Chapter 5

Cryptanalytic Time-Memory Tradeoff for Password Hashing Schemes

In previous chapters, we discussed the designing aspects of password hashing schemes (PHS). This chapter is dedicated to the analysis of PHS. A cryptanalytic technique known as time-memory tradeoff (TMTO) was proposed by Hellman for finding the secret key of a block cipher. This technique allows sharing the effort of key search between the two extremes of exhaustively enumerating all keys versus listing all possible ciphertext mappings produced by a given plaintext (i.e. table lookups). The TMTO technique has also been used as an effective cryptanalytic approach for PHS. The increasing threat of password leakage from compromised password hashes demands a resource consuming algorithm to prevent the pre-computation of the password hashes. A class of password hashing designs provides such a defense against time-memory tradeoff (TMTO) attack by ensuring that any reduction in the memory leads to exponential increase in runtime. These are called Memory hard designs and the security of such designs are defined by measuring their TMTO defense. Another important measure for a good PHS is its efficiency in utilizing all the available memory as quickly as possible, and fast running time when more than the required memory is available.

The Memory Hardness requirement for password hashing schemes is significant to prevent the dictionary attack. Usually, attackers use GPU clusters, FPGAs and ASICs to get tremendous amounts of computation power to brute-force frequently used passwords when a general purpose cryptographic hash function like SHA-1, SHA-2, BLAKE etc. is used. These constructions are extremely fast in hardware as well as software implementations thus enabling an attacker to perform billions of hashes per second. To prevent such attempt, memory hard designs are a good alternative. However, it is generally difficult to evaluate the “memory hardness” of a given password hashing design following its algorithmic description.

In this chapter, we present a simple technique to analyze the memory hardness of those algorithms whose execution can be expressed as a Directed Acyclic Graph (DAG). The nodes of the DAG correspond to the storage required by the algorithm and the edges correspond to the flow of the execution. Our proposed technique provides expected run-times at varied levels of available storage for the DAG. Most of the algorithms submitted to the PHC have
multiple variants and each can be represented as a different DAG. We experimentally show the TMTO of the algorithms representing the DAG with specific parameters. Specifically, we give a generic algorithm to traverse the DAG which allows to vary the memory storage and computes the increased algorithmic runtime (re-computation penalties) for different tradeoffs (varying memory and time) options. We show the effectiveness of our proposed technique by applying it to three designs from the “Password Hashing Competition” (PHC) - Argon2-Version 1.2.1 (the PHC winner), Catena-Version 3.2 and Rig-Version 2. Our analysis shows that Argon2i shows weak memory hardness which is also highlighted in a recent work by Corrigan-Gibbs et al. For Argon2i, the DAG representation varies depending on the output of a pseudorandom function which is non-uniform. Therefore, different DAGs for different input values are obtained. Consequently, the choice of nodes which should be kept in memory becomes probabilistic and hence difficult to analyze. Therefore, we first apply heuristic methods to find the optimal points for memory reduction and then apply the proposed traversal algorithm to obtain the TMTO results.

We analyze these PHS for performance under various settings of time and memory complexities. Our experimental results show (i) simple DAGs for PHS are efficient but not memory hard, (ii) complex DAGs for PHS are memory hard but less efficient, and (iii) combination of two simple graphs in the representation of a DAG for PHS achieves both memory hardness and efficiency.

This chapter is organized as follows: We first discuss the existing work on time-memory tradeoff analysis in Section 5.1. Next, the brief description on existing analysis of memory hard functions using graph pebbling is included in Section 5.2. We then present a brief overview of three cache-timing resistant password hashing algorithms namely, Argon2i, Catena and Rig in Section 5.3. This is followed by the preliminaries necessary for the understanding of the proposed time-memory tradeoff analysis technique in Section 5.4. The description of the proposed algorithm for TMTO is presented in Section 5.5. Subsequently, the re-computation penalties and performance analysis are presented in Section 5.6 and Section 5.7 respectively. Finally in Section 5.8 we summarize the whole chapter. The original contribution of this thesis is from Section 5.3 to Section 5.7.

5.1 Existing Results on Cryptanalytic Time-Memory Tradeoff

The idea of Time-Memory Tradeoff (TMTO) to optimize the cryptanalytic effort for a search which includes \( N \) possible solutions was introduced by Hellman in [97]. Specifically for TMTO, the cryptanalyst tries to optimize the product \( T \cdot M \) searching for a correct solution among \( N \) feasible choices where \( T \) is the number of operations performed (time), \( M \) is the number of words of memory and \( T \cdot M = N \). The relative cost of CPU cycles (\( T \)) is much lesser than RAM space (\( M \)), as a result most attacks attempt to reduce memory at the cost of increased algorithmic runtime.

The proposed technique of [97] for TMTO analysis considers chosen plaintext attack
scenario to find the key of a block cipher. Specifically, \( m \) uniformly random starting points are chosen and then \( m \) chains of length \( t \) of ciphertexts are computed where only the starting and ending points of the chains are stored that yields the tradeoff (see Chapter 2 for more details).

Utilizing the concept of Hellman \[97\], Philippe Oechslin introduced a cryptanalytic time-memory trade-off, named rainbow table \[130\]. This technique allows attacking a password hashing scheme by reducing the cryptanalysis time with the help of precomputed data stored in memory. It creates chains of password hashes choosing \( m \) uniformly random passwords from provided keyspace. It generates \( m \) chains of length \( t \) using \( t \) different reduction functions. The main advantage of rainbow table over the technique of Hellman \[97\] is that the chains can collide within the same table without merging (to merge, collision at the same position is required for two different chains). This is possible because rainbow table uses \( t \) different reduction functions to compute the chains. However, this technique is only applicable on password hashing schemes that do not consider salt as an input with the password (refer Chapter 2 for more details). A recent result on TMTO analysis based on precomputation method is covered in \[46\] for data-independent password hashing schemes and also provides a generic ranking method for data-dependent schemes. This proposed method is applied in \[45\] to prove the memory hardness of Argon2i. However, our analysis in Section 5.5 shows that Argon2i does not provide the required security level. Infact an attack showing the weak memory hardness of Argon2i was recently shown by Corrigan-Gibbs et al. \[98\].

5.2 On the Analysis of Memory-Hardness using Graph Pebbling

This section presents the existing analysis of password hashing schemes which is based on graph pebbling concepts (another significant area of research). We only provide the significant results in brief. The graph pebbling technique helps to prove the memory hardness property of password hashing schemes and recently proposed memory-hard designs of PHC \[16\] are analysed through graph pebbling technique implemented on their respective DAG representation.

Pebbling helps to determine the efficient way to use the registers of a CPU and/or the random-access memory of a general-purpose computer. In practice, we frequently come across the situation where the number of registers (space) available is insufficient to hold all the data on which a program operates and therefore registers needs to be reused. If the space is increased, the number of computation steps (time) can generally be reduced \[142\]. The tradeoffs between the number of storage locations and computation time are analysed by the pebble game. In theoretical computer science, the concept of pebbling is described with a game known as pebble game or graph pebbling. For a DAG, pebbling represents the operation of assigning space to a vertex of the graph. Considering the constraint on the usage of space by the vertices of the DAG, the pebble game helps to determine a time-memory tradeoff for the given algorithm by pebbling a predetermined vertex within the
corresponding DAG. The initial assumption for the game is that the DAG has no pebble. The player performs a number of certain actions until a predefined output vertex has been pebbled. Considering the execution of an algorithm which can be represented as a DAG, a pebble on a vertex of the DAG indicates that its value is in a register.

5.2.1 Pebble Game

The goal of the pebble game is to pebble the vertices of the graph with a number of pebbles (space) and steps (time) that are minimal, i.e., neither can be reduced without increasing the other. Hence, it examines tradeoffs between the number of storage locations (memory/space) and computation time of an algorithm which can be modeled by a Directed Acyclic Graph. Specifically, the technique helps to derive lower bounds on the exchange of space $S$ for time $T$ and commonly evaluated in the form of product $S.T$ on the input of size $N$ \[^{[132]}\]. Following are the rules of the game.

**Rules of the pebble game:**

- A free pebble can be placed on an input vertex at any time.
- A vertex can be pebbled only if all its predecessors are pebbled.
- A pebble can be removed at any time.
- Each output vertex must be pebbled at least once.

A pebbling strategy is the execution of the rules of the pebble game on the vertices of a DAG. The placement of a pebble on an input vertex represents the reading of input data and on a non-input vertex corresponds to computing the value associated with the vertex. The removal of a pebble represents the erasure or overwriting of the value associated with the vertex on which the pebble resides. When pebbles are allowed to be placed on input vertices at any time, it reflects that the inputs are readily available.

As already explained, the objective of the game is to successively pebble each vertex while minimizing the number of pebbles that are ever on the graph simultaneously. Following are the important terminologies and notations associated with the pebbling technique for a DAG.

- The size is the number of nodes $N$.
- The depth is the length of its longest path.
- The space $S$ denotes the number of pebbles, and time $T$ is the number of moves.
- $ST$-cost is the product of the maximum no. of simultaneously used pebbles and the number of steps needed to complete the pebbling.
5.2.2 Important Results

In this section we list the important conjectures and theorems which play a significant role in the understanding of research in the analysis of password hashing schemes. The following conjecture is from [118] which answers the question on how much $T$ increases as $S$ decreases from $N$ toward $\Omega \left(\frac{N}{\log N}\right)$. It is important to note that in theoretical computer science a polynomial-time algorithms are considered as feasible, whereas algorithms whose time complexity exceeds every polynomial are not. In this sense the conjecture asserts that there are graphs for which space savings of $S = O\left(\frac{N}{\log N}\right)$, though possible, are infeasible.

**Tradeoff Conjecture** [118] There are graphs of size $N$ that can be pebbled with $O\left(\frac{N}{\log N}\right)$ pebbles only in a time that grows superpolynomially in $N$.

Graph families whose pebbling time increases explosively from linear to superpolynomial are of special interest in the literature. Among the graphs, the bit-reversal permutation graph was studied extensively. The bit-reversal permutation is among the permutations that are most difficult to realize in serial computation schemes with restricted storage capacity. It has the property that it scatters adjacent numbers approximately evenly over the interval. A bit-reversal permutation graph with 8-vertices at each level where the total level is 2 is shown in Fig. 5.1 which is also explained in Section 5.4. The following theorem proves the time-memory tradeoff achieved for pebbling the bit-reversal permutation graph as presented in [142].

**Theorem 5.2.1** If $S \geq 2$, then pebbling the bit-reversal graph on $N$ elements with $S$ pebbles takes time $T > \frac{N^2}{16S}$

**Pebbling in Parallel Setting** [31] The standard pebbling game is defined over the sequential computational model. Motivated through the parallelism in the modern computational system, the pebbling paradigm to a parallel setting is introduced in 2015 [31].
modifies the rules of the standard pebbling game to obtain a parallelised generalization. It allows the rules to be applied batch-wise instead of at most one pebble movement per step of the sequential model. Hence, a new cost measure is introduced which is called the cumulative complexity (cc) of a graph. The cc of a given execution in the (parallel) pebbling game for graph $G$ is the sum of the number of pebbles lying on $G$ when summed across all steps in the execution.

- **Cumulative Pebbling Complexity:** Let $G$ be a DAG, $P = (P_0, \cdots, P_t)$ be an arbitrary pebbling of $G$ and $\Pi$ be the set of all complete pebblings of $G$. Then the (cumulative) cost of $P$ and then cumulative complexity (cc) of $G$ are defined respectively to be:

$$
 pcost(P) := \sum_{i=0}^{t} |P_i|
$$

$$
 cc(G) := \min\{pcost(P) : P \in \Pi\}
$$

**Pebbling of Bit-Reversal Graph in Parallel Setting** In sequential setting it is shown in Theorem 5.2.1 that the tradeoff complexity of the bit-reversal graph is $ST = O(N^2)$. However, in the parallel setting, the cumulative complexity is shown over the same graph as $O(N^{1.5})$ in [31]. Instead of taking a bit-reversal graph, the result is shown in a generalized graph called a sandwich graph. A sandwich graph is a chain of $N$ nodes (numbered 1 through $N$) with arbitrary additional edges connecting nodes from the first half of the chain with nodes of the second half of the chain such that no node has in-degree greater than 2. Bit-Reversal graph is a special case of Sandwich Graph.

**Lemma 5.2.2** Any Sandwich graph $G$ of size $N$ has

$$
 cc(G) = O(N^{1.5})
$$

The above lemma shows that bit-reversal permutation does not provide the expected tradeoff which was proved traditionally. Therefore, a theoretical analysis on memory hard functions has recently gained acceptance in cryptography. Several significant results on memory hardness were proposed afterward which includes the analysis on the winner of PHC, Argon2i. However, asymptotic attacks on several data Independent Memory Hard Functions (iMHFs) including Argon2i were dismissed as not practically relevant by the proponents of the quick standardization of password hashing designs. A recent result of Crypto 2016 [29] known as the AlwenBlocki attack and a proof on the design of iMHFs opens new directions of research on the design of memory hard functions. It demonstrates that no DAG with constant indegree can completely resist the attack. More precisely, they show that any iMHF is at best $c$-ideal for $c = \Omega(log^{1-\epsilon}N$ and any $\epsilon > 0$. In particular, this means that ideal iMHFs do not exist [29]. In practice, moderately hard to compute functions have proven to be useful security primitives.
Table 5.1: Bounds on the amortized energy complexity taken from [30].

<table>
<thead>
<tr>
<th>Memory Hard Function (MHF)</th>
<th>Building Block</th>
<th>$E_R(A)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rig.v2 59</td>
<td>Blake2 35</td>
<td>$O(N^{1.70})$</td>
</tr>
<tr>
<td>TwoCats 69</td>
<td>Blake2 35</td>
<td>$O(N^{1.75})$</td>
</tr>
<tr>
<td>Gambit 134</td>
<td>Keccak Permutation 40</td>
<td>$O(N^{1.75})$</td>
</tr>
<tr>
<td>Lyra2 102</td>
<td>Blake2 35</td>
<td>$O(N^{1.67})$</td>
</tr>
<tr>
<td>or Keccak Permutation 40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pomelo 167</td>
<td>Basic Operations</td>
<td>$O(N^{1.83})$</td>
</tr>
</tbody>
</table>

Memory Hardness of some Designs from PHC [16] The result from [29] shows an attack against any iMHF for which the DAG $G$ is not depth-robust. A DAG, $G$ is not depth-robust if there is a relatively small set $S$ of nodes such that after removing $S$ from $G$ the resulting graph (denoted $G - S$) has low depth (i.e. contains only short paths). The analysis from [30] on iMHFs uses the following concept and the attack is summarized in Table 5.1.

- **AT Complexity** $AT_R(A)$: The AT complexity of an algorithm $A$ is the product of area ($A$) of an implementation of the algorithm in an application specific integrated circuit (ASIC) and the time ($T$) it takes the circuit to produce output. AT complexity usually estimates the financial cost of implementing an algorithm. To upper-bound the cost of a brute-force attack on an Memory Hard Function (MHF), the amortized AT-complexity of the attack per instance of the MHF is computed. To make the results more independent of the technology it is used by parameterizing the complexity using a core-memory energy ratio $R \in \mathbb{R}^+$ denoting the ratio of the size of an (on-chip) implementation of the underlying function and area required to store one of its outputs.

- **Energy Complexity** $E_R(A)$: The energy complexity of an algorithm $A$ measures the amount of electricity required by an ASIC implementing the algorithm per evaluation. To make the results more technology independent, a core-memory area parameter $\bar{R} \in \mathbb{R}^+$ which denotes the ratio between the amount of electricity used to evaluate the function and the amount of electricity needed to store the output of the function for an equivalent amount of time is adapted.

- The AT and energy quality of the iMHF can be expressed in terms of the complexity of ‘pebbling’ the graph. These complexities are upper-bounded in terms of the depth-robustness of the DAG. There should not be an algorithm that evaluates an iMHF at energy and AT complexity lower than $N^2$.

5.3 Overview of Three Cache-timing Attack Resistant Password Hashing Algorithms

In this section, we briefly explain three cache timing attack resistant password hashing algorithms submitted to Password Hashing Competition (PHC) [16]. Overview of these
algorithms is provided ahead (though details of Argon2 and Rig has been explained in Chapter 3).

5.3.1 Argon2i [45]

Argon2 - Version 1.2.1 is the winner of PHC [16]. The first version was submitted as Argon and later updated to Argon2. Recently the new Version 1.3 of Argon2 has appeared which addresses the problem of memory optimization reported in [98]. For our work, we focus on the version 1.2.1 which was the PHC winner. Argon2 specifies two variants, Argon2d which follows input dependent memory access pattern and Argon2i which follows input independent memory access pattern. Both are effective for different use cases: Argon2d for computing crypto-currencies and Argon2i for password hashing, key derivation etc. [45]. Both the variants are different only at the point of index (of a matrix) computation. Our TMTO analysis is applicable to the variant Argon2i and we explain this design next.

The hash function Blake2b [35] represented as \( H \) and the compression function based on Blake2b permutation represented as \( G \) are used. First the variable length password \( P \) and salt \( S \) with other parameters are hashed using \( H \) to produce \( H_0 \). This \( H_0 \) is used to generate a memory matrix \( M_{i,j} \), \( 0 \leq i < p \) and \( 0 \leq j < q \) where \( p \) is the number of lanes (rows) and \( q = m/p \) is the number of columns computed as below.

\[
\begin{align*}
M_{i,0} &\leftarrow G(H_0, i \parallel 0), \\
M_{i,1} &\leftarrow G(H_0, i \parallel 1), \\
M_{i,j} &\leftarrow G(M_{i,j-1}, M_{\phi(i,j)}), \quad 0 \leq i < p, \quad 2 \leq j < q \\
\text{Output} &\leftarrow H(M_{0,q-1} \oplus M_{1,q-1} \oplus \cdots \oplus M_{p-1,q-1})
\end{align*}
\]

where the function \( \phi(i, j) \) computes the index of the matrix \( M \) and its computation is either password-dependent (Argon2d) or password-independent (Argon2i).

**Index Computation** \( \phi(i, j) \): The memory matrix \( M \) is further partitioned in \( S = 4 \) slices. Intersection of a slice and a lane (row) is mentioned as the segment of length \( q/S \). To compute the index, two round compression function \( G \) is run in counter mode with counter \( i \). The first round input to \( G \) is a string of all-zeroes and the second round input is constructed as follows:

\[
r \parallel l \parallel s \parallel m \parallel t \parallel x \parallel i \parallel 0
\]

where, \( r \) is the pass number, \( l \) is the lane number, \( s \) is the slice number, \( m \) is the total number of memory blocks, \( t \) is the total number of iterations, \( x \) is 1 for Argon 2i and \( i \) is the counter starting in each segment from 1. Each application of \( G \) produces a 64-bit value. The two applications of \( G \), therefore, produce a 128-bit value \( J_1 \parallel J_2 \) where \( |J_1| = |J_2| = 64 \). To get the memory index \( \phi(i, j) \), compute \( l = J_2 \mod p \) which determines the index of the lane from which the block will be taken. If \( r = s = 0 \), then \( l \) is set to the current lane index. Then determine the set of indices \( R \) that is referenced for given \( M_{i,j} \) according to the following rules as mentioned in [45]:

101
1. If \( l \) is the current lane, then \( \mathcal{R} \) includes all blocks computed in this lane, which are not overwritten yet, excluding \( M_{i,j-1} \).

2. If \( l \) is not the current lane, then \( \mathcal{R} \) includes all blocks in the last \( S - 1 = 3 \) segments computed and finished in lane \( l \). If \( M_{i,j} \) is the first block of a segment, then the very last block from \( \mathcal{R} \) is excluded.

Then take a block \( z \) from \( \mathcal{R} \) by enumerating blocks in \( \mathcal{R} \) in the order of construction as below. \( J_1 \rightarrow |\mathcal{R}|(1 - \frac{(j_1)^2}{2^{64}}), x = (|\mathcal{R}|*x)/2^{32}, y = (|\mathcal{R}|*x)/2^{32}, z = |\mathcal{R}| - 1 - y. \) For detailed design of Argon, one may refer to [45].

**Figure 5.2:** General overview of the design Single-pass Argon2 with \( p \) lanes and 4 slices, taken from [41].

### 5.3.2 Catena [88]

The design Catena provides two variants supporting input (password) independent memory access: Catena-Butterfly which is represented as a stack of double-butterfly graphs [51], and Catena-Dragonfly which is based on bit-reversal graphs [117]. Catena uses a function \( H \) which implements Blake2b [35], a function \( H' \) which implements a single round of Blake2b including finalization, denoted as Blake2b-1, a randomization layer \( \tau \) (optional), and a “memory-hard” function \( F \). Both the variants of Catena differ in the choice of this function \( F \) specified as \( F^g_{\lambda} \), where \( F \) can be represented as a DAG with depth \( \lambda \) and \( 2^g \) nodes at each level. The variable \( g \) is called the garlic parameter. First, the algorithm initializes the variable \( x \) by setting it equal to the hash value computed on the concatenation of the three inputs: tweak \( t \), salt and password. The garlic parameter \( g \) defines the time and memory requirements for Catena. The value \( x \) is then updated by the function \( flap \) to produce the final password hash as shown in Fig. [5.3]. The \( flap \) function has three phases. In the first phase, a memory of size \( 2^g \cdot n \) bits is initialized, where \( g_{low} \leq g \leq g_{high} \) and \( n \) (bits) is the output length of the underlying hash function. The second phase calls the function \( \tau \) (optional) which depends on the public input \( \gamma \). Finally, the third phase calls a memory-hard function \( F \). When \( F \) is instantiated with \( BRH^g_{\lambda} ( (g, \lambda) - Bit \ Reversal \ Hashing) \) it is
denoted as Catena-BRG and when $F$ is instantiated with $DBH^g_\lambda$ ( $(g, \lambda)$ - Double Butterfly Hashing) it is denoted as Catena-DBG. The overview of the design applying the function flap is shown in Fig. 5.3.

![Diagram of Catena design with flap function]

Figure 5.3: General overview of the design Catena applying the function $flap$, taken from [88]. Flap function is the $\lambda$-times implementation of the memory-hard function $F$, where $F$ can be a bit-reversal graph as shown in Fig. 5.8 or a double butterfly graph as shown in Fig. 5.10.

5.3.3 Rig [59]

The algorithm Rig provides two variants where the general construction is represented as Rig $[H_1, H_2, H_3]$. The strictly sequential variant is denoted as Rig $[Blake2b, BlakeCompress, Blake2b]$ and the optimized variant which improves the performance by performing memory operations in larger chunks is represented as Rig $[BlakeExpand, BlakePerm, Blake2b]$. Both the variants differ in the instantiations of the functions $H_1$, $H_2$ and $H_3$. We provide the general description of the design which is similar to the sequential variant.

The algorithm defines a round with four phases. Phase 1 is called the initialization phase which computes the hash of the value $x$ derived from password, salt and other parameters and produces the output $\alpha$. The hash function is represented as $H_1$ and instantiated with Blake2b. Next phase 2 is called the ‘setup phase’. This phase uses the value $\alpha$ and initializes two arrays $k$ and $a$ each of size $m = 2^{m_c}$ where $m_c$ is taken as the input to define the required memory units. Each element of both the arrays is generated from the output of a hash function. The function $H_2$ is implemented as 1-round of Blake2b. Next phase is the ‘iterative transformation phase’ and is designed to update the stored array values $\lambda$-times where $\lambda$ is the number of iterations. In this phase, each hash computation represented by $H_2$ takes input from both the arrays. Array $k$ is accessed computing bit-reversal permutation on the indices and array $a$ is accessed sequentially. The $m$- computations of $H_2$ at setup phase and $\lambda \times m$ computations of $H_2$ at iterative transformation phase altogether are denoted
as function $H_2$. Specifically, the function $H_2$ is the memory-hard function which accesses and updates $\lambda$-times the above mentioned arrays $a$ and $k$. Both the arrays are generated from the output of a hash function, therefore, without loss of generality a single node in the graphical view of the $H_2$ accommodates two elements, one each from two different arrays. Hence, the function $H_2$ can be represented as $\lambda$-times implementation of function $G$ where $G$ is a bit-reversal straight graph as explained in Fig. 5.9. The last phase is called the ‘output generation phase’. This phase computes one hash (represented by $H_3$) taking salt as input with the last chaining value, produces the final output of each round. If round=1, this output is considered as the password hash otherwise the output of this phase is considered as the input to the next round. The value of $m$ at round $i$, i.e. $m_i$ is updated at round $i+1$ as: $m_{i+1} = 2 \times m_i$ and other descriptions remain same. The overview of the Rig design where round=1, is shown in Fig. 5.4.

Figure 5.4: Overview of the design Rig taken from [59]. The function $H_2$ internally updates $\lambda$-times the values of two arrays $a$ and $k$ each having $m$-entries applying function $G$ where the function $G$ is a bit-reversal straight graph as shown in Fig. 5.9.

5.4 Preliminaries

In this Section we give a brief overview of different graph structures which are the key concepts of our proposed cryptanalysis technique.
Directed Acyclic Graph (DAG)  A directed graph is an ordered pair \((\mathcal{V}, \mathcal{E})\) such that \(\mathcal{V}\) is a set of nodes and \(\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}\). Every edge \(e = (X_i, X_j)\) in the set \(\mathcal{E}\) is ordered. A directed graph \(\mathcal{G}\) is acyclic if it does not contain any directed cycle.

Bit-Reversal Permutation \([117]\)  A bit-reversal permutation is a permutation of a sequence of \(m\) elements with \(m = 2^k\) where \(k \in \mathbb{N}\). The elements are indexed from 0 to \(m - 1\) and to permute the elements the bits of indices represented in binary form are reversed. Each element is then mapped to the new location as per the reversed value of indices from 0 to \(m - 1\). Example, for \(k = 3\) and \(m = 2^3\) elements, the bit reversal graph with indices 0, 1, \(\cdots\), 7 is shown in Fig. 5.5.

![Bit-Reversal Permutation of \(m = 2^3\) elements with 3-bit binary representation of indices.](image)

5.4.1 Description of Some Directed Acyclic Graphs

In this work, we analyze DAGs consisting of a two-dimensional matrix of nodes. The characteristics of the graph are determined by the connectivity (dependency) between the nodes. We next describe some graphs which are useful for analyzing the designs Catena [88] and Rig [59]. The graphical representation of these schemes is derived from the following 4 types of graphs which are named according to the property followed by their edges as shown in Fig. 5.6.

1. The Sequential graph is obtained by connecting all the nodes of the graph sequentially (level-wise).
2. The Vertical graph is obtained by connecting all the nodes of the graph vertically (level-wise).
3. The BitReversed graph is obtained by applying bit-reversal permutation \([117]\) on each node (level-wise).
4. The Butterfly graph [68][51] is obtained by placing two back-to-back Fast Fourier Transformation (FFT) graphs after omitting one row in the middle.

The analyzed password hashing schemes can be graphically represented by overlaying these 4 types of graphs in various combinations as described ahead.
A $(N, \lambda)$-Straight Graph with $V$ vertices and $E$ edges can be formed by overlaying the Sequential and Vertical edge types graphs. $\lambda$ denotes the depth of the graph and $N = 2^k$ where, $k \in \mathbb{N}$ is the number of nodes at each layer. An example of $(8, 2)$-Straight graph is shown in Fig. 5.7. It is a simple and symmetric graph where $\lambda = 2$ is the depth of the graph. The consecutive nodes of each level are connected sequentially and level-wise nodes are connected vertically. We use this graph to show the working of the DAG traversal algorithm (defined below) with respect to the designs Catena [88] and Rig [59].

A $(N, \lambda)$-Bit-Reversal Graph (Representing the Catena-BRG Construction [88]) A $(N, \lambda)$-Bit-Reversal Graph with $V = N$ vertices and $E$ edges can be formed by overlaying
the Sequential and BitReversed edge types graphs. λ is the depth of the graph and at each level number of nodes $N = 2^k$ where, $k \in \mathbb{N}$ (definition adapted from [59]). An example of (8, 2)-Bit-Reversal graph is shown in Fig. 5.8. As per the definition, it performs bit-reversal permutation at each level or it is a stack of $\lambda = 2$ bit-reversal permutation operations.

This graph represents the directed data dependency of Catena-BRG construction [88]. Specifically, it describes the flow of the flap function (see Fig. 5.3) with respect to its core memory-hard function $F$ as described in Section 5.3.2. The function $F$ instantiated with bit-reversal graph and denoted as $BRH^g_{\lambda}$, requires three inputs: $g$ that specifies the required number of nodes $(2^g)$ of the graph at each level, the value $x$ which is the input to process, and the value $\lambda$ which defines the depth of the graph. Therefore, Catena-BRG can be specified by a bit-reversal graph with $\lambda \times 2^g$ nodes representing the entire computation graph where the directed edges show the flow of the execution.

(\mathcal{N}, \lambda)-Bit-Reversal-Straight Graph (Representing the Rig Construction [59]) A $(\mathcal{N}, \lambda)$-Bit-Reversal-Straight Graph with $V = \mathcal{N}$ vertices and $E$ edges can be formed by overlaying the Sequential, Vertical and BitReversed edge types graphs. $\lambda$ is the depth of the graph and at each level the number of nodes $\mathcal{N} = 2^k$ where, $k \in \mathbb{N}$ and (definition adapted from [88]). An example of (8, 2)-Bit-Reversal-Straight graph is shown in Fig. 5.9 where $\lambda = 2$ is the depth of the graph and at each level, the number of nodes is 8.

This graph represents the directed data dependency of the $H_2$ function of the Rig construction [59], depicted in Fig. 5.4. The function $H_2$ accesses and updates values of two different arrays, each of size $m = 2^{m_c}$, at every level of the DAG (a level of the DAG is the
horizontal dashed line in Fig 5.9, which is further explained in Section 5.3.3). To simplify the graphical view, we consider storing both the arrays at the $i^{th}$ location of the memory arrays with a single node of the graph. Therefore $H_2$ is represented as bit-reversal-straight graph with $m$ nodes and each node of the graph accommodates two elements, one each from two different arrays. The number of iterations $n$ defines the depth of the graph i.e., the number of times the nodes are accessed and updated. The directed edges show the flow of execution of the algorithm.

$(N, \lambda)$-Double-Butterfly Graph (Representing the Catena-DBG Construction)\cite{88}

A $(N, \lambda)$-Double-Butterfly Graph with $V = N$ vertices and $E$ edges can be formed by overlaying the Sequential, Vertical and Butterfly edge type graphs. $\lambda$ is the depth of the graph and at each level the number of nodes $N = 2^k$ where, $k \in \mathbb{N}$ (definition adapted from \cite{88}). An example of $(8, 1)$-Double-Butterfly graph is shown in Fig. 5.10 where $\lambda = 1$ is the depth of the graph and the number of nodes at each level is 8.

This graph represents the directed data dependency of Catena-DBG construction \cite{88}. It describes the flow of the flap function (see Fig. 5.3) with respect to its core memory-hard function $F$ instantiated with double-butterfly graph. This function, represented as $DBH^g_{\lambda}$, requires three inputs: $g$ that specifies the required number of nodes ($2^g$) of the graph at each level, the value $x$ which is the input to process, and the value $\lambda$ which defines the depth of the graph. Specifically, the $(N = 2^g, 1)$-Double-Butterfly graph representation is stacked $\lambda$ times in Catena-DBG to create $(N = 2^g, \lambda)$-Double-Butterfly graph. The memory traversal pattern follows from the original FFT butterfly structure. Due to the significantly large number of operations and a large number of layers (for example, 5 in Fig. 5.10) at each level,
this graph traversal is significantly slower than the previous types. The re-computation effort increases exponentially with reduction in memory. This is due to the fact that in-degree of a node in the DAG corresponding to this design is high, e.g., each node has in-degree of 3 when the total number of nodes is 8 (see Fig. 5.10). The directed acyclic graph corresponding to Catena-DBG has $2^g$ nodes which are arranged as $\lambda$ stacks of double butterfly graphs. The directed edges in the DAG show the flow of execution of the algorithm as explained in Section 5.3.2.

5.5 Traversing a Dependency Graph to Analyse Trade-off Penalties

A password hashing design is considered memory hard, with respect to a pre-specified memory, if its implementation requires significantly larger runtime when the memory is reduced by even a fraction smaller than the pre-specified number. The expected increase in runtime is exponential in the amount of memory reduction.

Many designs in the PHC claim a strong time-memory tradeoff defense. However, there exists no general method to verify the claimed TMTO defense for a proposed algorithm. With the aim to give a solution, we provide a cryptanalytic approach and apply the technique on Catena, Rig and Argon2i. The representation of these algorithms as a directed acyclic graph, as explained in Section 5.4, accommodates the memory dependencies throughout their execution. Our proposed technique follows a simple approach to allow the flexibility to store the memory elements as per the choice of the implementor and then to perform on-the-fly computation of memory elements which are not stored at the time these elements are needed. This may increase the computation time from the usual implementation of the design. We compute the increased algorithmic runtime which we denote as re-computation penalty. This re-computation penalty provides the actual TMTO defense of the algorithm. However, an attacker is not obliged to follow the advice of the designer and may vary the memory storage to other nodes. This could potentially allow him to compute the password hash at a lower cost than the one envisaged by the designer. The algorithmic description of the proposed method is provided in Algorithm 5.1.

The graphical representation of a password hashing algorithm shows the memory dependencies between various memory elements as a password hash is computed in accordance with the design. The nodes of the graph represent the storage elements (memory) of the design and the arrows targeting the nodes show the dependencies. The Algorithm 5.1 traverses the nodes of the password hashing scheme following its actual implementation and computes the values that are not stored when required. Therefore ‘node’ plays an important role and below is the data structure defined to keep the state of the nodes during traversal.

```plaintext
structure Node
{
    integer X = 0, Y = 0;
}
```
boolean MemoryAllowed = false, MemoryValid = false, Traversed = false;
array Node [] Dependencies;
}

Initially, all the values of ‘node’ are set to false. Each ‘node’ keeps track of an array which includes all the nodes that derive its value. The password hashing schemes we analyze need memory equal to the number of nodes in one row. If enough memory is available, then there is no need to do any TMTO, and the computation takes the time it needs to process all the nodes once, i.e., the time of actual implementation of the scheme. If enough storage is unavailable then it requires to perform a tradeoff between time and memory and it is expected that it will require a significantly large number of operations to compute the values that are not stored. We explain the proposed technique with examples in the following Section.

5.5.1 Description of the Proposed Technique

The nodes of a graph are represented as a tuple specifying column and row numbers. Therefore the starting value (0, 0) contains the value corresponding the initial inputs of a password hashing algorithm and is assumed to be known. The algorithm takes as input the locations of the nodes that are allowed to be stored during the evaluation. The location can be all the nodes corresponding to a column or a row or some random locations throughout the graph. Therefore, there can be a large number of possible combinations of allowed-memory locations and the overall effort (computations) will depend on the allowed memory and its allocation in the complete graph. The structure Node has the field MemoryAllowed to let the algorithm know which node has storage and allow it to store the value when it is available/calculated during traversal and then mark the MemoryValid true to know the value is available for further computations. The algorithm starts traversing from the last node, i.e., from node (M-1, N-1) and runs iteratively backward, traversing node-to-node until all the dependencies are computed. To explain the traversal we follow the following notation.

\[(N^i, N^j) \rightarrow \{ (D^i, D^j), (D^{i+1}, D^{j+2}), \ldots \} \Rightarrow (N^{i+1}, N^{j+1}) \rightarrow \ldots\]
Algorithm 5.1: DAG Traverse

**Input:** graph(Node)-Dependency graph to traverse,
integer M- No. of columns, N- No. of rows

**Variables:**
Node n,
stack(Node) processing, dependency,
boolean dependencyfound,
list(Node) traverse

**Output:** list(Node) traverse - A list of nodes traversed by the algorithm.

1. n = graph[M-1, N-1];  \(\triangleright\) each node contains all its dependencies
2. while(true) do
3.  if(n.Traversed == false)
4.    foreach dependency in n.Dependencies do
5.      if (dependency.MemoryValid == false)
6.        dependency.push(dependency)
7.    end if
8.  end foreach
9.  n.Traversed ← true
10. else
11.    if dependency.count >0
12.      n = dependency.pop()
13.      processing.push(n)
14.      if (n.MemoryValid == false)
15.        add n to list traverse
16.    end if
17.  dependencyfound ← true;
18.  foreach Node d in n.Dependencies do
19.      if (d.MemoryValid == false)
20.        dependencyfound ← false
21.  end if
22. end foreach
23.  if (dependencyfound == true)
24.    while processing.count >0 do
25.      temp = processing.pop()
26.      if temp.MemoryAllowed == true
27.        temp.MemoryValid ← true
28.    end if
29.  end while
30.  graph.clearAllTraversed()  \(\triangleright\) clear graph to process next dependencies
31.  end if
32. else break  \(\triangleright\) when no dependency is left to process
33. end while
34. return list(Node) traverse
Where, \((N^i,N^j), ((N^{i+1},N^{j+1})) \ldots\) are the nodes, \((D^i,D^j),(D^{i+1},D^{j+2}), \ldots\) are the dependencies discovered during the traversal and \(i, j\) are the corresponding column and row numbers. Specifically, the aim of all computations is to find out the value of node \((M-1,N-1)\) where \(M\) is the number of columns and \(N\) is the number of rows, while the input \((0,0)\) is known. All the nodes in between need to be computed on the way. Algorithm 5.1 (as defined) is considering pointer arithmetic, so, when node \(n\) is pushed and then popped from the stack dependency, any changes to \(n\) will be reflected in the initial graph structure. For a better understanding of the proposed approach, we provide two examples which cover two different scenarios and also prove the validity of the method.

### 5.5.1.1 Example 1

A \((4,2)\)-Bit-Reversal Graph is shown in Fig. 5.11. Let us consider that all the nodes allow memory storage. The procedure can be performed by traversing the nodes in the following order starting with node \((3,2)\) where 3 is the column number and 2 is the row number. The traversal of \((4,2)\)-Bit-Reversal Graph follows our notation of traversal including the dependencies which is explained above and needs total 12 steps covering the following path.

\[
(3,2) \Rightarrow \{(2,2),(3,1)\} \Rightarrow (3,1) \Rightarrow \{(2,1),(3,0)\} \Rightarrow (3,0) \Rightarrow \{(2,0)\} \\
\Rightarrow (2,0) \Rightarrow \{(1,0)\}, \ (1,0) \Rightarrow \{(0,0)\} \Rightarrow (0,0) \Rightarrow \{} \\
\Rightarrow (2,1) \Rightarrow \{(1,1)\} \Rightarrow (1,1) \Rightarrow \{(0,1)\} \Rightarrow (0,1) \Rightarrow \{} \\
\Rightarrow (2,2) \Rightarrow \{(1,2)\} \Rightarrow (1,2) \Rightarrow \{(0,2)\} \Rightarrow (0,2) \Rightarrow \{} \\
\]

Considering the node \((3,2)\) as the starting point of Fig. 5.11 the initial dependencies are \((2,2)\) and \((3,1)\) which again have further dependencies. For each node, the chain of its dependency nodes are backtracked. Therefore, the dependency path of node \((3,1)\) includes \((2,1)\) and \((3,0)\) then from \((3,0)\) to \((2,0)\) which gives \((1,0)\) and finally for \((1,0)\) the dependency \((0,0)\) ends the current chain of dependencies. Whenever a dependency is found it is put in the stack.
Table 5.2: Traversal for Example 1

<table>
<thead>
<tr>
<th>Position</th>
<th>Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>(3,2)</td>
<td>(2,2), (3,1)</td>
</tr>
<tr>
<td>(3,1)</td>
<td>(2,1), (3,0)</td>
</tr>
<tr>
<td>(3,0)</td>
<td>(2,0)</td>
</tr>
<tr>
<td>(2,0)</td>
<td>(1,0)</td>
</tr>
<tr>
<td>(1,0)</td>
<td>(0,0)</td>
</tr>
<tr>
<td>(0,0)</td>
<td></td>
</tr>
<tr>
<td>(2,1)</td>
<td>(1,1)</td>
</tr>
<tr>
<td>(1,1)</td>
<td>(0,1)</td>
</tr>
<tr>
<td>(0,1)</td>
<td></td>
</tr>
<tr>
<td>(2,2)</td>
<td>(1,2)</td>
</tr>
<tr>
<td>(1,2)</td>
<td>(0,2)</td>
</tr>
<tr>
<td>(0,2)</td>
<td></td>
</tr>
</tbody>
</table>

As the value of (0,0) is always known, it helps to end the dependency chain and also to compute the values of the stack by popping them one-by-one. Therefore, all the nodes of the current stack are processed. As we are considering the scenario where all the values are allowed storage, the values at the memory location (1,0), (2,0) and (3,0) are updated after their first processing. Next, the dependency (2,1) is processed which needs only the dependency (1,1) as another dependency (1,0) is known. Dependency (1,1) needs (0,1) which is known from (0,0), (3,0). Next the values (0,1), (1,1), (2,1) and (3,1) are updated. The stack is then processed again and takes the value (2,2) and the process continues until all the nodes are processed and all the dependencies are met. The method takes total 12 computations as below.

\[(3,2) \rightarrow (3,1) \rightarrow (3,0) \rightarrow (2,0) \rightarrow (1,0) \rightarrow (0,0) \rightarrow (2,1) \rightarrow (1,1) \rightarrow (0,1) \rightarrow (2,2) \rightarrow (1,2) \rightarrow (0,2)\]

The traversal steps for Example 1 are shown in Table 5.2.

5.5.1.2 Example 2

A (4,2)-Bit-Reversal Graph is considered as shown in Fig. 5.12 where only the first column (nodes in blue) is allowed memory storage. The rest of the nodes have no memory and during traversal (*MemoryValid* set to false), they need to be re-computed every time they are encountered.

The complete traversal for a (4,2)-Bit-Reversal Graph as shown in Fig. 5.12 needs 35 steps and the steps take the following path.

\[(3,2) \rightarrow \{(2,2), (3,1)\} \Rightarrow (3,1) \rightarrow \{(2,1), (3,0)\} \Rightarrow (3,0) \rightarrow \{(2,0)\}\]
\[\Rightarrow (2,0) \rightarrow \{(1,0)\} \Rightarrow (1,0) \rightarrow \{(0,0)\} \Rightarrow (0,0) \rightarrow \{\}\]
\[\Rightarrow (2,1) \rightarrow \{(1,1), (1,0)\} \Rightarrow (1,0) \rightarrow \{\}\]
Figure 5.12: (4, 2)-Bit-Reversal Graph with only memory in the first column.

⇒ (1, 1) → {(0, 1), (2, 0)} ⇒ (2, 0) → {(1, 0)} ⇒ (1, 0) → {}
⇒ (0, 1) → {(3, 0)} ⇒ (3, 0) → {(2, 0)} ⇒ (2, 0) → {(1, 0)} ⇒ (1, 0) → {}
⇒ (2, 2) → {(1, 2), (1, 1)} ⇒ (1, 1) → {(2, 0)} ⇒ (2, 0) → {(1, 0)} ⇒ (1, 0) → {}
⇒ (1, 2) → {(0, 2), (2, 1)} ⇒ (2, 1) → {(1, 1), (1, 0)} ⇒ (1, 0) → {}
⇒ (1, 1) → {(2, 0)} ⇒ (2, 0) → {(1, 0)} ⇒ (1, 0) → {}
⇒ (0, 2) → {(3, 1)} ⇒ (3, 1) → {(2, 1), (3, 0)} ⇒ (3, 0) → {(2, 0)} ⇒ (2, 0) → {(1, 0)} ⇒ (1, 0) → {}
⇒ (2, 1) → {(1, 1), (1, 0)} ⇒ (1, 0) → {}
⇒ (1, 1) → {(2, 0)} ⇒ (2, 0) → {(1, 0)} ⇒ (1, 0) → {}

5.5.1.3 Limitation of the Proposed Technique

The proposed technique is a theoretical TMTO attack technique. Realistic application needs different resources and different approach which is beyond the scope of current work. However, we target to address it further in future. Therefore, currently it is difficult to provide the actual memory-hardness and time-complexity following the proposed algorithmic approach. The expected result to show memory hardness requires a higher minimum bound which is shown in Section 5.6 by explaining the results obtained from the implementation of three algorithms, Argon2i, Catena and Rig.

5.6 Results

All experiments documented in this section and in Section 5.7 follow the same setup as described ahead. In order to get consistent results for the different algorithms, we perform all the test on a single machine with the code compiled by the same compiler. The details are as follows:

- **CPU**: Intel Core i7 4770 (Turbo Boost: ON) - Working at 3.9 GHz
- **RAM**: Double Channel DDR3 16 GB (2400 MHz)
- **Compiler**: gcc / g++ v4.9.2 ( -march=native and -O3 flags were set if not already in the makefiles). This would cause the compiler to use the AVX-2 instructions.
• **OS:** UBUNTU 14.04.1, on HYPER-V, on Windows-8.1 with 8 GB allocated RAM to the VM. We also performed benchmarks on native Linux OS to make sure that the virtualization does not cause any changes in the results.

For consistency, we use single-threaded versions of the algorithms. The obtained figures of re-computation penalties for algorithms Cantena and Rig are independent of the choice of input parameters. This is due to the deterministic behavior of both the algorithms. However, we experiment taking similar values for all algorithms to maintain consistency. The design Argon2i provides different graph structures for different public parameters (independent of password and salt), therefore we explain its TMTO analysis separately in Section 5.6.1. Hence, in case of Argon2i, it is important to mention the parameter choices (the exact values) to ease the verification of claimed results.

The DAG traversal algorithm provided in Section 5.5 can be used to compute the re-computation penalties for any graph by varying the memory storage. To apply Algorithm 5.1, a large number of memory configuration is possible. For example, a column of a graph can be enabled or disabled, i.e., when a column is enabled all the nodes for that column have memory storage abilities, otherwise not. We apply Algorithm 5.1 to the following cases and come up with re-computation penalties in different memory sizes. For our experiments, we only allow a limited set of configurations, i.e. columns are enabled or disabled. The considered password hashing designs are regularly structured and therefore the approach used to enable/disable columns is easy to implement. For analyzing reduced memory scenarios, we try to evenly distribute the memory along columns starting from the first column. The design Argon2i provides different graph structures for different implementation parameters, therefore we explain its TMTO analysis separately in Section 5.6.1. Following are the graphs that represent a fixed structure for all parameter choices and we compute the re-computation penalties for them.

- $(N, \lambda)$-Straight Graph (SG)
- $(N, \lambda)$-Bit-Reversal Graph (Catena BRG)
- $(N, \lambda)$-Double Butterfly Graph (Catena DBG)
- $(N, \lambda)$-Bit-Reversal-Straight Graph (Rig Graph)

The cumulated results (including Argon2i) for the graphs are as shown in Fig. 5.13. It shows the comparison of the re-computation penalty with change in allowed memory proportion. It is clear from the results that the re-computation penalty increases drastically for reductions in memory size. For the experiments, we fix $M = 64$ (columns) even though we experimented with larger values upto 512. This is because the characteristics of the DAGs do not depend significantly on the value of $M$ and follow similar pattern of rate of growth, but the runtime becomes large. All data from Table 5.3, 5.4, 5.5 and 5.7 are included in Fig. 5.13.
Figure 5.13: Re-computation Penalties for Graphs

Table 5.3: Re-computation Penalties for Graphs with 4 rows ($\lambda = 3$)

<table>
<thead>
<tr>
<th>Memory (%) Proportion (%)</th>
<th>Straight Graph</th>
<th>Bit-Reversal Graph (Catena BRG)</th>
<th>Bit-Reversal-Straight Graph (Rig)</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>15</td>
<td>19</td>
<td>97</td>
</tr>
<tr>
<td>25</td>
<td>75</td>
<td>93</td>
<td>551</td>
</tr>
<tr>
<td>12.5</td>
<td>460</td>
<td>909</td>
<td>5036</td>
</tr>
<tr>
<td>6.25</td>
<td>3181</td>
<td>8019</td>
<td>43143</td>
</tr>
</tbody>
</table>
Table 5.4: Re-computation Penalties for Graphs with 5 rows ($\lambda = 4$)

<table>
<thead>
<tr>
<th>Memory (%)</th>
<th>Straight Graph</th>
<th>Bit-Reversal Graph</th>
<th>Bit-Reversal-Straight Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>62.5</td>
<td>32</td>
<td>48</td>
<td>445</td>
</tr>
<tr>
<td>31.25</td>
<td>254</td>
<td>401</td>
<td>4190</td>
</tr>
<tr>
<td>15.625</td>
<td>2724</td>
<td>10215</td>
<td>92483</td>
</tr>
<tr>
<td>7.8125</td>
<td>35120</td>
<td>197389</td>
<td>1707950</td>
</tr>
</tbody>
</table>

Table 5.5: Re-computation Penalties for Graphs with 6 rows ($\lambda = 5$)

<table>
<thead>
<tr>
<th>Memory (%)</th>
<th>Straight Graph</th>
<th>Bit-Reversal Graph</th>
<th>Bit-Reversal-Straight Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>75</td>
<td>63</td>
<td>113</td>
<td>1971</td>
</tr>
<tr>
<td>37.5</td>
<td>777</td>
<td>1736</td>
<td>31270</td>
</tr>
<tr>
<td>18.75</td>
<td>14378</td>
<td>119358</td>
<td>2152596</td>
</tr>
<tr>
<td>9.375</td>
<td>341447</td>
<td>5060331</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 5.6: Re-computation Penalties for Double-Butterfly Graph ($\lambda = 1$)

<table>
<thead>
<tr>
<th>Memory Proportion (%)</th>
<th>Double-Butterfly Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>18</td>
</tr>
<tr>
<td>25</td>
<td>922</td>
</tr>
<tr>
<td>12.50</td>
<td>60504</td>
</tr>
<tr>
<td>6.25</td>
<td>2043702</td>
</tr>
</tbody>
</table>

The algorithm uses stacks during calculations, but, the maximum size of the stack is bounded by the maximal length of a single path (plus sub paths). As a result, the algorithm does not consume a large amount of memory even for huge re-computation dependency tree calculations.

5.6.1 Re-computation Penalty for Argon2i

The design Argon2 is the winner of the PHC competition [16] and an IETF internet-draft [43] proposes it to be made a standard password hashing design for internet protocols. Therefore it is important to have detailed theoretical as well as practical analysis of this design. Our proposed approach can help analyze any password hashing scheme whose execution can be represented as a DAG. As we discuss in Section 5.3.1, Argon2i falls in the cache-timing attack resistant category for having password independent memory access pattern. However, it does not follow a fixed DAG structure over all parameter choices as the memory access depends on the pseudorandom output of the compression function $G$ which is non-uniform. Due to the non-uniform $G$, we get different DAGs depending on the chosen input. The nodes of the graph at each level represent the memory elements of the corresponding row of the
memory matrix of Argon2i as described in Section 5.3.1. Except for the first two nodes of the first level, all nodes are connected to its previous node and a node value derived from the output of the compression function $G$. We provide three different graphs for different parameter choices in Fig. 5.14, 5.15 and 5.16. The parameters $p$ is set to 1 which means we are considering the single threaded version of Argon2i.

A recent attack on Argon2i, presented by Corrigan-Gibbs et al. [98], shows that having a pseudorandom memory access pattern for Argon2i does not provide the advantage over fixed memory access as employed in designs like Catena and Rig. In fact, Corrigan-Gibbs et al. exploit the non-uniform distribution of memory accesses against Argon2i. We explain the main idea of the attack against Argon2i with an example. Consider the memory access pattern in the last few nodes of Argon2i in Fig. 5.14. The last 6 nodes accessed among the total 8 in this figure follow the sequence 0, 1, 1, 3, 3, 3. This implies that these 8 nodes can be computed with only 3 nodes (current node and the previous ones numbered 1 and 3). If the design was memory hard, we should have required all 8 nodes to compute the output of Argon2i but we can manage to compute this while storing only 3 nodes. Hence there is no recomputation penalty. Note that this behavior is caused by the one-to-many mapping in Argon2i (e.g. node 3 is used to compute nodes 5, 6 and 7 in Fig. 5.14). This provides the attacker with an opportunity to reduce the required number of memory units throughout the Argon2i computations without paying any penalty.

In order to make a fair comparison between Argon2i, Catena and Rig, we analyze these designs with a similar number of memory units. Since each node of Argon2i takes 1024 KiB memory while each node of Rig and Catena takes 512 KiB, we compare $n$ nodes of Argon2i with $2n$ nodes of Catena and Rig. Argon2i creates the memory array depending on the previous node and a node derived from the pseudorandom output of function $G$. Thus, the memory access pattern for Argon2i at the first level itself starts to contribute to its memory hardness. On the other hand, the memory access pattern for Catena and Rig at the first level is derived from the initial input sequentially. For levels $l > 1$, the memory access depends on the previous node and a node derived from a fixed permutation. Therefore, as far as the computation effort is concerned, iteration $i$ for Argon2i is equivalent to computations with iteration $i + 1$ of the designs Catena and Rig. We consider this equivalence in our experimental results.

To obtain the recomputation-penalty for Argon2i, we provide the dependency graph (DAG) generated from the above mentioned input parameters. To compute TMTO, the selection of nodes to allow memory are obtained after applying different heuristic approaches. This is required for Argon2i because of the non-uniform dependencies on nodes, as already mentioned. Since the selection of optimal nodes varies depending on the given input for Argon2i, it is not feasible to get a closed-form solution for it. However, following the probabilistic analysis on the execution pattern of Argon2i, it is possible to provide a randomized algorithm to get the optimal nodes considering the frequency of dependencies. We later explain the basis of our heuristics for searching the optimal nodes.

To select the nodes with a higher frequency of dependencies, we used the following strategy. First, we list the nodes by sorting them on decreasing order of dependencies, and then
choose the nodes maintaining a range of distance between any two nodes. The reason for our choice is that if two nodes are close-by in the DAG, then we can compute the next node from the previous one with a small cost. This can be seen as a “greedy strategy” with regard to the locally optimal nodes in order to obtain the globally optimum solution. Our heuristic was chosen based on several experiments we performed on graphs of small order and checking the number of computations required as the memory was reduced.

Our experimental results show that the non-uniform distribution of Argon2i affects the memory-hardness of the design, causes a weakness. The results of our experiments showing the re-computation penalty for Argon2i are shown in Table 5.7.

<table>
<thead>
<tr>
<th>Memory Proportion (%)</th>
<th>λ = 2</th>
<th>λ = 3</th>
<th>λ = 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>75</td>
<td>118</td>
<td>157</td>
</tr>
<tr>
<td>25</td>
<td>163</td>
<td>382</td>
<td>1383</td>
</tr>
<tr>
<td>12.5</td>
<td>4008</td>
<td>80906</td>
<td>845846</td>
</tr>
</tbody>
</table>

Figure 5.14: Argon2i at t=1, m= 8 blocks

Figure 5.15: Argon2i at t=2, m= 8 blocks

Figure 5.16: Argon2i at t=3, m= 16 blocks
5.6.1.1 Argon2i - Version 1.3 [42]

This version was developed recently to mitigate the attack shown in [98] on the previous version which was the PHC winner. This is similar to the design explained in Section 5.3.1 except for iterations $t > 1$ where for each computation of the block $M_{i,j}$, the new block is XORed to its previous value instead of just overwriting the previous one. The problem without this XOR operation was that for each block computation there was a time gap between the moment the block was used for the last time (through index computation) and the moment it is overwritten. Therefore it was possible to drop the block after last referenced. This is no longer possible in the modified version. However, this version 1.3 gives little overhead on the performance as $(t - 1) \times i \times j$ more XOR operations are performed.

5.7 Performance Analysis

For consistency, we use single-threaded versions of the algorithms. There was no significant difference between the best case and worst case running time of the algorithms. Therefore all the experiments were run 5 times and the average timings have been taken. The performance graph is shown in Fig. 5.17 which shows the execution time vs. memory for all algorithms benchmarked. All parameters and the experimental setup are same as mentioned in Section 5.6. For Argon2, we consider only the cache-timing attack resistant variant, Argon2i, for the performance analysis. The latest version of the code is cloned from [19]. The performance of Argon2i is similar to Catena Dragonfly and slower than the design Rig. The performance figure shows that Catena-Dragonfly is much faster than Catena-Butterfly, but, the Dragonfly version is shown not to be memory-hard in [10]. The latest version of code at the time of benchmark was cloned from [17] and optimized SSE implementation is used for both the benchmarks. One of the reasons for the slow nature of the Butterfly version is the need for $2 \cdot g$ rows for processing. This property of Catena-DBG, combined with the relatively small read-writes to the RAM makes the overall structure significantly slow. Even the fastest version of Catena-Butterfly-Blake2b-1 can only achieve overall memory hashing speed of around 80 MiB/s. The Catena-Dragonfly is much faster due to the significantly reduced number of rounds as compared to Catena-Butterfly. In addition to this, every node in the Catena-BRG graph has a dependency on two previous ancestors as opposed to three in Catena-DBG. This leads to reduced number of random memory accesses and faster speeds.

For the performance of the design Rig we use the latest version from [20]. We use the optimized implementation with the Blake2b round using AVX-2 instructions. All default settings are used as described in the code and Makefile. One source code improvement is the removal of the writing of the data back to the memory in the last row, this change resulted in around 5% improvement in overall performance for small values of $N$.

We also include the performance of ‘Scrypt’ [132] algorithm which is the first memory-hard algorithm for password hashing. There are several implementations of Scrypt available, we use one of the fastest variants of the implementation from [18] with AVX2 implementation...
Figure 5.17: Execution time vs. memory, faster feeling of memory is better.
5.8 Summary

It is difficult to provide a general technique to analyse the time-memory tradeoff for memory hard designs. In this chapter, we propose a technique for traversal of DAGs to analyze the TMTO. Therefore, it is applicable to the designs which can be represented as a DAG. We apply the proposed technique on three cache-timing attack resistant designs namely, Argon2i, Catena and Rig by performing preliminary analysis with various parameters and TMTO options.

The performance graph in figure 5.17 shows the execution time vs. memory for all the memory-hard algorithms benchmarked. It is clear that Catena-Butterfly is the slowest and take a significant amount of time in hashing passwords with moderate to large amounts of memory. The performance of Catena is unlikely to significantly improve even with native assembly implementation.

Argon2i and Rig-Version 2 provide good performance in a wide range of use cases, though Argon2i is slower than Rig. The attack of [98] on Argon2i shows reduction over claimed TMTO defense which is also visible in our analysis as shown in Table 5.7.

It would be interesting to apply our proposed time-memory tradeoff analysis technique to evaluate other complex cryptographic designs similar to password hashing schemes where proposing a standard mathematical/theoretical model for analysis is significantly difficult. One possible extension of this work can be to apply the proposed analysis technique on other password hashing designs as well. Our experimental results show that simple DAGs are efficient in terms of performance but are not memory hard. Use of complex DAG provides expected security against time-memory tradeoff but shows poor efficiency. Combining two simple graphs in the representation of a DAG achieves both security and efficiency. Hence, to provide a general technique to combine different simple graphs for the efficient design (both in terms of TMTO defense and performance) of a password hashing scheme can be another interesting problem to explore.