, we introduce the filters and a lower bound on the precomputation in Section 3. We provide a distributed version of the filters in Section 4, with an optimization algorithm for their positions, and we finally illustrate the theory with practical results in Section 5. Note that Appendix D contains a table that recaps the notations used through this paper.

2 Background

2.1 Rainbow Tables

Given a hash function $h : A \to B$, and given $h(x) \in B$, the purpose of a TMTO is to retrieve $x \in A$. To do so, the precomputation phase of the TMTO precomputes rainbow *matrices*, which consist of *chains* of elements $x_i \in A$ $(0 \leqslant i \leqslant t)$ such that $x_{i+1} = f_i(x_i)$, where x_0 is an arbitrary value, introduced below, and f_is are *hash-reduction* functions defined as follows:

$$\begin{aligned} f_i : A &\to A \\ x &\mapsto R_i(h(x)) \end{aligned}$$

where each $R_i : B \to A$ is a *reduction* function, that is a function that aims to map each value in B to a value in A. The choice of the reduction functions is out of the scope of this article, and interested readers can refer to [2]. Note however that using a different reduction function in every iteration is a key feature of rainbow-based TMTOs that reduces the number of merging chains. It is also important to note that the execution time of a reduction function is negligible compared to that of a hash function.

Once the precomputation phase is completed, the matrix (see Figure 1) consists of $m \times (t+1)$ values denoted $X_{j,i}$ with $0 < j \leqslant m$ and $0 \leqslant i \leqslant t$, and $X_{j,i}$ the element in row j and column i.

The *length* t, of a chain is the number of application of the Hash-Reduction functions f_i performed to construct the chain.

$$\begin{array}{ccccccc}
 & f_1 & & f_2 & & & \\
\mathbf{X_{1,0}} & \longrightarrow & X_{1,1} & \longrightarrow & X_{1,2} & \ldots. & \mathbf{X_{1,t}} \\
 & f_1 & & f_2 & & & \\
\mathbf{X_{2,0}} & \longrightarrow & X_{2,1} & \longrightarrow & X_{2,2} & \ldots. & \mathbf{X_{2,t}} \\
\vdots & & \vdots & & \vdots & & \vdots \\
 & f_1 & & f_2 & & & \\
\mathbf{X_{m,0}} & \longrightarrow & X_{m,1} & \longrightarrow & X_{m,2} & \ldots. & \mathbf{X_{m,t}}
\end{array}$$

Fig. 1. Rainbow Matrix

Once the rainbow matrix is computed, only the first column that contains the so-called *Start Points* (SP) and the last column that contains the *End Points* (EP) are saved in what is called a *table*[3]. All intermediary columns are discarded to save memory.

Using a different reduction function in each column reduces the number of merging chains. Indeed, with rainbow tables, two chains merge if they collide in the same column.

From here on we consider a single matrix for the sake of clarity, but a rainbow-based TMTO usually consists of a few independently-computed matrices in order to reach a high success rate, typically 4 tables guarantee the success of the online phase with a probability greater than 99.96%.

2.2 Clean Rainbow Tables

Using a different reduction function per column reduces the number of merging chains, but two chains colliding in the same column can still happen. Detecting such a merge is however trivial, as it necessarily leads to equal EPs.

[3] This paper will not discuss how to use the rainbow tables during the online phase, as the approach is developed in other papers.

Given that colliding chains make the TMTO less memory-efficient due to the overlapping values, Philippe Oechslin introduced [2] tables without merges, which are called *clean* tables[4][14].

Merging chains are deleted to obtain clean tables, so if a matrix contains m_0 chains, the corresponding clean table only contains $m_t < m_0$ chains, with m_t the number of chains with distinct EPs in a matrix of length t.

2.3 Maximum Rainbow Tables

Given a set A with $N = |A|$, there is a limit to the number of EPs that can be obtained without duplicates. This number, denoted $m_t^{\max}$, depends on N and the number of columns t in the table[5]. $m_t^{\max}$ is obtained from Eq. (1) [12] where m_i is the theoretical number of different elements in column i:

$$m_i \approx \frac{2N}{i+\gamma}, \quad \text{with} \quad \gamma = \frac{2N}{m_0}. \tag{1}$$

Given a number of columns $t+1$ and a sufficiently large number of elements N, the expected maximum number of chains $m_t^{\max}$ per clean rainbow table is hence given by Theorem 1 [12].

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

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

Proof. The proof is presented in [12].

In practice, computing $m_0 = N$ chains is prohibitively expensive, so the chosen m_0 is generally markedly smaller than N. This produces tables of size $\alpha\, m_t^{\max}$ with $0 < \alpha < 1$. Usually, chains are generated until a satisfactory α is reached and m_0 is then determined retrospectively. A satisfactory α allow to have an online success rate close to the rate of a maximal table[6]. When α is close to one (e.g., $\alpha = 0.95$), the table is said to be *quasi-maximum*.

3 Filtering Chains

3.1 Preliminary Result on Quantifying Precomputation

In order to generate clean tables containing $m_t = \alpha\, m_t^{\max}$ chains, the common approach consists in computing chains until obtaining the desired number of

[4] Initially called *perfect* tables by Philippe Oechslin.

[5] The number of elements in a clean table is constant which implies that t is inversely proportional to m. The larger m is, the faster the online phase will be, but the more memory is needed for storage and inversely.

[6] The probability of success for a single maximal table is 86% when t is large. For the same t and a table of size $0.95m_t^{max}$, the probabilitity of success for a single table is 85%

unique end points. We provide in Lemma 1 and Proposition 1 formulas to predict the value m_0 required to reach on average $\alpha\, m_t^{\max}$ unique end points. α is called the *maximality factor*.

Lemma 1. *Let $r = m_0/m_t^{\max}$. The expected number of unique EPs is given by:*

$$m_t \approx \frac{1}{(1+\frac{1}{r})} m_t^{\max}.$$

Proof. From Eq. (1), we have $m_0 = \frac{2N}{\gamma}$. Given that $m_0 = r\, m_t^{\max}$, we can write $\gamma = \frac{2N}{r\, m_t^{\max}}$. Using Theorem 1, we then obtain $\gamma \approx \frac{t+2}{r} \approx \frac{t}{r}$. Replacing γ in Eq. (1) for $i = t$, we finally have:

$$m_t \approx \frac{2N}{t(1+\frac{1}{r})} = \frac{1}{(1+\frac{1}{r})} m_t^{\max}.$$

□

Proposition 1. *With a target of $m_t = \alpha\, m_t^{\max}$ unique endpoints, $m_0 = r m_t^{\max}$ chains need to be generated, with:*

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

Proof.

From Lemma 1:	$m_t \approx \frac{1}{(1+\frac{1}{r})} m_t^{\max}$.
Since $m_t = \alpha\, m_t^{\max}$:	$\alpha \approx \frac{1}{(1+\frac{1}{r})}$
Which conduct to :	$\frac{1}{r} \approx \frac{1}{\alpha} - 1 \Leftrightarrow r \approx \frac{\alpha}{1-\alpha}$

□

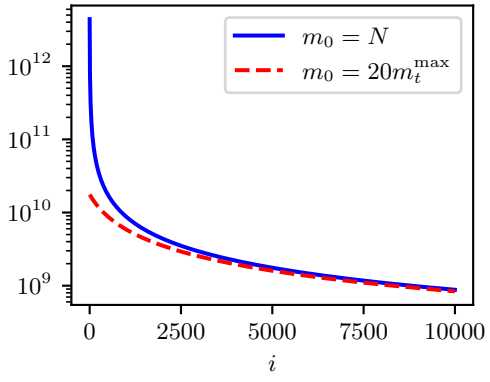

Fig. 2. Number of unique elements m_i remaining in column i (log scale) according to the value of m_0 for $N = 2^{42}$ and $t = 10\,000$.

To illustrate Proposition 1, let us consider the case $r = 20$, which results in $m_t \approx 0.95\, m_t^{\max}$. This is a reasonable number that provides a high quality table (m_t relatively close to $m_t^{\max}$), while significantly reducing the precomputation cost (compared to the $m_0 = N$ case). With $N = 2^{42}$ and $t = 10\,000$ for instance, generating $20\, m_t^{\max}$ chains instead of N brings the number of chains to be generated from 440×10^{10} to 1.7×10^{10} – a 258-fold reduction – while keeping the number of endpoints very close to the maximum, and thus preserving the density of the trade-off.

Figure 2 illustrates the difference between the scenarios $m_0 = rm_t^{\max}$ and $m_0 = N$ in terms of maximum number of elements in each column.

3.2 Intermediary Filtration

Classically, computing a rainbow table requires $m_0 \times (t+1)$ elements to compute, although only $m_t \times (t+1)$ elements are represented in the final (clean) matrix. Discarding merged chains at the end of the precomputation is a wasted effort, because a single chain is kept among multiple chains with the same EP (i.e., hash operations performed after a merge occurs are useless). An option to mitigate this waste is to remove duplicated values progressively.

So, instead of computing the full chains in a row, from the SPs to the EPs, chains are divided into sub-chains, and merging chains are detected and discarded at the end of each sub-chain. A *sub-chain* is delimited by Intermediary Points (IPs). Computation of chains is thus performed "column by column" (or group of columns after group of columns), as opposed to the typical "chain-by-chain" method. A *filter* is placed in selected columns (IPs): when all sub-chains have been computed up to the filter, a *filtration* is performed: only one of the merged chains is saved. Figure 7 in appendix C, presents the mechanism of filtration with an example zooming on 2 filters.

3.3 Filtration in Each Column

The minimum number of elements to be computed to generate a table is obtained when duplicates are removed in each column, i.e., if each chain of length t is divided in t sub-chains of length 1.

Proposition 2. *Let m_i denote the number of unique elements in column i of a rainbow matrix. The number P of hash operations to precompute a $m_t \times (t+1)$ clean rainbow matrix is lower bounded by:*

$$P \geq \sum_{i=0}^{t-1} m_i.$$

Proof. Given that the minimum hash operations to compute in order to obtained a table is when duplicates are removed in each column and that m_i denote the number of unique elements in column i, then the expression of the lower bound is trivial.

□

Theorem 2 quantifies this with results from Section 3.1.

Theorem 2. *Given $m_0 = rm_t^{\max}$ the number of SPs, $t+1$ the number of columns, and $r \ll t$, we have that the naïve precomputation cost is:*

$$P_{naive} = m_0 t \approx 2rN, \tag{2}$$

and the minimum precomputation cost is:

$$P_{min} = \sum_{i=0}^{t-1} m_i \approx 2N \ln(1+r). \tag{3}$$

Proof. The proof of Eq. (2) follows directly from $m_0 = \frac{2N}{\gamma} \approx \frac{2rN}{t}$. For Eq. (3), we have:

$$\begin{aligned}\sum_{i=0}^{t-1} m_i &= 2N \sum_{i=0}^{t-1} \frac{1}{i+\gamma} = 2N \sum_{i=\gamma}^{t+\gamma-1} \frac{1}{i} = 2N \left[\sum_{i=1}^{t+\gamma-1} \frac{1}{i} - \sum_{i=1}^{\gamma-1} \frac{1}{i} \right] \\ &= 2N \left[H_{t+\gamma-1} - H_{\gamma-1} \right] \approx 2N \left[\ln(t+\gamma-1) - \ln(\gamma-1) \right] \\ &= 2N \ln\left(\frac{t+\gamma-1}{\gamma-1} \right)\end{aligned}$$

with $H_n = \sum_{k=1}^{n} \frac{1}{k}$ the n-th harmonic number. Using $\gamma \approx \frac{t}{r}$ and given that $r \ll t$, the exected result is obtained. □

For values of r such that $m_0 \ll N$ (i.e., "reasonable" values), we can make the approximation that γ is large (leading itself to the asymptotic approximation of H_n). This allows an expression of $P_{\min}$ that only depends on N and r and is in particular virtually independent of t. For $m_0 = N$ however these approximations do not hold, and the resulting expression of $P_{\min}$ does depend on t (Corollary 1). We remark that the precomputation cost in all cases is linear in N.

Corollary 1. *For the case $m_0 = N$, precomputation costs are respectively $P_{naive} = Nt$ and $P_{min} \approx 2N(H_{t+1} - 1)$, with H_n the n-th harmonic number.*

Proof. The expression for $P_{\min}$ results from instantiating $2N \left[H_{t+\gamma-1} - H_{\gamma-1} \right]$ (similarly to the proof of 2) to $\gamma = 2$ (from Eq. (1)).

From Theorem 2, we observe, for instance, that using a filter in each column with a typical $r = 20$ reduces the number of performed hash operations by about 85% (regardless of N or t). Tables 1 and 2 display the maximum speedup $P_{\text{naive}}/P_{\min}$ that can be obtained when filtering in each column with respect to no intermediary filtering. Results in Table 1 are valid for any (sensible) N and t.

Table 1. Speedup for quasi-maximum tables for different values of r.

r	10	15	**20**	30	50
$\frac{r}{\ln(1+r)}$	4.17	5.41	**6.57**	8.74	12.72

Table 2. Speedup for maximum tables of various lengths.

t	1 000	10 000	100 000
$\frac{t}{2(H_{t+1}-1)}$	77.03	568.98	4508.50

3.4 Filtration in Chosen Columns

In practice, it may not always be beneficial to filter in every column, because this may involve an excessive overhead due to the filtering cost (results provided in Section 3.3 consider the number of hash operations, but they do not consider the additional time due to filtration and communication). Before considering this cost, an intermediary step consists in evaluating the number P of hash operations to be performed if the filtering technique is applied to $a < t+1$ columns only:

$$P = \sum_{i=0}^{a} m_{c_i}(c_{i+1} - c_i), \tag{4}$$

where c_i the column of the i-th filter, and $c_0 = 0$ and $c_a = t+1$. Given a number of filters a and t very large compared to this number[7], the optimal average number of hash operations is given in Theorem 3.

Theorem 3. *The optimal average number of hash operations for precomputation with a filters and $a \ll t$ is:*

$$P = 2Na \left[\left(\frac{t+\gamma-1}{\gamma} \right)^{\frac{1}{a}} - 1 \right].$$

The optimal placement of the filters is given by:

$$c_i = \gamma \left(\frac{t+\gamma-1}{\gamma} \right)^{\frac{i}{a}} - \gamma + 1.$$

Proof. See Appendix A. □

[7] If a is too close to t, several filters could be affected to the same column. Choosing $a \ll t$ is not a problem, as presented in Figure 3.

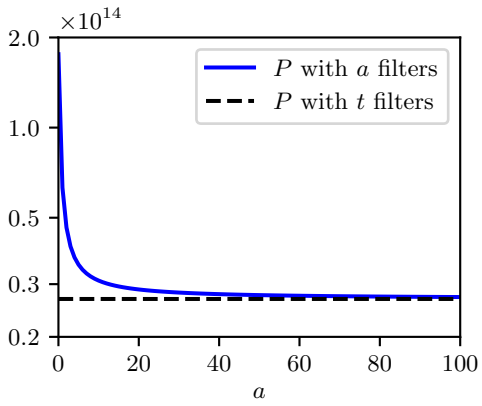

Fig. 3. Number of hash operations P (log scale) according to the number of filters used a, with $N = 2^{42}$, $t = 10\,000$, and $r = 20$. P with a filters according to Theorem 3 and P with t filters according to Theorem 2.

Figure 3 illustrates Theorem 3 for $N = 2^{42}$, $t = 10\,000$, and $r = 20$. It shows the number of hash operations needed for precomputation with varying number of filters a, placed according to Theorem 3. It also displays the lower bound (P_{min}) in terms of hash operations, which is reached when a filter is applied in each column. The case $a = 1$ corresponds to P_{naive} (filtration only in the last column).

It indicates diminishing returns in increasing a. For instance in that scenario, P with a single filter is 2.88 times faster than using no filter. On the other hand, using a filter in each column is only about 1% faster than using 50 filters. Note that the optimal distribution of filters is actually not uniform. Indeed, detecting merges as soon as they occur avoids to waste time computing the useless remaining parts of the chains. As a consequence filters are mostly located on the left-hand parts of the chains.

This, together with the fact that the cost of non-hashing operations is not necessarily negligible in a practical implementation, implies that a limited number of filters is preferable. The next sections discuss an implementation and quantify this effect.

4 Distributing Precomputation

4.1 Distribution and Filtration

Even with filters, generating rainbow tables on a single computing node takes too much time when considering practical cases. To compute m_0 chains of length t, $m_0 t = 2rN$ hash operations are needed. For example, for $N = 2^{42}$ and $r = 20$, building a clean table of about $0.95\, m_t^{\max}$ chains with optimal filters requires

2.7×10^{13} hash operations, which would take about 40.5 days on a single processor core.

To generate tables of such size, precomputation should be distributed. Without filters, the precomputation phase is easily parallelizable. If n_h hashing nodes are available, the total precomputation time is simply divided by n_h, with each hashing node performing $\frac{2rN}{n_h}$ hash operations (assuming only the hashing time is taken into account and all hashing nodes have the same performances).

However, as seen in Section 3.2, precomputing a (quasi-maximum) clean rainbow table without intermediary filtering wastes significant effort due to merging chains. Therefore considering both distributing and filtering is essential[8].

4.2 Distributed Architecture

Distribution of the precomputation phase with intermediate filtering requires the nodes to communicate. We consider in what follows that n_h nodes are dedicated to perform hash operations and n_f nodes to filter chains, with $n_h + n_f = n$. Filtration and computation of chains are carried out in parallel[9], in an effort to minimize the number of non-hashing operations. Sub-chains computed by the n_h *hashing nodes* are sent eagerly to the n_f *filtration nodes*.

In the environments and problem sizes we considered, the filtration effort was not significant enough compared to the hashing effort to warrant using more than a single filtration node. It is possible that for other environments (e.g., low bandwidth) or larger spaces, dedicating more nodes to filtering could be beneficial.

4.3 Estimation of the Precomputation Time

Precomputation Process In an environment with a single filtration node, this node is also in charge of the different tasks sequencing. A *job* is defined as a number of sub-chains to be computed between two filters. A job of size s contains s SP-IP pairs. Precomputations are hence divided in two main parts: jobs to be performed between filters and the filtration of these jobs. Precomputations consists of managing, for each filter, the sending of jobs to the hashing nodes and the filtration of the completed jobs.

The rationale behind choosing to bundle sub-chains into jobs is to mitigate the communication overhead. Using $s = 1$ is, for instance, a bad idea, because the overhead due to the communication, would significantly hinder performance. On the other hand, using a very large s may result in additional idle time by the computing nodes, for instance if there is no available chains left to compute for an idling computing node but some other computing nodes are still busy[10]. In what follows, we consider that the value of s is reasonable.

[8] Distributed filtration-computation has negligible impact on the online phase, including its many improvements. See Appendix B for more details.

[9] Technically, filtration starts and stops slightly after computation of chains.

[10] Experiments show that, for the typical problem sizes and architecture considered, choosing s to be anywhere from 1 000 to 100 000 mitigates both of these issues. The

Once receiving a job, a hashing node starts from each IP to compute the new IPs corresponding to the column of the next filter. Once these computations are done, it returns the SPs and the new IPs to the filtration node, which sends a new job back to it. The filtration node purpose is to receive jobs from hashing nodes and send new ones as soon as it receive it. At the meantime the filtration node filters already received jobs. This procedure is repeated between each filter until the end of the computation phase. As described in Section 3.4, it is not possible in a distributed architecture to filter in every column since the filtration adds an overhead. The problem of choosing the positions of the filters is discussed in Section 4.4.

Impacting Functionalities Several functionalities are required to perform the precomputation phase: hashing (i.e., computing sub-chains), filtration, and communication. In this section the way to evaluate the time needed for each of these functionalities is described.

Hashing time (H). Jobs computations are carried out by hashing nodes. Given a filters with $a < t+1$, the total number of hash operations to be performed is $\sum_{i=1}^{a} m_{c_{i-1}}(c_i - c_{i-1})$ (Eq. (4)) with c_i the column of the i-th filter, $c_0 = 0$ and $c_a = t+1$. A hashing node can perform v_h applications of $f_i = R_i \circ h$ per second. v_h is determined before the beginning of precomputations and depends of the hashing nodes performance. Computations of chains are considered to be done in parallel, by n_h hashing nodes with equal performances. The total hashing time can therefore be estimated as:

$$H = \frac{1}{n_h v_h} \sum_{i=1}^{a+1} m_{c_{i-1}}(c_i - c_{i-1}). \tag{5}$$

Filtration time (F). For a filter in column i, the number of points that have to be filtered is $m_{c_{i-1}}$. The total number of points that have to be filtered in the entire precomputation is hence $\sum_{i=1}^{a+1} m_{c_{i-1}}$. We consider that a filtration node can perform v_f filtrations per second. The total time due to filtration is thus:

$$F = \frac{1}{v_f n_f} \sum_{i=1}^{a+1} m_{c_{i-1}}. \tag{6}$$

We also model for a potential overhead due to the filtration. This can for instance result from processing the output of the filtration into jobs to be sent to hashing nodes. This overhead depends on the number of elements generated and depends on the implementation (filtration algorithm, memory allocation, etc.). Given d_o the average overhead time per point, the total overhead O can be expressed as:

$$O = d_o \sum_{i=1}^{a+1} m_{c_{i-1}}. \tag{7}$$

particular choice of s therefore has negligible impact on the performance, provided it lies in that range

Communication time (C). Communication time is the time needed for the communication of jobs between filtration nodes and computing nodes. Let d_c be the average time for a job to be sent from a filtration node to a hashing node and back. We assume that when a communication is in progress with one hashing node all the other hashing nodes are computing. The impacting communication time can then be estimated by:

$$C = \frac{d_c}{n_h} \sum_{i=1}^{a+1} m_{c_{i-1}} \tag{8}$$

Total time. Given that computation of sub-chains and filtration are performed in parallel, the most impacting component in the total time spend to generate a rainbow table is the maximum time between the hashing time H and the filtration time F i.e., $\mathrm{Max}(H, F)$[11]. To obtain the total time, the communication time has to be added as well as the overhead time due to filtration. The total time T needed to generate a clean rainbow table is hence:

$$T = \mathrm{Max}(H, F) + C + O. \tag{9}$$

4.4 Optimal Configuration

The number of filters and their positions have a considerable impact on the precomputation time. Let a *configuration* be a set $C = \{c_1, \ldots, c_a\}$, where a is the number of filters, and c_i the position (column number) of the i-th filter. Let C_a^* be the configuration of a filters that minimizes Eq. (9), and $C^* = \min_a C_a^*$.

Due to the various operations outside of hashing, in particular the filtering process (which, to some extent, can be done in parallel to hashing) and other communication/data processing overheads, the configuration given by Theorem 3 typically gives sub-optimal results. For this reason we rely instead of numerical minimization of Eq. (9), which models the precomputation time given by our implementation.

We settled on a truncated-Newton method [15], an optimization algorithm suitable to solving bounded optimization problems with many variables (see e.g., [16] for a thorough description). The minimization is used to find C_a^*, coupled with an exhaustive search on a[12]. The optimality of the configuration found by the numerical minimization is predicated on the two following conjectures: (1) C_a^* is a convex function of a and (2) Eq. (9) is smooth enough (w.r.t. C) to guarantee or approach the conditions of optimality of the truncated-Newton

[11] In general, for an architecture with a single filtration node, hashing time is much bigger than the filtering time. If the parameters of the problem and the architecture are such that it is not the case, then an other architecture with several filtration nodes should be considered.

[12] To keep things efficient, the search is from 0 up to a reasonable upper bound $a_{\max}$. A more sophisticated approach could be used here (e.g., Newton descent on a), but we found it to be unnecessary.

search[13] [15]. We offer no proof of these conjectures, but note that they seem to hold true both intuitively and after extensive testing.

Regardless of the validity of these conjectures however, the configuration obtained through numerical minimization presents a significant improvement over the analytical minimization that assumes no overhead or filtration cost (Theorem 3). In addition, the estimated precomputation time comes very close to the theoretical minimum, as detailed in Section 5.

5 Experiments

5.1 Computing Environments

We conducted our experiments on two different environments that are described below. We benchmarked these two environments before starting the precomputation phase in order to measure the hashing speed v_h, the filtration speed v_f, the overhead d_o related to the implementation of the filtration, and the communication cost d_c. The benchmark has been done by generating tables on a small-sized problem ($N = 2^{32}$) with filters placed according to Theorem 3.

Environment 1 consists of a computer hosting two AMD EPYC 7742 3.2 GHz processors composed of 64 cores each, for a total of 128 cores[14]. The benchmark measured $v_h = 7\,747\,002$ hashs per seconds, $v_f = 15\,949\,709$ filtrations per second, and $v_0 = 1.37 \times 10^{-10}$. The communication overhead is negligible compared to v_h and v_f, it can hence be considered that $d_c = 0$ which implies that $d_o + \frac{d_c}{n_h} = d_o = 1.37 \times 10^{-10}$ seconds to treat one point.
Environment 2 is a cluster of 8 computers with 2 CPUs per machine and 14 cores for each CPU, i.e., a total of 224 cores. Each CPU is an Intel Xeon E5-2680 v4 (Broadwell, 2.40GHz, 14 cores). The computers are directly connected through switches, meaning that they commnicate using the ethernet protocol. The benchmark provided $v_h = 6\,403\,611$ hashs per seconds, $v_f = 7\,918\,745$ filtrations per second, and $d_o + \frac{d_c}{sn_h} = 7.5 \times 10^{-10}$ seconds to treat one point.

5.2 Filtration Implementation

For the filtration, we used an open addressing hash table with the following parameters:

- A load factor $\lambda = 2/3$, which is a good compromise between size overhead and low probability of collision.
- The number of slots of the table is $k = m_{c_i}/\lambda = 1.5m_{c_i}$ with m_{c_i} the theoretical number of different points after filtration (given by Eq. (1)).

[13] Namely strong convexity and Lipschitz-continuous Hessian.
[14] We used 127 of them to be sure that all cores are fully exploited for the precomputation, and the last core was left available for the basic operations performed by the system.

– The hash function used is $IP \bmod k$.

We used linear probing for collision resolution (with interval of 1). This is appropriate because inputs to the hash table are uniformly distributed in A (by construction). At each filtration the following steps are carried out:

– A hash table of size $k = 1.5m_{c_i}$ is created.
– As soon as a job is received by the filtration node this job is filtered as follow:
  1. For each couple (SP;IP) of the job, $IP \bmod k$ is computed.
  2. The index $IP \bmod k$ of the table is checked.
  3. If at the index computed no value is present then IP and its corresponding SP are inserted at this index.
  4. If at the index computed the same value equals to IP is present then a merge has occurred between two chain and IP and its corresponding SP are deleted.
  5. If at the index computed, a different value of IP is present then a new value of index is computed and is equals to $IP + 1 \bmod k$, the index $IP + 1 \bmod k$ is checked and the steps 3. to 5. are repeated.
– When all jobs have been filtered, the hash table is scanned and all the IPs and their corresponding SPs are transferred by copying them to an array in a form facilitating the sending of the jobs.
– The hash table is deleted.

5.3 Positions of the Filters

We conducted experiments where filters were optimally placed using the Truncated Newton Constrained (TNC) algorithm (Section 4.4) applied to Eq. (9).

For environment 1, the optimal configuration was 31 filters, with positions as provided in Figure 4. The latter figure also displays the positions of the 31 filters in the theoretical case where filtering and communicating are free (Theorem 3).

Figure 5 displays the number of hash operations needed to generate a clean rainbow table in our scenario, when: (left case) there are no filters, which is the current state of the art; (middle case) there are filters optimally placed, which is our approach ; and (right case) filtration and communication are free, with so a filter in each column, which is the theoretical lower bound. It is worth noting that our approach is tightly close to the theoretical lower bound (about 10% of the theoretical lower bound).

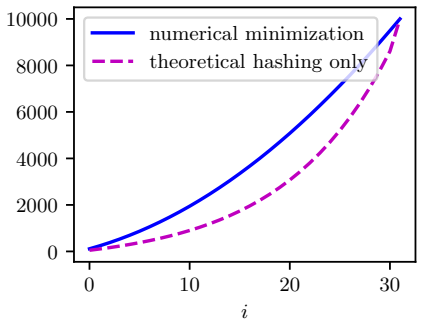

Fig. 4. Positions of the 31 filters

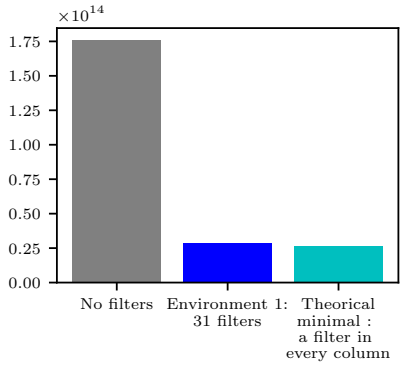

Fig. 5. Number of hash operations

5.4 Considered Parameters

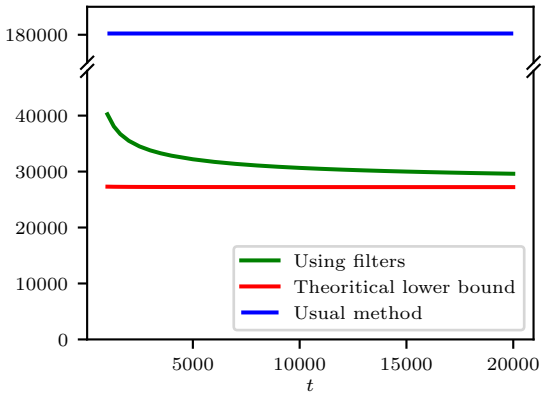

Fig. 6. Time (sec) to generate a clean rainbow table according to t ($N = 2^{42}$, $r = 20$)

Experiments were performed with the following parameters that represents a realistic scenario: $N = 2^{42}$, $t = 10\,000$, and $r = 20$.

Corollary 1 provides the number of starting points in the experiment: $m_0 = 2rN/t \approx 1.76 \times 10^{10}$. According to Proposition 1, the expected number of chains in a single such table is: $0.9524\, m_t^{\max} \approx 8.38 \times 10^8$.

The chain length ($t = 10\,000$) has an impact on the online phase: when t decreases, the time required for the online phase decreases as well, but the required memory increases. The chain length also has an impact on the precomputation

phase when filters are used, but this effect is *much* smaller. As shown in Figure 6, the smaller t is, the greater the precomputation time.

We chose $t = 10\,000$ because this value leads to a very fast online phase in the order of a few seconds with a reasonably-sized memory (for $N = 2^{42}$). Choosing $t = 20\,000$ for instance would provide a 20% faster precomputation but would increase the time of the online phase four-fold.

Figure 6, also shows that our experimental results (green curve) is close to the theoretical lower bound (red curve) and that our method is much more efficient that the usual way to generate table (blue curve) as detailed in Section 5.5.

5.5 Results

Environment 1. Precomputing a single rainbow table without any filter requires $m_0\,t$ hash operations in our scenario, which is 100×2^{42}. Given that $v_h = 7\,747\,002$ and the environment consists of 127 cores (each core corresponding to one node) with 1 filtration node and 126 hashing nodes, the precomputation time is estimated to be 180 225 seconds (50 hours and 3 minutes), which is very close from our experimental result of 179 850 seconds (49 hours and 57 minutes).

Now, using 31 filters optimally placed significantly reduces the precomputation time. Using Eq. (9), we obtain that the predicted precomputation time is as low as 30 657 seconds (about 8 hours). Filters thus divide by 5.9 the precomputation time in this scenario. The experimental result is 31 029 seconds.

Environment 2. According to the TNC algorithm, without any filter, the precomputation time of a single rainbow table should be around 123 194 seconds (about 34 hours and 13 minutes).

If 11 filters optimally placed are used, our experimental results show that a table can be generated in 26 499 seconds (about 7 hours and 20 minutes). According to TNC algorithm with the parameters for this second environment, given in Section 5.1, this precomputation time is estimated to 26 139 seconds (about 7 hours and 12 minutes). Our experimental results are therefore very close from the predicted one.

Utilization of filters on this environment hence allows to generate a table in about 5 times less time than the naive method. Table 3 provides a summary of the results obtained for environments 1 and 2.

Table 3. Summary of the results ($N = 2^{42}$, $t = 10\,000$, $r = 20$)

Scenario	#Filters	#Hashes ($\times 10^{12}$)	#Cores	Time	
				Experimental	Predicted
Environment 1 (state of the art)	0	176	127	179 850	180 225
Environment 1 (our approach)	31	28	127	31 029	30 657
Environment 1 (theoretical bound)	10 000	26.8	127	-	27 452
Environment 2 (state of the art)	0	176	224	123 717	123 194
Environment 2 (our approach)	11	31	224	26 499	26 139
Environment 2 (theoretical bound)	10 000	26.8	224	-	18 765

6 Conclusion

This paper introduces the concept of distributed filters to precompute rainbow tables. Such tables are widely used by the community of security experts, especially, but not only, to tests passwords. Given that the precomputation phase is highly resource-consuming, the technique we introduce in this paper has a strong practical impact. It also comes with formulas to compute the optimal positions of the filters, and to evaluate the precomputation time.

We illustrate our technique on a typical scenario, namely a problem of size $N = 2^{42}$ ($t = 10\,000$ and $r = 20$). In such a scenario, the precomputation phase requires 1.76×10^{14} hash operations, which takes about 50 hours on a 128-core computer, while our technique requires 2.8×10^{13} hash operations, which were performed (including filtering) in about 8 hours and 36 minutes on the same 128-core computer. Distributed filtration-computation thus divides by about 6 the expected precomputation time. It is also close to the theoretical lower bound of 27 452 seconds i.e., 7 hours and 33 minutes (for $r = 20$), with the difference due to the filtering and communication overheads.

We considered a typical scenario in the sense that we used quasi-maximum tables ($r = 20$) instead of maximum tables. Such maximum tables (corresponding to $r = 5\,001$ in our case) are usually considered in the literature, but they are never used in practice because precomputing maximum tables is prohibitive. However, considering maximum tables would make distributed filtration-computation much more valuable still (e.g., increasing the speedup from 6 to 4 500 with the same parameters).

It is worth noting that our technique to speed up the precomputation phase has been applied to classical rainbow tables but we state in this article that it is fully compliant with the improvements published during the last years to make the online phase more efficient.

References

1. Martin E. Hellman. A cryptanalytic time-memory trade-off. *IEEE Trans. Inf. Theory*, 26(4):401–406, 1980.
2. Philippe Oechslin. Making a faster cryptanalytic time-memory trade-off. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003*, pages 617–630, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.
3. Dorothy Elizabeth Robling Denning. Cryptography and data security. USA, 1982. Addison-Wesley Longman Publishing Co., Inc. p.100.
4. Ga Won Lee and Jin Hong. Comparison of perfect table cryptanalytic tradeoff algorithms. volume 80, page 473–523, USA, September 2016. Kluwer Academic Publishers.
5. Jin Hong and Sunghwan Moon. A comparison of cryptanalytic tradeoff algorithms. volume 26, pages 559–637. Springer, 2013.
6. 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*, pages 131–145, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
7. Francois-Xavier Standaert, Gael Rouvroy, Jean-Jacques Quisquater, and Jean-Didier Legat. A time-memory tradeo. using distinguished points: New analysis & fpga results. In Burton S. Kaliski, çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002*, pages 593–609, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.
8. Gildas Avoine, Adrien Bourgeois, and Xavier Carpent. Analysis of rainbow tables with fingerprints. In Ernest Foo and Douglas Stebila, editors, *Information Security and Privacy*, pages 356–374, Cham, 2015. Springer International Publishing.
9. 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*, pages 144–157, Cham, 2014. Springer International Publishing.
10. Gildas Avoine and Xavier Carpent. Heterogeneous rainbow table widths provide faster cryptanalyses. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, page 815–822, New York, NY, USA, 2017. Association for Computing Machinery.
11. Gildas Avoine, Xavier Carpent, and Cédric Lauradoux. Interleaving cryptanalytic time-memory trade-offs on non-uniform distributions. In Günther Pernul, Peter Y A Ryan, and Edgar Weippl, editors, *Computer Security – ESORICS 2015*, pages 165–184, Cham, 2015. Springer International Publishing.
12. Gildas Avoine, Pascal Junod, and Philippe Oechslin. Characterization and improvement of time-memory trade-off based on perfect tables. volume 11, New York, NY, USA, July 2008. Association for Computing Machinery.
13. Alex Biryukov, Sourav Mukhopadhyay, and Palash Sarkar. Improved time-memory trade-offs with multiple data. In Bart Preneel and Stafford Tavares, editors, *Selected Areas in Cryptography*, pages 110–127, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
14. 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*, pages 306–323, Cham, 2017. Springer International Publishing.
15. Stephen G. Nash. A survey of truncated-newton methods. volume 124, pages 45–59, 2000. Numerical Analysis 2000. Vol. IV: Optimization and Nonlinear Equations.

16. Stephen G Nash. A survey of truncated-newton methods. *Journal of computational and applied mathematics*, 124(1-2):45–59, 2000.

Appendix

A Proof of Theorem 3

We have $P = \sum_{i=0}^{a} m_{c_i}(c_{i+1} - c_i)$. Deriving for each filter column position, we obtain:

$$\frac{\partial P}{\partial c_i} = \frac{\partial}{\partial c_i}\left[m_{c_i}(c_{i+1} - c_i) + m_{c_{i-1}}c_i)\right].$$

Inserting $m_i = \frac{2N}{i+\gamma-1}$ we have:

$$\begin{aligned}\frac{\partial P}{\partial c_i} &= 2N\frac{\partial}{\partial c_i}\left[\frac{c_{i+1} - c_i}{c_i + \gamma - 1} + \frac{c_i}{c_{i-1} + \gamma - 1}\right] \\ &= 2N\left[\frac{1}{c_{i-1} + \gamma - 1} - \frac{c_{i+1} + \gamma - 1}{(c_i + \gamma - 1)^2}\right].\end{aligned}$$

To minimize P, we must have $\frac{\partial P}{\partial c_i} = 0$, and thus:

$$c_i = \sqrt{(c_{i-1} + \gamma - 1)(c_{i+1} + \gamma - 1)} - \gamma + 1.$$

It is easy to verify that a solution to this recurrence relation with terminal conditions $c_0 = 1$ and $c_a = t$ is:

$$c_i = \gamma\left(\frac{t + \gamma - 1}{\gamma}\right)^{\frac{i}{a}} - \gamma + 1.$$

Replacing in the expression for P gives the expected result.

B Online Phase Improvements and their Impact on Precomputation

There exists many significant algorithmic improvements on the online phase and optimizations of the storage of tables. Their impact on the distribution and intermediate filtering of precomputation is briefly discussed below.

- Chain storage optimizations (prefix/suffix decomposition or compressed delta encoding [9]): lossless compression can be applied at the end of the table generation, with no impact on the precomputation process.
- Truncated endpoints [8]: Endpoints can be truncated at the end of the table generation, again with no impact.
- Checkpoints [12,8]: Saving checkpoints can be done during the filtered and distributed precomputation with ease, although specific care must be taken. Hashing nodes must be made aware of which columns are checkpoint columns, and the filtration node needs to keep track of this. This adds no significant burden on either.

– Heterogeneous tables [10]: Precomputation of tables of different shapes is done independently, regardless of whether they operate on the same input set. Consequently The use of heterogeneous tables has no impact on precomputation improvements.

– Interleaving [11]: Just like with heterogeneous tables, the different tables are independently computed, again having no impact on precomputation improvements

C Intermediary Filtration

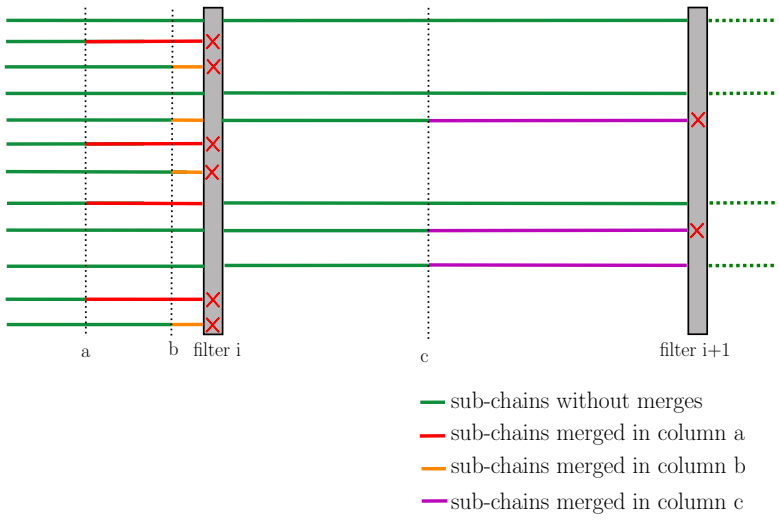

Fig. 7. Intermediary filtration with 2 filters.

D Notation Through this Paper

Table 4. Notation

N	Cardinality of A
t	Number of columns in a table
m	Number of rows in a table
m_i	Number of different elements in column i
a	Number of filters
n_h	Number of computing nodes
n_f	Number of filtration nodes
n	Number of nodes $n = n_h + n_f$
α	maximality factor
r	$\frac{\alpha}{1-\alpha}$
c_i	Column of the ith filter
s	Job size
v_h	hashing speed
v_f	Filtration speed
d_c	Communication cost
d_o	Filtration overhead