How to Handle Rainbow Tables with External Memory

Gildas Avoine[1,2,5], Xavier Carpent[3], Barbara Kordy[1,5], and Florent Tardif[4,5]

[1] INSA Rennes, France
[2] Institut Universitaire de France, France
[3] University of California, Irvine, USA
[4] University of Rennes 1, France
[5] IRISA, UMR 6074, France
florent.tardif@irisa.fr

Abstract. A cryptanalytic time-memory trade-off is a technique that aims to reduce the time needed to perform an exhaustive search. Such a technique requires large-scale precomputation that is performed once for all and whose result is stored in a fast-access internal memory. When the considered cryptographic problem is overwhelmingly-sized, using an external memory is eventually needed, though. In this paper, we consider the rainbow tables – the most widely spread version of time-memory trade-offs. The objective of our work is to analyze the relevance of storing the precomputed data on an external memory (SSD and HDD) possibly mingled with an internal one (RAM). We provide an analytical evaluation of the performance, followed by an experimental validation, and we state that using SSD or HDD is fully suited to practical cases, which are identified.

Keywords: time memory trade-off, rainbow tables, external memory

1 Introduction

A cryptanalytic time-memory trade-off (TMTO) is a technique introduced by Martin Hellman in 1980 [14] to reduce the time needed to perform an exhaustive search. The key-point of the technique resides in the precomputation of tables that are then used to speed up the attack itself. Given that the precomputation phase is much more expensive than an exhaustive search, a TMTO makes sense in a few scenarios, e.g., when the adversary has plenty of time for preparing the attack while she has a very little time to perform it, the adversary must repeat the attack many times, or the adversary is not powerful enough to carry out an exhaustive search but she can download precomputed tables. Problems targeted by TMTOs mostly consist in retrieving the preimage of a hashed value or, similarly, recovering a cryptographic key through a chosen plaintext attack.

Related work. Since Hellman's seminal work, noumerous variants, improvements, analyses, and successful attacks based on a TMTO have been published.

There exist so two major variants: the *distinguished points* [12] introduced by Ron Rivest and the *rainbow tables* [24] proposed by Philippe Oechslin. According to Lee and Hong [20], the rainbow tables outperform the distinguished points, though. As a consequence, we focus on the rainbow tables only, although most of our results might apply to other approaches as well. It is also worth noting that there exist time-memory-data trade-offs [8, 13, 17], which are particularly (but not only) relevant for attacking stream ciphers. We do not consider this particular family in this work.

Several algorithm-based improvements of the original rainbow table method have been suggested. They reduce either the average running time of the attack or the memory requirement, which are actually directly related to each other. In practice, any, even tiny gain can have a significant impact. Important improvements published so far include the checkpoints [5], the fingerprints [1], the delta-encoding storage [2], and the heterogenous tables [3, 4]. For more details about the analysis of TMTOs and their variants, we refer the reader to [7, 15, 16, 19].

Technology-related enhancements have also been suggested, for example on the implementation of TMTOs on specialized devices such as GPUs or FPGAs [10, 21, 23, 25]. GPU indeed provide a lot of parallel processing power at very affordable prices, and were therefore considered as a support of the rainbow scheme, but as far as hash function are involved, they are mainly used, e.g. in commercial products, to perform exhaustive searches. However, improvements benefiting from the technological advances in data storage have not yet been addressed so much. Most scientific articles published so far assume that the tables fit into the internal memory (RAM). In such a case, accessing the memory is fast enough to be neglected in the evaluation of the TMTO performance. As a consequence, only the computational cost of the cryptographic function to be attacked is considered in the analytic formulas [24]. Nevertheless, implemented tools, e.g., OphCrack [29] and RainbowCrack [27], deal with large-space problems that tend to outweigh the available internal memory. The tools must then use both the internal memory and some external memory. The algorithms used to balance the tables between the memories are poorly documented. To the best of our knowledge, only Kim, Hong, and Park [18] and Spitz [28] formally address this issue. In their article, Kim et al. explain which algorithms the existing tools use.

Finally, examples of successful attacks based on TMTO include (but are not limited to) breaking A5/1 [9] and LILI-128 [26], cracking Windows LM-Hash passwords [24] and Unix passwords [22], recovering keys from Texas Instruments' digital signature transponders [11] and from Megamos Crypto vehicle immobilizers [30].

Contribution. Storing rainbow tables in an external memory has been ignored up to now because this approach was considered impractical with mechanical hard disk drives (HDD). Indeed, HDDs are efficient in sequential reads but perform poorly when random accesses to the disk are required. TMTOs rely mostly on random accesses to the precomputed tables. However, storage devices improve a lot these years. In particular, solid state drives (SSD) are much faster than

HDDs and, although they are still expensive, their price has already decreased significantly. SSDs provide smaller latencies than HDDs because they do not have mechanical parts.

In this paper, we study the behavior of the rainbow tables when they do not fit in RAM. We consider two algorithms. The first one, provided by Lee and Hong in [20], consists in storing the tables in an external memory (Lee and Hong consider the SSD case only) and then filling the RAM with as many table rows as possible; the memory is then emptied and refilled with the subsequent rows. The second algorithm, which we suggest, consists in keeping the tables in the external memory and performing direct accesses to that memory. RAM is very fast but also very expensive, and its size is still quite limited today. SSD is slower but reasonably priced. Finally, HDD is slow but also very cheap. We analyze the relevance of storing the precomputed data on an external memory (SSD and HDD) possibly mingled with an internal one (RAM). We provide an analytical evaluation of the performance, followed by an experimental validation, and we state that using SSD or HDD is fully suited to practical cases, which are identified in the following sections.

2 Primer on Rainbow Tables

2.1 Mode of Operation

Let $h\colon A \to B$ be a one-way function, defined as being a function that is easy to compute but practically impossible to invert. Such a function is typically a cryptographic hash function. Given the image y in B of a value in A, the problem we want to solve is to find the preimage of y, i.e., x in A satisfying $h(x) = y$. The only way to solve the problem consists in picking values in A until the equation holds. This approach is called a *brute force* or an *exhaustive search* if the set A is exhaustively visited. The attack is practical if the set A is not too large. Providing a numerical upper bound is difficult because it depends on the running time of h, on the available processing ressources, and on the time that can be devoted to the attack. Roughly speaking, an academic team can today easily perform 2^{48} cryptographic operations during a single day, using a cluster of CPUs. Nonetheless, if the attack is expected to be repeated many times, e.g., to crack passwords, then restarting the computations from scratch every time is cumbersome. A TMTO consists in performing heavy precomputations once, to make the subsequent attacks less resource-consuming.

2.2 Precomputations

Building Tables. The objective of the precomputations is to build *tables*, which is done by computing *matrices* first. As depicted in Fig. 1, a matrix consists of a series of chains built by iterating alternatively h and *reduction* functions $r_i : B \to A$, such that r_i maps any point in B to an arbitrary point in A, in an efficient and uniformly distributed fashion. The *starting points* of the chains are

chosen arbitrarily and the chains are of fixed length t, which defines the *ending points*. This process stops when the number of chains with different ending points is deemed satisfactory. A *table* then consists of the first and last columns of the matrix, and the remaining intermediary values are discarded. A table is said *rainbow* if the reduction functions r_i, for $1 \leq i \leq t-1$, are all different, while a single reduction function used all along the table leads to classical Hellman-like tables. Rainbow tables are then usually filtered to remove duplicated ending points: such tables are called *clean* rainbow tables [2, 4] or *perfect rainbow tables* [6, 24]. Similarly, we have clean matrices.

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

Fig. 1. Matrix computed from m starting points.

Maximum Size. A table of *maximal size* is obtained when the starting points fully cover the set A. Given that the functions r_i are not injective, many chains collide, though, and the number of rows in a clean rainbow table is consequently much smaller than N. Let t be the chain length, then the *maximum* number of rows in a table is provided by Oechslin in [24]:

$$m_{\max} = \frac{2N}{t+1}. \tag{1}$$

Success Rate. The success rate of a clean rainbow table is the probability P for a random value in A to appear in the associated matrix:

$$P = 1 - \left(1 - \frac{m}{N}\right)^t \approx 1 - e^{\frac{mt}{N}}. \tag{2}$$

We observe that the maximum probability is obtained when $m = m_{\max}$, and the maximum probability consequently tends towards 86% when t tends to infinity. To increase this success rate, several tables can be used. For instance, for $\ell = 4$ tables, the success rate is greater than 99.9%.

2.3 Attack

Procedure. Given $y \in B$, the attack consists in retrieving $x \in A$, such that $h(x) = y$. To do so, a chain starting from y is computed and the check whether

the generated value matches one of the ending points from the table is performed at each iteration.

Given that the reduction functions are different in every column, the attack procedure is quadratic with respect to the chain length. It is also worth noting that the process is applied to the ℓ tables, and the optimal order of search is to go through each table at the same pace. This means that the right-most unvisited column of each table is explored, then the process is iterated until the left-most columns are reached.

Once a matching ending point is found, the corresponding starting point stored in the table is used to recompute the chain until reaching y. If the latter does not belong to the rebuilt chain, then we say that a *false alarm* occurred. False alarms exist because the reduction functions r_i are not injective. Then, the process goes on, until y is found in a column more on the left or the tables have been fully explored.

Evaluation. The analytic formula to evaluate the number of h-operations that are required on average to recover a preimage is given by Avoine, Oechslin, and Junod in [6]:

$$T \approx \gamma \frac{N^2}{M^2}, \tag{3}$$

where γ is a small factor that depends on $c = \frac{mt}{N}$ (the matrix stopping constant) and ℓ (the number of tables), and $M = m\ell$. See e.g. Theorem 2 in [20].

3 Performance of the Algorithms

3.1 Terminology and Assumptions

Rainbow tables can be stored in either internal memory (RAM) or external memory (e.g., SSD or HDD). An alternative is to use two complementary memories, e.g., RAM & SSD or RAM & HDD to benefit from the advantages of both of them.

The attack presented in Sect. 2, possibly combined with practical improvements, works on a single internal or external memory. It consists in performing direct lookups into the memory for matching ending points. We refer to it as $\mathsf{Algo}_{\mathsf{DLU}}$ (for Direct Look Up). The software Ophcrack [29] employs $\mathsf{Algo}_{\mathsf{DLU}}$ in the case when only RAM is used. Kim, Hong, and Park analyze in [18] another algorithm, hereafter denoted $\mathsf{Algo}_{\mathsf{STL}}$, that is used by RainbowCrack [27] and rcracki-mt [32]. Note that tables are generated the same way, regardless of the algirthm used for the attack. A same set of table can thus be used for $\mathsf{Algo}_{\mathsf{DLU}}$ or $\mathsf{Algo}_{\mathsf{STL}}$ interchangeably.

We describe below these algorithms and analyze their performance, taking both computation time and access time into account. In the rest of this paper, $\mathsf{Algo}_{\mathsf{DLU/RAM}}$, $\mathsf{Algo}_{\mathsf{DLU/SSD}}$, and $\mathsf{Algo}_{\mathsf{DLU/HDD}}$ refer to $\mathsf{Algo}_{\mathsf{DLU}}$ using respectively RAM, SSD, or HDD only. The same holds for $\mathsf{Algo}_{\mathsf{STL/SSD}}$ and $\mathsf{Algo}_{\mathsf{STL/HDD}}$

which refer to $\mathsf{Algo}_{\mathsf{STL}}$ using SSD and HDD respectively in addition to the RAM (RAM-only $\mathsf{Algo}_{\mathsf{STL}}$ is not meaningful).

The model used throughout this paper is the following. The attack (excluding the precomputation phase) is performed on a single computer with access to RAM and external memory (SSD or HDD). We denote τ_F the time (seconds) taken for the CPU to compute an iteration of h. For the external memory, we use two parameters – τ_S and τ_L – which revolve around the concept of *page* which is the smallest amount of external memory that can be addressed by the system. The seek time τ_S (seconds) corresponds to the time taken to read a single random page. The sequential read time τ_L (seconds) is the time to read a page during the read of many consecutive pages.

When the tables fit in RAM, the costs of a memory access and of an h-operation are of the same order of magnitude, i.e., a few hundred CPU cycles. However, the number of memory accesses grows linearly with the length of the table, while the number of h-computations is quadratic. Consequently, when the table is wide enough, the memory access time can be neglected (i.e., $\tau_S = \tau_L = 0$), and the attack time is equal to T multiplied by the cost of a single h-operation.

3.2 $\mathsf{Algo}_{\mathsf{DLU}}$

The algorithm $\mathsf{Algo}_{\mathsf{DLU}}$ is the attack described in Sect. 2.3. Its performance is provided in Theorem 1.

Theorem 1. $\mathsf{Algo}_{\mathsf{DLU}}$*'s average wall-clock time is* $T_{\mathsf{DLU}} = \gamma \frac{N^2}{M^2}\tau_F + \frac{N}{m}\log_2 m\tau_S$.

Proof. The first term of T_{DLU} is the portion of the time used by computations. The second term corresponds to the overhead of seeking data in the memory. As already stated in [6], the attack performs $\frac{N}{m}$ lookups on average in order to find a preimage. Each lookup requires $\log_2 m$ seeks in the memory if a dichotomic search is performed. Finally, each seek costs τ_S seconds on average.

We now look at the case where the memory that is used is RAM. In such a case, the algorithm $\mathsf{Algo}_{\mathsf{DLU/RAM}}$ is based on the assumption that the tables entirely reside in RAM, and no external memory is used. It takes advantage of fast RAM accesses given that we assume the RAM access time is negligible. In this case, the previous theorem is simplified and leads to Corollary 1.

Corollary 1. $\mathsf{Algo}_{\mathsf{DLU/RAM}}$*'s average wall-clock time is* $T_{\mathsf{RAM}} = \gamma \frac{N^2}{M^2}\tau_F$.

3.3 $\mathsf{Algo}_{\mathsf{STL}}$

The algorithm $\mathsf{Algo}_{\mathsf{STL}}$, described by Kim, Hong, and Park in [18], significantly differs from the other algorithm, mainly because the attack starts with computing all the t possible chains from the value y in B whose preimage is looked for. The tables are then loaded in RAM according to the following procedure. Given

the k-th table table_k ($1 \leq k \leq \ell$), containing m ordered pairs (starting point, ending point), a *sub-table* of table_k is a table that contains m/s ordered pairs belonging to table_k. In $\mathsf{Algo}_{\mathsf{STL}}$, tables are stored in an external memory (SSD or HDD) and each of them is partitioned into s non-overlapping sub-tables of a given size. Each of the s sub-tables are loaded into RAM, one at a time, which explains the acronym of the algorithm: Sub-Table Loading. For each sub-table loaded, the t possible chains are used to discover matching endpoints and discard false alarms, as it is done in $\mathsf{Algo}_{\mathsf{DLU/RAM}}$.

The efficiency of $\mathsf{Algo}_{\mathsf{STL}}$ is investigated in [18] and summarized in Theorem 2 and Theorem 3. The proofs are provided in [18].

Theorem 2. $\mathsf{Algo}_{\mathsf{STL}}$*'s average wall-clock time is*

$$T_{\mathsf{STL}} = L \cdot \tau_L + F \cdot \tau_F, \tag{4}$$

where $L = \frac{mP}{c\beta}, F = \delta\frac{N^2}{L^2}, \delta \approx \frac{P^3}{\beta^2}\left(\frac{1}{2(1-e^{-c})} + \frac{1}{6} - \frac{c}{48}\right)$*, and where* $c = \frac{mt}{N} < 2$*,* β *is the number of table entries (starting point – ending point pair) per page, and* $P = 1 - e^{-c\ell}$ *is the total probability of success.*

Theorem 3. *In optimal configuration, that is when the memory is of optimal size for a given problem,* $\mathsf{Algo}_{\mathsf{STL}}$*'s average wall-clock time is*

$$T^*_{\mathsf{STL}} = \frac{3}{2^{\frac{2}{3}}}\tau_L^{\frac{2}{3}}\tau_F^{\frac{1}{3}}\delta^{\frac{1}{3}}N^{\frac{2}{3}}, \tag{5}$$

and the memory associated to this situation corresponds to

$$m = \left(\beta\frac{\tau_F}{\tau_L}\right)^{\frac{1}{3}}\left(\frac{1}{1-e^{-c}} + \frac{1}{3} - \frac{c}{24}\right)^{\frac{1}{3}} cN^{\frac{2}{3}}.$$

Compared with [18], note that we changed the notations $\bar{\mathsf{R}}_{\mathsf{tc}}$ to δ, R_{tc} to γ, $\bar{c}$ to c, and $\bar{\mathsf{R}}_{\mathsf{ps}}$ to P, for consistency with the other algorithms and other notations in the literature. Also note that the definition of $\bar{\mathsf{R}}_{\mathsf{tc}}$ (δ) is inconsistent in [18] – sometimes multiplied by a factor of β^2. We chose to stick with the approximation used in Theorem 2, which corresponds to Proposition 5 in [18].

$\mathsf{Algo}_{\mathsf{STL}}$ has an optimal amount of memory at which it operates. This is because $T = O\left(m + \frac{1}{m^2}\right)$. Beyond a certain threshold, the decrease of the F factor fails to compensate for the increase of the L factor. This behavior is further commented on in Sect. 5.

The value s (number of sub-tables per table) is thoroughly discussed in [18]. If s is too small, sub-tables are very large, and when the search ends, it is likely that significant time was wasted loading the last sub-table. If s is too big, read operations are done on a small amount of memory, which is sub-optimal. As stated in [18] however, the value of s has relatively little impact on the efficiency of $\mathsf{Algo}_{\mathsf{STL}}$, provided it is "reasonable" (ranging from 45 to 100 in the examples discussed in [18]). In what follows, we assume such a reasonable s is used.

4 Alogrithm constants

The algorithms analyzed in this paper rely on the τ_S, τ_L and τ_F parameters heavily. These are machine-specific constants which can only be determined experimentally. We measured these values for the configuration used in our experimental validation of $\mathsf{Algo}_{\mathsf{DLU}}$, presented in Sect. 6.

4.1 Experimental Setup

The measurements have been done on a single machine with an Intel `E5-1603 v3` CPU clocked at 2.8 Ghz, and with 32GB of RAM available. It uses Intel SSD DC-3700 external memory with a capacity of 400GB, which is separated from the disk containing the operating system.

The Intel SSD use Non-Volatile Memory Express technology, so-called NVMe, which is an interface that provides smaller latencies, by connecting the SSD directly via PCI-Express to the processor and memory instead of going through an intermediate controller. This also allows for better stability in measurements.

4.2 Determination of Values for τ_S, τ_L and τ_F

The time measurements are made with the processor's internal time stamp counter, via the `RDTSC` instruction. This instruction is constant with respect to the power management re-clocking, and is synchronized across all cores on this CPU model. The processor does not have dynamic over-clocking, i.e., *Turbo Boost* capabilities, so the time stamp counter always increments 2.8 billion times per seconds. This allows for accurate measurements up to nanosecond precision.

Computation Time τ_F. We use the `MD5` hash function as the one-way function h. We assume that, during the execution of the TMTO, the CPU is warmed-up, i.e., it is running at its nominal frequency, which is expected in usual conditions. In this case, the time taken by successive applications of h is constant. We have estimated the time τ_F taken by a single application of h by averaging over the measurement of 10^6 applications of h, which gives $\tau_F = 1.786 \cdot 10^{-7}$s.

Sequential Block Read Time τ_L. In the context of external memory model with sub-tables loading, the constant τ_L refers to the time taken to read a page on disk during a sequential read of many blocks. The sub-tables are typically chosen to reach the maximal read throughput of the disk, with sizes in the order of the dozen or hundreds of megabytes. We note that, since disk have usually better performance in sequential reads than in random access, we should have $\tau_L \ll \tau_S$.

We measured the time to load 1000 arbitrary random data files, of size ranging from 10 to 500MB, in a RAM allocated array. We obtained $\tau_L = 4.59 \cdot 10^{-6}$s with a standard deviation $\sigma = 0.73 \cdot 10^{-6}$s. For reference, the same test on a 5400rpm HDD gave us $\tau_L = 20.99 \cdot 10^{-6}$s with $\sigma = 7.65 \cdot 10^{-6}$s.

Single Block Read Time τ_S. We measured the time taken by successive single-page reads of values at random positions in at least 1GB files. Each read has been measured separately. The value obtained, averaged over 500 measurements, is $\tau_S = 149.6 \cdot 10^{-6}$s, $\sigma = 20.7 \cdot 10^{-6}$s, which is indeed much larger than τ_L. On HDD, we obtained $\tau_S = 7.41 \cdot 10^{-3}$s and $\sigma = 3.79 \cdot 10^{-3}$s.

5 Analysis

This section compares the two algorithms described in Sect. 3 on different memory types and aims to characterize which of them has better performance depending on various parameters.

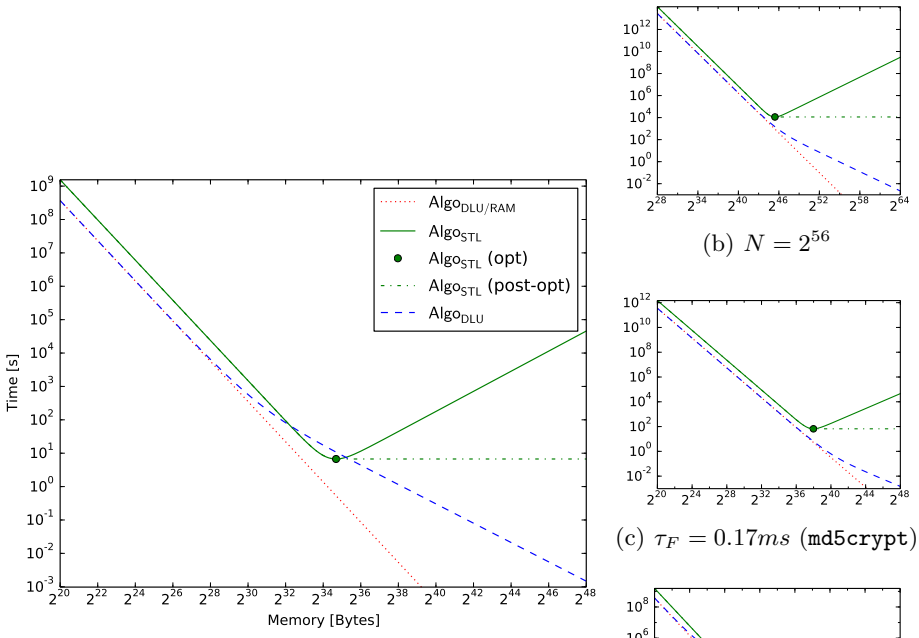
Analysis of $\mathsf{Algo}_{\mathsf{STL}}$ and comparison between $\mathsf{Algo}_{\mathsf{STL/HDD}}$ and $\mathsf{Algo}_{\mathsf{DLU/RAM}}$ was previously done in [18]. However, this comparison is limited in several ways. Most importantly, it only accounts for $\mathsf{Algo}_{\mathsf{STL}}$ in optimal configuration, that is with a fixed memory size. Furthermore, it only considers two data points of memory for $\mathsf{Algo}_{\mathsf{DLU/RAM}}$ and one for $\mathsf{Algo}_{\mathsf{STL/HDD}}$, and does not study $\mathsf{Algo}_{\mathsf{DLU/HDD}}$. The conclusion drawn in [18] is that $\mathsf{Algo}_{\mathsf{STL}}$ is superior for large problems, but the comparison is inconclusive for smaller problems.

In the analysis presented in the current paper, we overcome the aforementioned limitations and also study the case of the SSD memory. We base our comparison on the "Small Search Space Example" given in [18], on which $\mathsf{Algo}_{\mathsf{DLU/RAM}}$ and $\mathsf{Algo}_{\mathsf{STL/HDD}}$ have been compared. The problem space corresponds to passwords of length 7 over a 52-character alphabet (standard keyboard), which gives $N = 52^7 = 2^{39.903}$. The other parameters are $P = 0.999$, $\ell = 4$, $c = 1.7269$, $\delta = 0.73662/\beta^2$, $\gamma = 8.3915$, 16 bytes per chain ($\beta = 256$ for 4KB pages). For τ_F, τ_L, τ_S, we use values obtained experimentally – see Sect 4 for details on the methodology – instead of those given in [18]. The reason is that in [18], τ_S was not provided and the constants emanated from a different machine.

5.1 Comparing $\mathsf{Algo}_{\mathsf{STL}}$ and $\mathsf{Algo}_{\mathsf{DLU}}$

Fig. 2(a) presents the average wall-clock time for the three algorithms for varying amount of memory available when a SSD is used. Note that $\mathsf{Algo}_{\mathsf{DLU/RAM}}$ is presented at a somewhat unfair advantage since it uses RAM instead of external memory, and is only represented in Fig. 2(a) for completeness.

The main conclusions are the following: (1) The cost of $\mathsf{Algo}_{\mathsf{STL}}$ stops decreasing beyond a certain amount of memory available. This is due to the fact that the time taken for loading increasing amount of chains in RAM is not made up for by the decrease in computation. It is assumed that the optimal amount of external memory is used when possible, even when more is available. This also means that $\mathsf{Algo}_{\mathsf{STL}}$ has an inherent minimal average search time which can never be improved regardless of the memory available. $\mathsf{Algo}_{\mathsf{DLU}}$ has no such threshold. (2) The area (in terms of the external memory amount) where $\mathsf{Algo}_{\mathsf{STL}}$ is more efficient than $\mathsf{Algo}_{\mathsf{DLU}}$ is very small.



(a) Average wall-clock time, SSD, reference parameters

(b) $N = 2^{56}$

(c) $\tau_F = 0.17ms$ (`md5crypt`)

(d) HDD

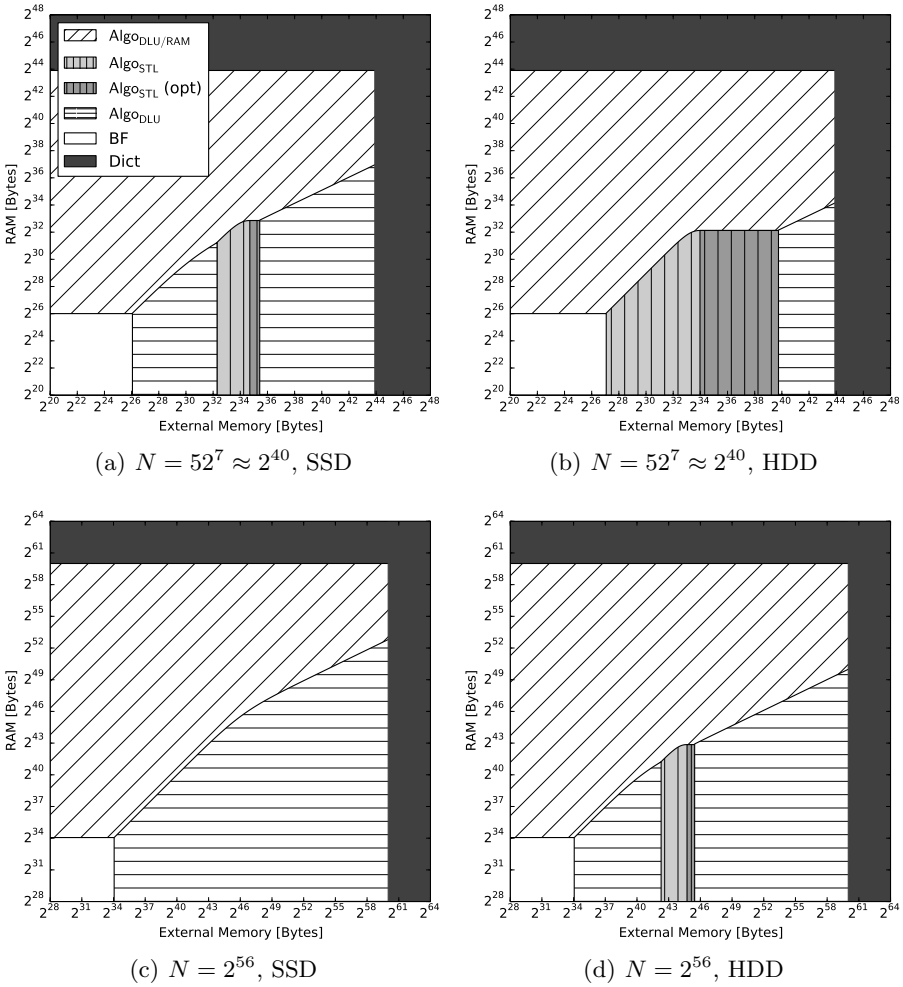
Fig. 2. Average wall-clock time, depending on the memory available.

We can further elaborate that the curves for T_{STL} and T_{DLU} do not intersect if $\mathsf{Algo}_{\mathsf{DLU}}$ is always more efficient, otherwise they intersect at two points (possibly only one point in a degenerated case). Whether there are 0 or 2 (and the positions of these) intersections depends on problem parameters (N, M, c, ℓ) and machine parameters (β, τ_F, τ_L, τ_S). To illustrate this, Fig. 2 shows the effect of changing some parameters, with all other parameters left untouched.

- Fig. 2(b): When $N = 2^{56}$ the algorithm $\mathsf{Algo}_{\mathsf{DLU}}$ is superior throughout.
- Fig. 2(c): The `md5crypt` function is 955 times slower than `MD5`. When used, it gives $\tau_F = 0.171ms$, and $\mathsf{Algo}_{\mathsf{DLU}}$ is again superior throughout.
- Fig. 2(d): Using a hard disk drive instead of SSD implies $\tau_S = 7.41$ms and $\tau_L = 20.99\mu$s. Expectedly, $\mathsf{Algo}_{\mathsf{DLU}}$ suffers from longer seek time, and $\mathsf{Algo}_{\mathsf{STL}}$ dominates on a larger area (but not globally).

5.2 Comparison with RAM

Fig. 3 presents regions, in terms of RAM and external memory available, in which each algorithm, completed with naive online brute-force and dictionary methods, is the most efficient. Formulas for average wall-clock time described in Sect. 3 were used for $\mathsf{Algo}_{\mathsf{STL}}$ and $\mathsf{Algo}_{\mathsf{DLU}}$. An average of $\frac{N}{2}\tau_F$ is used for online brute-force, and the dictionary method is assumed to dominate as long as it has sufficient memory available, i.e., $16N$ bytes (`MD5` hashes are 16 bytes).



(a) $N = 52^7 \approx 2^{40}$, SSD

(b) $N = 52^7 \approx 2^{40}$, HDD

(c) $N = 2^{56}$, SSD

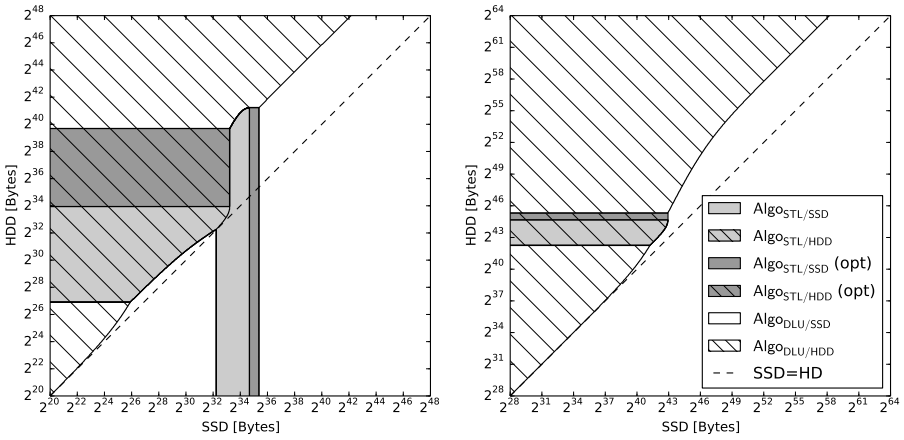
(d) $N = 2^{56}$, HDD

Fig. 3. Regions, in terms of RAM and external memory, where each algorithm has minimum time, in four different scenarios.

It is difficult to conclude unequivocally on $\mathsf{Algo}_{\mathsf{STL}}$ and $\mathsf{Algo}_{\mathsf{DLU}}$, as their respective performances highly depend on parameters. It can however be observed that (1) $\mathsf{Algo}_{\mathsf{DLU}}$ typically outperforms $\mathsf{Algo}_{\mathsf{STL}}$ on large problems, and when h is expensive; and (2) the seek time is of crucial importance for $\mathsf{Algo}_{\mathsf{DLU}}$, which performs poorly compared to $\mathsf{Algo}_{\mathsf{STL}}$ on devices (such as hard disk) with slow seek time but high sequential read performance.

5.3 HDD and SSD

Fig. 4 compares the performances of $\mathsf{Algo}_{\mathsf{DLU}}$ and $\mathsf{Algo}_{\mathsf{STL}}$ on SSD and HDD. The dashed line represents points where SSD memory and HDD memory are equal. Nowadays, HDD memory is cheaper than SSD, which corresponds to the region above this line.



(a) $N = 52^7 \approx 2^{40}$ (b) $N = 2^{56}$

Fig. 4. Regions, in terms of SSD and HDD memory, where each algorithm has minimum time.

5.4 Discussion

Conclusion of the Comparisons. The various comparisons done in this section show that many parameters influence the choice of the algorithm and memory type to use, and it is difficult to make a simple judgment as to which is best. These parameters include problem parameters (N, M, c, ℓ) and machine/technology parameters $(\beta, \tau_F, \tau_L, \tau_S)$. A few observation can be made however.

- $\mathsf{Algo}_{\mathsf{DLU}}$ performs better on larger problem spaces than $\mathsf{Algo}_{\mathsf{STL}}$.
- $\mathsf{Algo}_{\mathsf{DLU}}$ performs better on slower hash functions than $\mathsf{Algo}_{\mathsf{STL}}$.

– $\mathsf{Algo}_{\mathsf{STL}}$ handles better than $\mathsf{Algo}_{\mathsf{DLU}}$ the use of slower memories such as HDD.
– In many scenarios, a large portion of where $\mathsf{Algo}_{\mathsf{STL}}$ is most appropriate is *not* in its optimal configuration.
– In some scenarios, the region where $\mathsf{Algo}_{\mathsf{STL}}$ is most appropriate is also close to the typical memory size.

Limits of the Analysis. First of all, our results and observations are based on the measures given in Sect 4. Using a particularly fast or slow HDD, for instance, might influence the results in a non-negligible way. Likewise, using clusters of many disks to reach high quantities of memory might affect τ_S and τ_L enough that the conclusions would be different.

Furthermore, the analysis is based on Sect. 3, and does not consider optimizations such as checkpoints [1,5], endpoint truncation [1,20], and prefix/suffix or compressed delta encoding for chain storage [2]. Likewise, it does not consider optimizations exploiting the architecture, such as loading sub-tables while computing chains in $\mathsf{Algo}_{\mathsf{STL}}$.

Including these optimizations would make the analysis much more complex, and we believe that taking them into consideration would not change our conclusions. While some optimizations might favor one algorithm more than the other, it is very unlikely that the frontiers between regions of best performance would shift significantly.

6 Experimentation

We have set up experiments in order to validate the analytical results described in Sect. 3 and Sect. 5. The formulas established in Sect. 3 assume that $\mathsf{Algo}_{\mathsf{DLU/SSD}}$ and $\mathsf{Algo}_{\mathsf{DLU/HDD}}$ do not use RAM at all. We show that, in reality, these algorithms actually do use RAM because operating systems use cache techniques to speed up lookups. Thus, a value read from an external memory is temporarily saved in cache to prevent (to some extent) from accessing the external memory again to get the same value. As a consequence, the results provided in Sect. 3 correspond to upper bounds in practice. We refine the formulas to take the caching effect into account, and we then show that the refined formulas describe more accurately the experimental results.

6.1 Parameters and Methodology

We have conducted the experiments on two problems of size $N = 2^{31}$ and 2^{36}, using the `MD5` hash function for a number of columns $t \in \{100, 200, \ldots, 900\}$. The size of the problems allowed us to precompute the matrices in a reasonable time frame. For the 2^{36}-problem, the precomputation of the full matrix took 5 hours on 400 processor cores. Sorting the ending points and removing the duplicated ones required a couple of days due to the network latencies.

For each problem, $\ell = 4$ tables were computed with a matrix stopping constant of $c = 1.92$ ($m = 0.96 m_{\max}$), giving $P \approx 0.999$. Each (starting point, ending point) pair is stored on 8 and 16 bytes for $N = 2^{31}$ and $N = 2^{36}$ respectively. The tables are clean and ordered with respect to the ending points.

To evaluate the average running time of $\mathsf{Algo}_{\mathsf{DLU}}$, we average the measured attack time for the hashes of 1000 randomly-generated values in the problem space. The timings were based on the processor timestamp counter (`RDTSC`). In order to keep the experiments independent, the environment is reset before each tests. Indeed there are side effects to be expected due to the way the operating system handles files, some of which also affect the measurements themselves. We discuss them in the subsequent sections.

6.2 Paging and Caching Mechanisms

For every access to data stored on external memory, the full page containing the data is actually returned to the operating system. Since the 60s, the *paging mechanism* is based on 4KB pages in most operating systems and external memories [31].

Due to the *caching mechanism*, the data is not fetched directly from the external memory every time we perform a lookup. Instead, the page containing the data is first copied from the external memory to the internal memory, and only then it can be read. Such a mechanism allows the system to speed up memory accesses: as long as the internal memory is not reclaimed for another use, the content of the page remains in it. This means that, if the same page is accessed again, it can be retrieved directly from the internal memory instead of waiting for the external memory.

If several lookups are performed on values that are located close enough in the external memory, then only the earliest lookup will require accessing the external memory. This phenomenon happens when a lookup is performed in the dichotomic search. As a consequence, at some point, every external memory access fetches elements that are located in the same page. Taking paging and caching mechanisms into account, Theorem 1 can be refined to yield Theorem 4.

Theorem 4. *Given β (starting points, ending points) pairs per page. Taking the paging and caching mechanisms into account,* $\mathsf{Algo}_{\mathsf{DLU}}$*'s average wall-clock time is*

$$T_{\mathsf{DLU}} = \gamma \frac{N^2}{M^2} \tau_F + \frac{N}{m} (\log_2 m - \log_2 \beta) \cdot \tau_S. \tag{6}$$

Proof. The loading of pages instead of single values corresponds to a dichotomic search tree that is $\log_2 \beta$ levels shallower. Thus, each lookup consists in $\log_2 m - \log_2 \beta$ page loads instead of $\log_2 m$.

6.3 Reducing the Caching Impact

To get proper experimental results and bypass the operating system's built-in caching mechanism, we use a cache eviction technique and restrain the internal memory allocated to the program.

Every page that remains in memory after each experiment needs to be reset to a blank state. We use the `madvise` system call to tell the kernel that no additional pages are required. Although the kernel could ignore the setting and keep the pages in memory anyway, it did not seem to happen in our experiments. Alternative methods exist, such as requesting a cache drop, but they might affect the experiments by wiping data needed by the operating system.

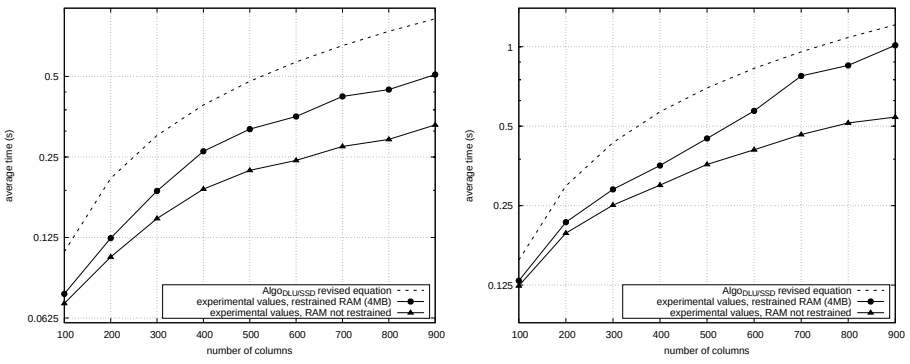
We also restrain the internal memory that the program can use to a few megabytes, using the `cgroup` kernel subsystem. Software limitation was used instead of physically limiting the internal memory because physical limitation may cause the operating system to starve for memory, which would greatly affect the results.

6.4 Experimental Results

As expected, our implementation of $\mathsf{Algo}_{\mathsf{DLU/RAM}}$, which mainly depends on τ_F, follows closely the curve given by Corollary 1.

The experimental results concerning $\mathsf{Algo}_{\mathsf{DLU}}$ are presented in Fig. 5. The dashed curves are computed from the revised formula for $\mathsf{Algo}_{\mathsf{DLU}}$, provided by Eq. (6). For each problem, we give the timings when the RAM is restrained (line with dots) and when it is not (line with triangles).

Some caching can still be noticed on the curves, but trying to restrain RAM even further resulted in failures which could be explained by the fact that there is no distinction for the OS between file caching and the caching of the executable itself. Thus, we have to overprovision the RAM to be certain that the program itself does not starve during the execution. Nevertheless, we observe that the experimental curve is below the analytic dashed-line curve. This confirms that Eq. (6) is a more accurate upper bound to the practical wall-clock time of $\mathsf{Algo}_{\mathsf{DLU}}$ than the formula of Theorem 1.



(a) Problem $N = 2^{31}$ (b) Problem $N = 2^{36}$

Fig. 5. TMTO attack average online running time per column, on 1000 hashed values

Finally, we remark that even though we have restrained RAM drastically, we can notice the gain in time due to the caching done by the OS. The zones concerning $\mathsf{Algo}_{\mathsf{DLU}}$ in Fig. 2 would therefore have smaller areas, in practice. The impact of the native caching operated by the OS is difficult to predict with exact precision, though.

7 Conclusion

In this paper, we have studied the use of external memory for time-memory trade-offs based on rainbow tables. The use of external memory is motivated by large problems for which precomputed tables are too big to fit in RAM. Two approaches were compared: the first one relying on the classical $\mathsf{Algo}_{\mathsf{STL}}$ algorithm which takes advantage of RAM by processing sub-tables in internal memory; and the second, based on the $\mathsf{Algo}_{\mathsf{DLU}}$ algorithm, which uses the standard RAM-based rainbow table algorithm directly on external memory (we are not aware of public implementations or analyses of $\mathsf{Algo}_{\mathsf{DLU}}$ on external memory).

We evaluate the two algorithms and compare their efficiency on different memory types and different problem parameters. Conclusions are subject to parameters, but several major observations are made. $\mathsf{Algo}_{\mathsf{DLU}}$ performs better than $\mathsf{Algo}_{\mathsf{STL}}$ on larger problem spaces, slower hash functions, and faster memories. At the very least, it is not the case that $\mathsf{Algo}_{\mathsf{STL}}$ is unequivocally more efficient on larger problems, contrarily to what was previously thought.

Costs of the various memory types were not considered formally in our analysis because of the great variability in prices and setups. Very roughly speaking, we can observe that nowadays, a gigabyte of RAM costs between 5 and 20 times more than a gigabyte of SSD, which itself costs between 3 and 8 times more than a gigabyte of HDD. Such prices and comparisons might help an implementer make an educated decision regarding the memory type to use.

We implemented $\mathsf{Algo}_{\mathsf{DLU}}$ and validated its efficiency analysis (analysis and validation of $\mathsf{Algo}_{\mathsf{STL}}$ was previously done in [18]). It shows that the analysis is close, but in fact pessimistic due to caching in RAM. Exploiting the RAM in addition to external memory might give an extra edge to $\mathsf{Algo}_{\mathsf{DLU}}$. Optimizations on $\mathsf{Algo}_{\mathsf{STL}}$ might exist as well, such as computing chains and loading tables in parallel. Analysis of such optimizations, along with other algorithmic optimizations on rainbow tables and other trade-off variants might be an interesting continuation of this work.

Acknowledgments. This work has been partly supported by the COST Action IC1403 (Cryptacus). Xavier Carpent was supported, in part, by a fellowship of the Belgian American Educational Foundation.

References

1. Avoine, G., Bourgeois, A., Carpent, X.: Analysis of rainbow tables with fingerprints. In: Foo, E., Stebila, D. (eds.) Australasian Conference on Information Se-

curity and Privacy – ACISP 2015. pp. 356–374. Lecture Notes in Computer Science, Springer, Brisbane, Australia (June 2015)
2. Avoine, G., Carpent, X.: Optimal storage for rainbow tables. In: Kim, S., Kang, S.Y. (eds.) International Conference on Information Security and Cryptology – ICISC 2013. Lecture Notes in Computer Science, vol. 8565, pp. 144–157. Springer, Seoul, South Korea (November 2013)
3. Avoine, G., Carpent, X.: Heterogeneous rainbow table widths provide faster cryptanalyses. In: ACM Asia Conference on Computer and Communications Security – ASIACCS 2017. pp. 815–822. ASIA CCS '17, ACM, Abu Dhabi, UAE (April 2017)
4. Avoine, G., Carpent, X., Lauradoux, C.: Interleaving cryptanalytic time-memory trade-offs on non-uniform distributions. In: Pernul, G., Ryan, P.Y.A., Weippl, E. (eds.) European Symposium on Research in Computer Security – ESORICS 2015. Lecture Notes in Computer Science, vol. 9326, pp. 165–184. Springer, Vienna, Austria (September 2015)
5. Avoine, G., Junod, P., Oechslin, P.: Time-memory trade-offs: False alarm detection using checkpoints. In: Maitra, S., Veni Madhavan, C. E.and Venkatesan, R. (eds.) Progress in Cryptology – INDOCRYPT'05. Lecture Notes in Computer Science, vol. 3797, pp. 183–196. Cryptology Research Society of India, Springer, Bangalore, India (December 2005)
6. Avoine, G., Junod, P., Oechslin, P.: Characterization and improvement of time-memory trade-off based on perfect tables. ACM Transactions on Information and System Security 11(4), 17:1–17:22 (July 2008)
7. Barkan, E.P.: Cryptanalysis of Ciphers and Protocols. Ph.D. thesis, Technion – Israel Institute of Technology, Haifa, Israel (March 2006)
8. Biryukov, A., Mukhopadhyay, S., Sarkar, P.: Improved time-memory trade-offs with multiple data. In: Preneel, B., Tavares, S. (eds.) Selected Areas in Cryptography – SAC 2005. Lecture Notes in Computer Science, vol. 3897, pp. 110–127. SAC Board, Springer, Kingston, Canada (August 2005)
9. Biryukov, A., Shamir, A., Wagner, D.: Real time cryptanalysis of A5/1 on a PC. In: Schneier, B. (ed.) Fast Software Encryption – FSE'00. Lecture Notes in Computer Science, vol. 1978, pp. 1–18. Springer, New York, USA (April 2000)
10. Bitweasil: Cryptohaze (2012), `http://cryptohaze.com/`, accessed: 2017-04-19
11. Bono, S., Green, M., Stubblefield, A., Juels, A., Rubin, A., Szydlo, M.: Security Analysis of a Cryptographically-Enabled RFID Device. In: USENIX Security Symposium – USENIX'05. pp. 1–16. USENIX, Baltimore, Maryland, USA (July–August 2005)
12. Denning, D.E.: Cryptography and Data Security, p. 100. Addison-Wesley, Boston, Massachusetts, USA (June 1982)
13. Dunkelman, O., Keller, N.: Treatment of the initial value in time-memory-data tradeoff attacks on stream ciphers. Information Processing Letters 107(5), 133–137 (August 2008)
14. Hellman, M.: A cryptanalytic time-memory trade off. IEEE Transactions on Information Theory IT-26(4), 401–406 (July 1980)
15. Hoch, Y.Z.: Security Analysis of Generic Iterated Hash Functions. Ph.D. thesis, Weizmann Institute of Science, Rehovot, Israel (August 2009)
16. Hong, J., Jeong, K., Kwon, E., Lee, I.S., Ma, D.: Variants of the distinguished point method for cryptanalytic time memory trade-offs. In: Chen, L., Mu, Y., Susilo, W. (eds.) Information Security Practice and Experience. Lecture Notes in Computer Science, vol. 4991, pp. 131–145. Springer, Sydney, Australia (April 2008)

17. Hong, J., Sarkar, P.: New applications of time memory data tradeoffs. In: Roy, B. (ed.) Advances in Cryptology – ASIACRYPT'05. Lecture Notes in Computer Science, vol. 3788, pp. 353–372. IACR, Springer, Chennai, India (December 2005)
18. Kim, J.W., Hong, J., Park, K.: Analysis of the rainbow tradeoff algorithm used in practice. IACR Cryptology ePrint Archive (2013)
19. Kim, J.W., Seo, J., Hong, J., Park, K., Kim, S.R.: High-speed parallel implementations of the rainbow method in a heterogeneous system. In: Galbraith, S.D., Nandi, M. (eds.) Progress in Cryptology – INDOCRYPT 2012. Lecture Notes in Computer Science, vol. 7668, pp. 303–316. IACR, Springer, Kolkata, India (December 2012)
20. Lee, G.W., Hong, J.: Comparison of perfect table cryptanalytic tradeoff algorithms. Designs, Codes and Cryptography 80(3), 473–523 (September 2016)
21. Lu, J., Li, Z., Henricksen, M.: Time-memory trade-off attack on the GSM A5/1 stream cipher using commodity GPGPU - (extended abstract). In: Malkin, T., Kolesnikov, V., Bishop Lewko, A., Polychronakis, M. (eds.) Applied Cryptography and Network Security - 13th International Conference – ACNS 2015. pp. 350–369 (June 2015)
22. Mentens, N., Batina, L., Preneel, B., Verbauwhede, I.: Cracking Unix passwords using FPGA platforms. SHARCS - Special Purpose Hardware for Attacking Cryptographic Systems (February 2005)
23. Mentens, N., Batina, L., Preneel, B., Verbauwhede, I.: Time-memory trade-off attack on FPGA platforms: UNIX password cracking. In: Bertels, K., Cardoso, J.M.P., Vassiliadis, S. (eds.) Reconfigurable Computing: Architectures and Applications, ARC 2006. Lecture Notes in Computer Science, vol. 3985, pp. 323–334. Springer, Delft, The Netherlands (March 2006)
24. Oechslin, P.: Making a faster cryptanalytic time-memory trade-off. In: Boneh, D. (ed.) Advances in Cryptology – CRYPTO 2003. Lecture Notes in Computer Science, vol. 2729, pp. 617–630. IACR, Springer, Santa Barbara, California, USA (August 2003)
25. Quisquater, J., Standaert, F., Rouvroy, G., David, J., Legat, J.: A cryptanalytic time-memory tradeoff: First FPGA implementation. In: Glesner, M., Zipf, P., Renovell, M. (eds.) Field-Programmable Logic and Applications, 12th International Conference – FPL 2002. Lecture Notes in Computer Science, vol. 2438, pp. 780–789. Springer, Montpellier, France (September 2002)
26. Saarinen, M.J.O.: A time-memory tradeoff attack against LILI-128. In: Fast Software Encryption – FSE'01. vol. 2365, pp. 231–236. Leuven, Belgium (February 2001)
27. Shuanglei, Z.: Rainbowcrack (2017), `http://project-rainbowcrack.com/`, accessed: 2017-04-19
28. Spitz, S.: Time Memory Tradeoff Implementation on Copacobana. Master's thesis, Ruhr-Universität Bochum, Bochum, Germany (June 2007)
29. Tissières, C., Oechslin, P.: Ophcrack (2016), `http://ophcrack.sourceforge.net/`, accessed: 2017-04-19
30. Verdult, R., Garcia, F.D., Ege, B.: Dismantling megamos crypto: Wirelessly lockpicking a vehicle immobilizer. In: Proceedings of the 22nd USENIX Security Symposium – USENIX'13. pp. 703–718. Washington, DC, USA (August 2013)
31. Weisberg, P., Wiseman, Y.: Using 4KB page size for virtual memory is obsolete. In: Proceedings of the IEEE International Conference on Information Reuse and Integration – IRI 2009. pp. 262–265. Las Vegas, Nevada, USA (August 2009)
32. Westergaard, M., Nobis, J., Shuanglei, Z.: Rcracki-mt (2014), `http://tools.kali.org/password-attacks/rcracki-mt`, accessed: 2017-04-19