Cryptanalytic Time-Memory Tradeoff for Password Hashing Schemes

Donghoon Chang, Arpan Jati, Sweta Mishra, and Somitra Kumar Sanadhya

Indraprashtha Institute of Information Technology, Delhi (IIIT-Delhi), India
{donghoon,arpanj,swetam,somitra}@iiitd.ac.in

Abstract. 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 password hashing schemes (PHS). Increasing threat of password leakage from compromised password hashes demands a resource consuming algorithm to prevent the precomputation of the password hashes. A class of password hashing designs provide such a defense against TMTO attack by ensuring that any reduction in the memory leads to exponential increase in runtime. These are called *Memory hard* designs. However, it is generally difficult to evaluate the "memory hardness" of a given PHS design.
In this work, we present a simple technique to analyze TMTO for any password hashing schemes which can be represented 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. Although our technique is generic, we show its efficacy by applying it on three designs from the "Password Hashing Competition" (PHC) - Argon2i (the PHC winner), Catena and Rig. Our analysis shows that Argon2i fails to maintain the claimed memory hardness. In a recent work Corrigan-Gibbs et al. indeed showed an attack highlighting the weak memory hardening of Argon2i. We also analyze these PHS for performance under various settings of time and memory complexities.

Keywords: Time-Memory tradeoff, password, hashing, graph traversal, bit-reversal graph, double butterfly graph

1 Introduction

Passwords are the most convenient, cost effective and widely used solution for user authentication. 'Password Hashing' is the technique to perform one way transformation of the input password to convert it into a fixed length random output. These hashed passwords are commonly stored in the server database avoiding the plaintext storage of the passwords to maintain the minimum security of password authentication. As passwords are usually human generated and ideally required to be memorized entities, they are easy to guess [1]. The most common attack for password hashing is 'dictionary attack' [2] where an attacker creates a dictionary of most commonly used passwords with the corresponding password hashes using the password hashing algorithm. Creating the dictionary exhausting most commonly used passwords has been very efficient when implemented with cheap, massively parallel hardware like GPUs (graphics processing units). An attacker having the precomputed dictionary only needs to get the server database to match the entries which then discloses client passwords. Therefore, the design requirements for password hashing schemes highly depend on technological improvement over computational efficiency of the hardware. The state-of-the-art of current processing speed and the cost of memory impose the following major requirements for password-hashing construction [3].

- **One-wayness:** It should be difficult to obtain the password from the password hash.
- **Memory Hardness:** The construction should consume a fixed amount of memory (RAM) during execution. The operation should take significantly longer time (preferably exponential) if less than this pre-specified and fixed amount of memory is available.
- **Constant Time Operation:** The execution time of the construction should be fixed over all input choices for a fixed set of parameters. In particular, even if the password lengths are different, the time to compute password hashes should be the same.
- **Cache Timing Attack Resistance:** The construction should not leak information about the password due to cache timing or memory leakage at the time of physical implementation.
- **Client Independent Update:** The construction should support upgradation of the existing password hash to a different cost setting without the involvement of the client or the input password. For example, a password hashing design may provide for 128-bit security level today, but it may need to be updgraded to a 160-bit security at a future date. This upgradation should be possible while being oblivious to the client.

The *Memory Hardness* requirement for password hashing schemes is significant to prevent 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. A memory-hard design may suffer from side channel attack depending on the memory access pattern followed by the algorithm. Most existing password hashing designs follow a memory-access pattern which is either password-dependent or password-independent throughout their execution. If the memory-access pattern depends on the password then an attacker may leak some information about the secret password by observing this pattern. In fact, such a cache-timing attack on Scrypt [4] was described in [5]. Consequently, side channel attack resistance (i.e., password-independent memory access) has become a crucial requirements for password hashing.

Our Contribution: Most of the submissions of Password Hashing Competition [3] claim memory-hardness. However, there exists no easy way to verify the memory hardness claim of the designs following their algorithmic description. In this work, we provide an easy technique to analyze the memory hardness of those algorithms whose execution can be expressed as a Directed Acyclic Graph (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 trade-offs (varying memory and time) options. We apply the proposed technique on three cache-timing attack resistant algorithms, namely, Argon2i [6] (the winner of PHC [3]), Catena [7] and Rig [8] to obtain TMTO values for various combinations of options in these algorithms. 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 (specified in Section 6.1) to find the optimal points for memory reduction and then apply the proposed traversal algorithm to obtain the TMTO results.

We also analyze the above mentioned schemes for performance under various settings of time and memory complexities. We attempt to benchmark the said algorithms at similar levels of memory consumption.

Organization: The rest of the document is organised as follows. In Section 2 we present the related work and Section 3 covers a brief overview of three cache-timing resistant password hashing algorithms namely, Argon2i, Catena and Rig. This is followed by the preliminaries necessary for the understanding of the proposed technique in Section 4. The description of the proposed algorithm for TMTO is presented in Section 5. Subsequently, the re-computation penalties and performance analysis are presented in Section 6 and Section 7. Finally, in Section 8, we provide the conclusions of the paper.

2 Related Work 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 [9]. 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 [9] 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.

Utilizing the concept of Hellman [9], Philippe Oechslin introduced a cryptanalytic time-memory trade-off, named rainbow table [10]. 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 [9] is that the chains can collide within the same table without merging (to merge, collision at 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 password. A recent result on TMTO analysis based on precomputation method is covered in [11] for data-independent password hashing schemes and also provides a generic ranking method for data-dependent schemes. This proposed method is applied in [6] to prove the memory hardness of Argon2i. However, our analysis in Section 5 shows that Argon2i does not provide the claimed security level. Infact an attack showing the weak memory hardness of Argon2i was recently shown by Corrigan-Gibbs et al. [12].

3 Overview of the Analysed Password Hashing Algorithms

In this work we briefly explain three cache timing attack resistant password hashing algorithms submitted to Password Hashing Competition (PHC) [3]. Overview of these algorithms is provided ahead.

3.1 Argon2i [6]

Argon2 - Version 1.2.1 is the winner of PHC [3]. 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 [12]. 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 efficient for different use cases: Argon2d for computing cryptocurrencies and Argon2i for password hashing, key derivation etc. [6]. 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 [13] represented as $\mathcal{H}$ 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 $\mathcal{H}$ 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.

$$\begin{aligned}
M_{i,1} &\leftarrow G(H_0, i \parallel 0), && 0 \leq i < p\\
M_{i,2} &\leftarrow G(H_0, i \parallel 1), && 0 \leq i < p\\
M_{i,j} &= G(M_{i,j-1}, M_{\phi(i,j)}), && 0 \leq i < p, 2 \leq j < q\\
\text{Output} &\leftarrow \mathcal{H}(M_{0,q-1} \oplus M_{1,q-1} \oplus \cdots \oplus M_{p-1,q-1})
\end{aligned}$$

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 $l = J_2 \bmod p$ 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 $\mathcal{R}$ that is referenced for given $M_{i,j}$ according to the following rules as mentioned in [6]:

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. $J_1 \rightarrow |\mathcal{R}|(1 - \frac{(j_1)^2}{2^{64}})$, $x = (j_1)^2/2^{32}$, $y = (|\mathcal{R}| * x)/2^{32}$, $z = |\mathcal{R}| - 1 - y$. For detailed design of Argon, one may refer to [6].

3.2 Catena [7]

The design Catena provides two variants supporting input (password) independent memory access: Catena-Butterfly which is represented as a stack of double-butterfly graphs, and Catena-Dragonfly which is based on bit-reversal graphs. Catena uses a function H which implements Blake2b, a function H' which implements a single round of Blake2b including finalization, denoted as Blake2b-1, a randomization layer τ (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 λ 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. 1. 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 τ (optional) which depends on the public input γ. Finally, the third phase calls a memory-hard function F. When F is instantiated with ${BRH^g}_\lambda$ ((g, λ) - Bit Reversal Hashing) it is denoted as Catena-BRG and when F is instantiated with ${DBH^g}_\lambda$ ((g, λ) - Double Butterfly Hashing) it is denoted as Catena-DBG. The overview of the design applying the function flap is shown in Fig. 1.

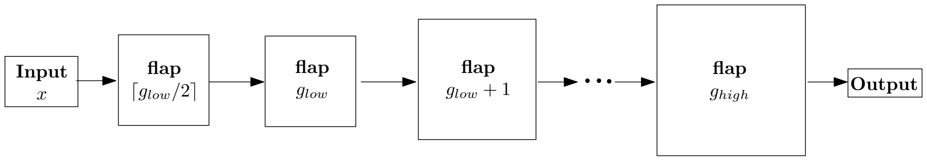
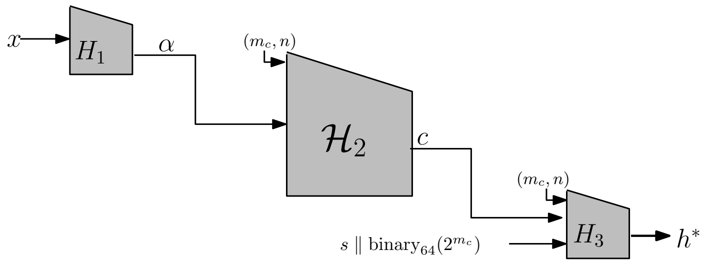


Fig. 1: General overview of the design Catena applying the function *flap*, taken from [7].

3.3 Rig [8]

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 α. 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 α 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 n-times where n 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 $n \times m$ computations of H_2 at iterative transformation phase altogether are denoted as function $\mathcal{H}_2$. 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. 2.



Input x = pwd $\parallel$ binary$_{64}$(pwd$_l$) $\parallel$ s $\parallel$ binary$_{64}$(s_l) $\parallel$ binary$_{64}$(n) $\parallel$ binary$_{64}$(l)
s = Salt, m_c=memory count, n = no. of iterations

Fig. 2: Overview of the design Rig taken from [8].

4 Preliminaries

In this Section we give a brief overview of the key concepts used in our proposed cryptanalysis technique.

4.1 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 $\mathcal{E} \subset V \times V$. 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.

4.2 Bit-Reversal Permutation [14]

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. 3.

4.3 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 is determined by the connectivity (dependency) between the nodes. We next describe

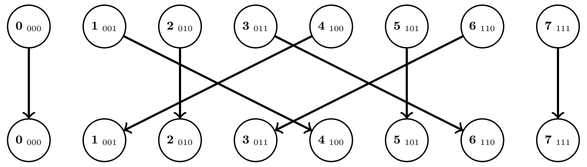


Fig. 3: Bit-Reversal Permutation of $m = 2^3$ elements with 3-bit binary representation of indices.

some graphs which are useful for analyzing the designs Catena [7] and Rig [8]. 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. 4.

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 [14] on each node (level-wise).
4. The *Butterfly* graph [15, 16] 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.

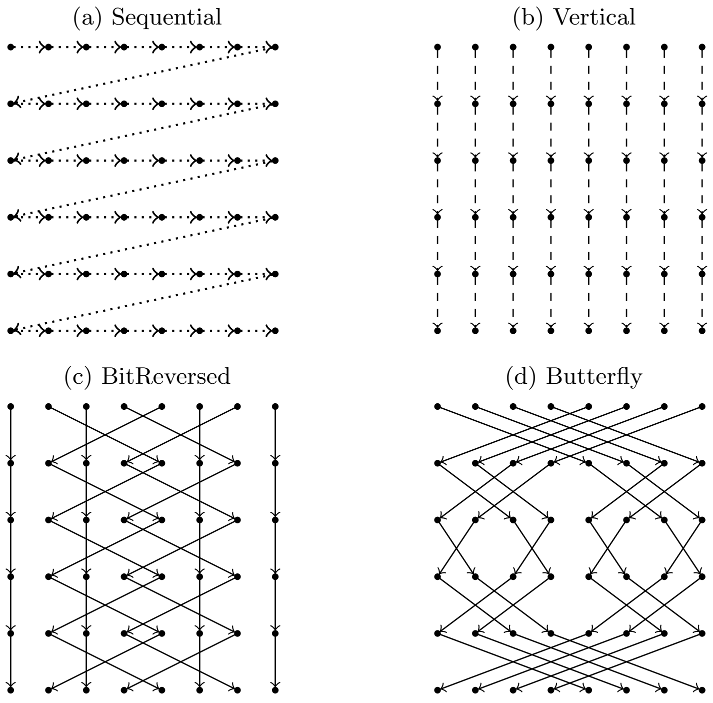


Fig. 4: Graphs based on different types of edges

$(\mathcal{N}, \lambda)$-Straight Graph A $(\mathcal{N}, \lambda)$-Straight Graph with $\mathcal{V}$ vertices and $\mathcal{E}$ edges can be formed by overlaying the *Sequential* and *Vertical* edge types graphs. λ denotes the depth of the graph and $\mathcal{N} = 2^k$ where, $k \in \mathbb{N}$ is the number of nodes at each layer. An example of $(8, 2)$-Straight graph

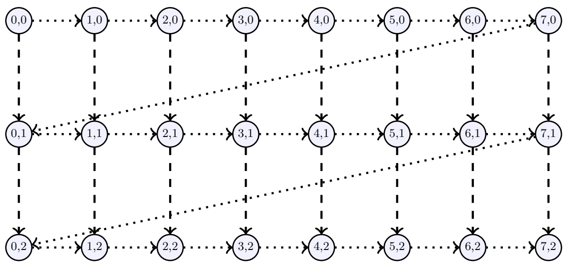


Fig. 5: (8,2)-Straight Graph

is shown in Fig. 5. 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 [7] and Rig [8].

$(\mathcal{N}, \lambda)$-Bit-Reversal Graph (Representing the Catena-BRG Construction [7]) A $(\mathcal{N}, \lambda)$-Bit-Reversal Graph with $\mathcal{V} = \mathcal{N}$ vertices and $\mathcal{E}$ 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 $\mathcal{N} = 2^k$ where, $k \in \mathbb{N}$ (definition adapted from [8]). An example of $(8, 2)$-Bit-Reversal graph is

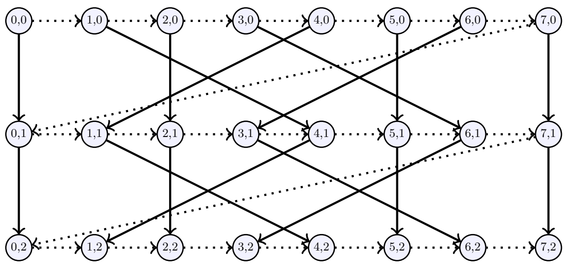


Fig. 6: (8,2)-Bit-Reversal Graph

shown in Fig. 6. 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 [7]. Specifically, it describes the flow of the flap function (see Fig. 1) with respect to its core memory-hard function F as described in section 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 λ 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 [8]) A $(\mathcal{N}, \lambda)$-Bit-Reversal-Straight Graph with $\mathcal{V} = \mathcal{N}$ vertices and $\mathcal{E}$ edges can be formed by overlaying the *Sequential*, *Vertical* and *BitReversed* edge types graphs. λ 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 [7]). An example of $(8, 2)$-Bit-Reversal-Straight graph is shown in Fig. 7 where $\lambda = 2$ is the depth of the graph and at each level, the number of nodes is 8.

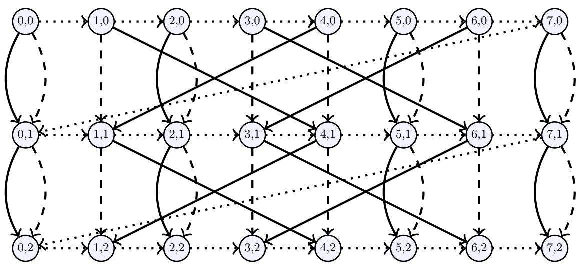


Fig. 7: (8,2)-Bit-Reversal-Straight Graph

This graph represents the directed data dependency of the $\mathcal{H}_2$ function of the Rig construction [8], depicted in Fig. 2. The function $\mathcal{H}_2$ 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 7, which is further explained in Section 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 $\mathcal{H}_2$ 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.

$(\mathcal{N}, \lambda)$-Double-Butterfly Graph (Representing the Catena-DBG Construction [7]) A $(\mathcal{N}, \lambda)$-Double-Butterfly Graph with $\mathcal{V} = \mathcal{N}$ vertices and $\mathcal{E}$ edges can be formed by overlaying the *Sequential*, *Vertical* and *Butterfly* edge type graphs. λ is the depth of the graph and at each level the number of nodes $\mathcal{N} = 2^k$ where, $k \in \mathbb{N}$ (definition adapted from [7]). An example of $(8, 1)$-Double-Butterfly graph is shown in Fig. 8 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 [7]. It describes the flow of the flap function (see Fig. 1) 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 λ which defines the depth of the graph. Specifically, the $(\mathcal{N} = 2^g, 1)$-Double-Butterfly graph representation is stacked λ times in Catena-DBG to create $(\mathcal{N} = 2^g, \lambda)$-Double-Butterfly graph. The memory traversal pattern follows from the original FFT

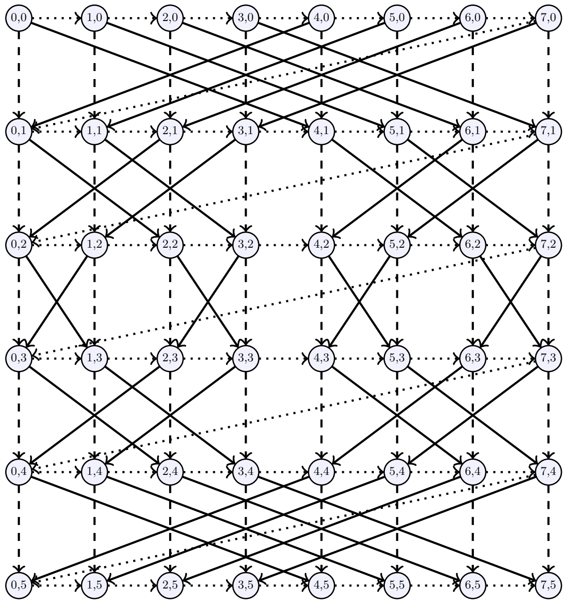


Fig. 8: $(8, 1)$-Double-Butterfly Graph

butterfly structure. Due to the significantly large number of operations and large number of layers (for example, 5 in Fig. 8) 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 are 8 (see Fig. 8). The directed acyclic graph corresponding to Catena-DBG has 2^g nodes which are arranged as λ stacks of double butterfly graphs. The directed edges in the DAG show the flow of execution of the algorithm as explained in Section 3.2.

5 Traversing a Dependency Graph to Analyse Tradeoff 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 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 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 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.

```
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 significantly large number of operations to compute the values that are not stored. We explain the proposed technique with examples in the following Section.

5.1 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.

```
Algorithm 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];    ▷ 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()    ▷ clear graph to process next dependencies
       31.     end if
       32.    else break    ▷ when no dependency is left to process
       33.    end while
       34. return list⟨Node⟩ traverse
```

$$(N^i, N^j) \rightarrow \{\ (D^i, D^j),(D^{i+1}, D^{j+2}), \dots \} \Rightarrow (N^{i+1}, N^{j+1}) \rightarrow \dots$$

Where, (N^i,N^j), ((N^{i+1},N^{j+1})) ... are the nodes, (D^i,D^j),(D^{i+1},D^{j+2}), ... 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 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 better understanding of the proposed approach we provide two examples which cover two different scenarios and also prove the validity of the method.

Example 1 A $(4, 2)$-Bit-Reversal Graph is shown in Fig. 9.

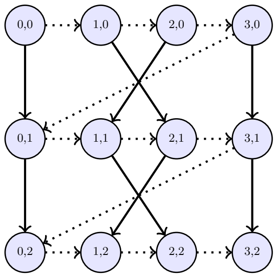


Fig. 9: $(4, 2)$-Bit-Reversal Graph

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. 9, 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.

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

Table 1: Traversal for Example 1

Position	Dependency
(3,2)	(2,2), (3,1)
(3,1)	(2,1), (3,0)
(3,0)	(2,0)
(2,0)	(1,0)
(1,0)	(0,0)
(0,0)	
(2,1)	(1,1)
(1,1)	(0,1)
(0,1)	
(2,2)	(1,2)
(1,2)	(0,2)
(0,2)	

$(1,0)$ 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) → (3,1) → (3,0) → (2,0) → (1,0) → (0,0) →
(2,1) → (1,1) → (0,1) → (2,2) → (1,2) → (0,2)

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

Example 2 A $(4,2)$-Bit-Reversal Graph is considered as shown in Fig. 10 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.

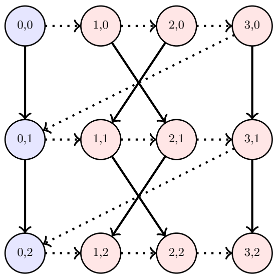


Fig. 10: $(4,2)$-Bit-Reversal Graph with only memory in the first column.

The complete traversal for a $(4,2)$-Bit-Reversal Graph as shown in Fig. 10 needs 35 steps and the steps take the following path.

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

$\Rightarrow$ (2,1) $\rightarrow$ {(1,1), (1,0)} $\Rightarrow$ (1,0) $\rightarrow$ {}
$\Rightarrow$ (1,1) $\rightarrow$ {(0,1), (2,0)} $\Rightarrow$ (2,0) $\rightarrow$ {(1,0)} $\Rightarrow$ (1,0) $\rightarrow$ {}
$\Rightarrow$ (0,1) $\rightarrow$ {(3,0)} $\Rightarrow$ (3,0) $\rightarrow$ {(2,0)} $\Rightarrow$ (2,0) $\rightarrow$ {(1,0)} $\Rightarrow$ (1,0) $\rightarrow$ {}
$\Rightarrow$ (2,2) $\rightarrow$ {(1,2), (1,1)} $\Rightarrow$ (1,1) $\rightarrow$ {(2,0)} $\Rightarrow$ (2,0) $\rightarrow$ {(1,0)} $\Rightarrow$ (1,0) $\rightarrow$ {}
$\Rightarrow$ (1,2) $\rightarrow$ {(0,2), (2,1)} $\Rightarrow$ (2,1) $\rightarrow$ {(1,1), (1,0) } $\Rightarrow$ (1,0) $\rightarrow$ {}
$\Rightarrow$ (1,1) $\rightarrow$ {(2,0)} $\Rightarrow$ (2,0) $\rightarrow$ {(1,0) } $\Rightarrow$ (1,0) $\rightarrow$ {}
$\Rightarrow$ (0,2) $\rightarrow$ {(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) $\rightarrow$ {}
$\Rightarrow$ (2,1) $\rightarrow$ {(1,1), (1,0)} $\Rightarrow$ (1,0) $\rightarrow$ {}
$\Rightarrow$ (1,1) $\rightarrow$ {(2,0)} $\Rightarrow$ (2,0) $\rightarrow$ {(1,0) } $\Rightarrow$ (1,0) $\rightarrow$ {}

6 Results

The DAG traversal algorithm provided in Section 5 can be used to compute the re-computation penalties for any graph by varying the memory storage. To apply Algorithm 1, a large number of memory configuration are 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 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 6.1. Following are the graphs that represent a fixed structure for all parameter choices and we compute the re-computation penalties for them.

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

The cumulated results (including Argon2i) for the graphs are as shown in Fig. 11. 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 2, 3, 4 and 6 are included in Fig. 11.

Table 5: Re-computation Penalties for Double-Butterfly Graph ($\lambda = 1$)

Memory Proportion (%)	Double-Butterfly Graph
50	18
25	922
12.50	60504
6.25	2043702

Fig. 11: Re-computation Penalties for Graphs

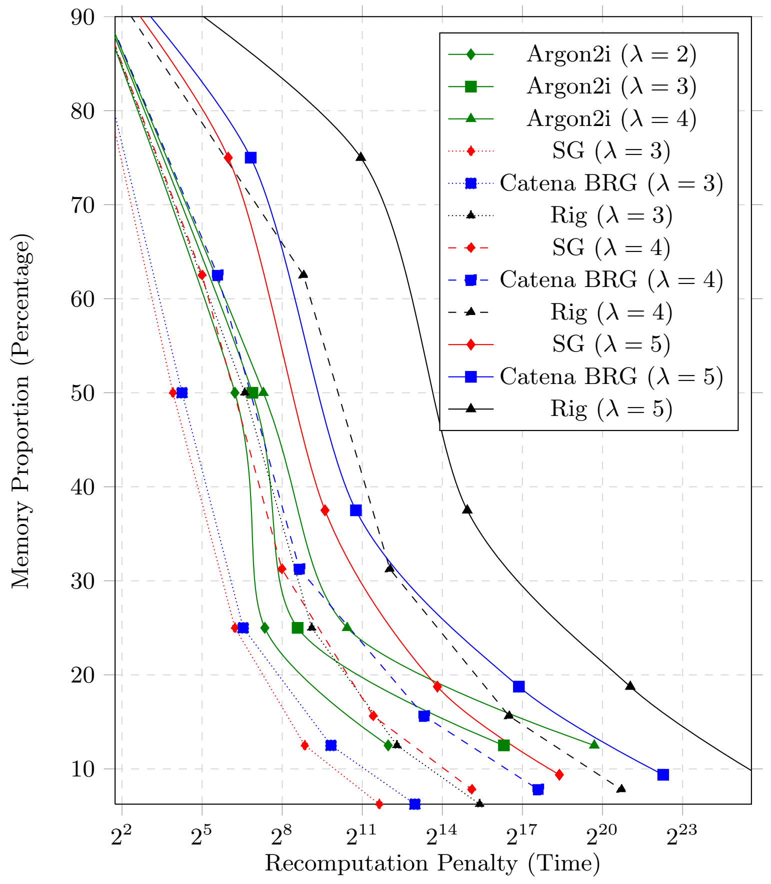


Table 2: Re-computation Penalties for Graphs with 4 rows ($\lambda = 3$)

Memory (%) Proportion (%)	Straight Graph	Bit-Reversal Graph (Catena BRG)	Bit-Reversal-Straight Graph (Rig)
50	15	19	97
25	75	93	551
12.5	460	909	5036
6.25	3181	8019	43143

Table 3: Re-computation Penalties for Graphs with 5 rows ($\lambda = 4$)

Memory (%) Proportion (%)	Straight Graph	Bit-Reversal Graph	Bit-Reversal-Straight Graph
62.5	32	48	445
31.25	254	401	4190
15.625	2724	10215	92483
7.8125	35120	197389	1707950

Table 4: Re-computation Penalties for Graphs with 6 rows ($\lambda = 5$)

Memory (%) Proportion (%)	Straight Graph	Bit-Reversal Graph	Bit-Reversal-Straight Graph
75	63	113	1971
37.5	777	1736	31270
18.75	14378	119358	2152596
9.375	341447	5060331	-

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 large amount of memory even for huge re-computation dependency tree calculations.

6.1 Re-computation Penalty for Argon2i

The design Argon2 is the winner of the PHC competition [3] and an IETF internet-draft [17] 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 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 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 corresponding row of the memory matrix of Argon2i as described in Section 3.1. Except for the first two nodes of first level, all nodes are connected with 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. 12, 13 and 14. The parameters password, salt and p are taken to be "password", "salt" and 1, respectively. Note that $p = 1$ means we are considering the single threaded version of Argon2i.

A recent attack on Argon2i, presented by Corrigan-Gibbs et al. [12], shows that having a pseudorandom memory access pattern for Argon2i does not provide 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. 12. 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. 12). 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 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 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 vary 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 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 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 6.

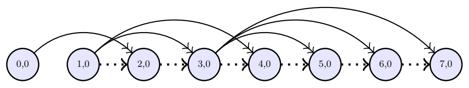


Fig. 12: Argon2i at t=1, m= 8 blocks

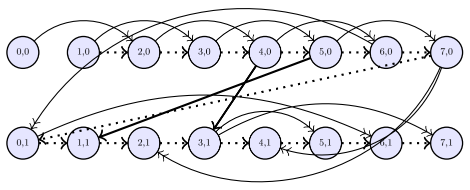


Fig. 13: Argon2i at t=2, m= 8 blocks

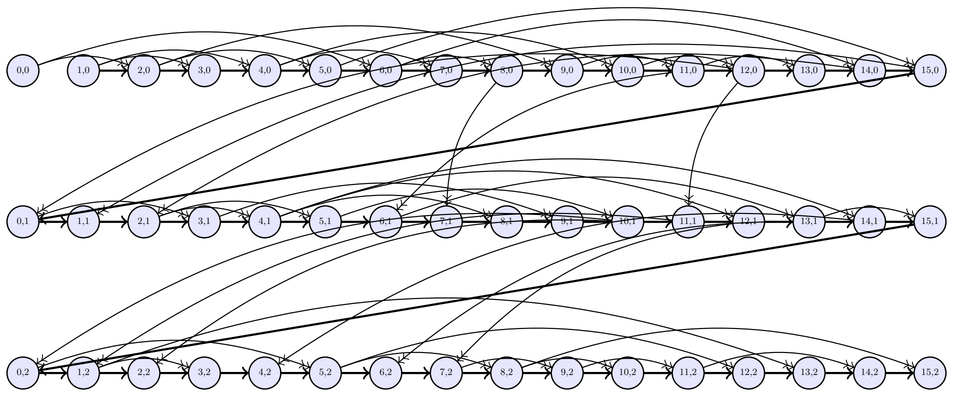


Fig. 14: Argon2i at t=3, m= 16 blocks

Table 6: Re-computation Penalties for Argon2i (32KiB) at varying λ (t-cost)

Memory Proportion (%)	$\lambda = 2$	$\lambda = 3$	$\lambda = 4$
50	75	118	157
25	163	382	1383
12.5	4008	80906	845846

Argon2i - Version 1.3 [18] This version was developed recently to mitigate the attack shown in [12] on the previous version which was the PHC winner. This is similar to the design explained in Section 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.

7 Performance Analysis

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.

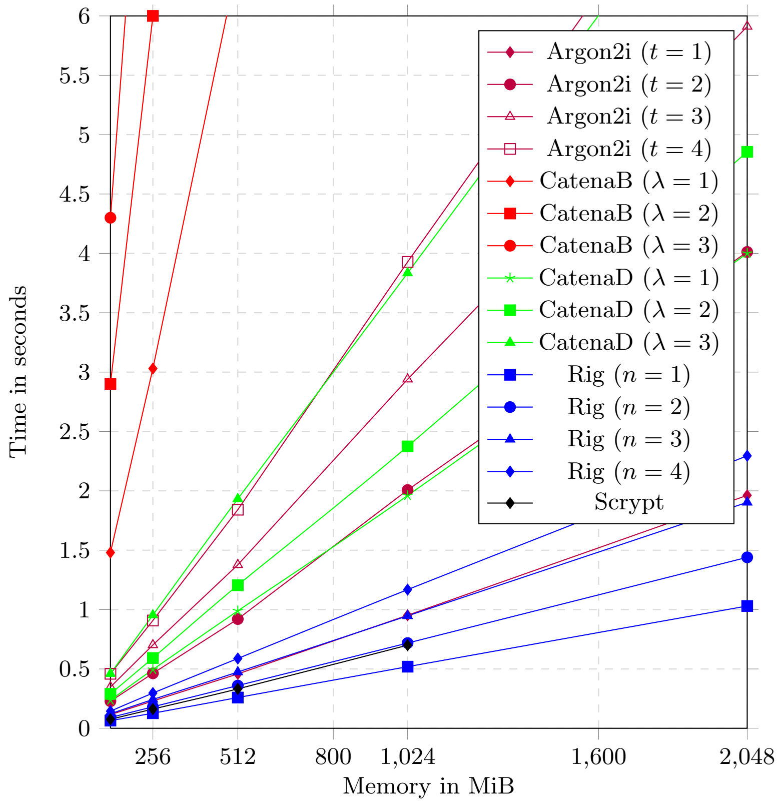


Fig. 15: Execution time vs. memory, fast feeling of memory is better.

For consistency we use single threaded versions of the algorithms. All the experiments were run at-least 5 times and the average timings were taken. The performance graph is shown in Fig. 15 which shows the execution time vs. memory for all algorithms benchmarked.

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 [11]. The latest version of code at the time of benchmark was cloned from [20] 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 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 [21]. 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 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' [4] 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 [22] with AVX2 implementation using Blake2b and Salsa64/8.

8 Conclusions

It is difficult to provide a general technique to analyse the time-memory tradeoff for memory hard designs. 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 proposed DAG traversal technique is flexible enough to be applied (may require minor simplification) on various other complex cryptographic designs for which making a mathematical model is significantly difficult. Our TMTO analysis shows that Argon2i does not follow the claimed memory hardness.

The performance graph in figure 15 shows the execution time vs. memory for all the memory-hard algorithms benchmarked. It is clear that Catena-Butterfly is the slowest and take 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-v2 provide good performance in a wide range of use cases, though Argon2i is slower than Rig. The attack of [12] on Argon2i shows reduction over claimed TMTO defense which is also visible in our analysis as shown in Table 6.

References

1. Jerome H. Saltzer. Protection and the control of information sharing in MULTICS. In *Proceedings of the Fourth Symposium on Operating System Principles, SOSP 1973, Thomas J. Watson, Research Center, Yorktown Heights, New York, USA, October 15-17, 1973.*
2. Robert Morris and Ken Thompson. Password Security: A Case History , 1979. `http://cs-www.cs.yale.edu/homes/arvind/cs422/doc/unix-sec.pdf`.
3. Password Hashing Competition (PHC), 2014. `https://password-hashing.net/#phc`.
4. Colin Percival. Stronger key derivation via sequential memory-hard functions. In *BSDCon*, 2009. `http://www.bsdcan.org/2009/schedule/attachments/87_scrypt.pdf`.
5. Christian Forler, Stefan Lucks, and Jakob Wenzel. Catena: A memory-consuming password scrambler. *IACR Cryptology ePrint Archive*, 2013:525, 2013.
6. Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. Argon2: the memory-hard function for password hashing and other applications . Submission to Password Hashing Competition (PHC), 2015. https://password-hashing.net/argon2-specs.pdf.

7. Christian Forler, Stefan Lucks, and Jakob Wenzel. The Catena Password-Scrambling Framework. Submission to PHC, 2015. `https://www.uni-weimar.de/fileadmin/user/fak/medien/professuren/Mediensicherheit/Research/Publications/catena-v3.3.pdf`.
8. Donghoon Chang, Arpan Jati, Sweta Mishra, and Somitra Kumar Sanadhya. Rig: A simple, secure and flexible design for password hashing. In *Information Security and Cryptology - 10th International Conference, Inscrypt 2014, Beijing, China, December 13-15, 2014, Revised Selected Papers*, pages 361–381, 2014.
9. Martin E. Hellman. A cryptanalytic time-memory trade-off. *Information Theory, IEEE Transactions on*, 26(4):401–406, 1980.
10. Philippe Oechslin. Making a faster cryptanalytic time-memory trade-off. In *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, pages 617–630, 2003.
11. Alex Biryukov and Dmitry Khovratovich. Tradeoff cryptanalysis of memory-hard functions. In *Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part II*, pages 633–657, 2015.
12. Henry Corrigan-Gibbs and Dan Boneh and Stuart Schechter. Balloon Hashing: Provably Space-Hard Hash Functions with Data-Independent Access Patterns . *IACR Cryptology ePrint Archive*, 2016.
13. Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. BLAKE2: simpler, smaller, fast as MD5. *IACR Cryptology ePrint Archive*, 2013:322, 2013.
14. Thomas Lengauer and Robert Endre Tarjan. Upper and lower bounds on time-space tradeoffs. In *Proceedings of the 11h Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1979, Atlanta, Georgia, USA*, pages 262–277, 1979.
15. James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.
16. William F. Bradley. Superconcentration on a pair of butterflies. *arXiv preprint arXiv:1401.7263*, 2014.
17. A. Biryukov, D. Dinu, D. Khovratovich, and S. Josefsson. The memory-hard Argon2 password hash and proof-of-work function . IRTF Crypto Forum Research Group Active Internet-Draft (cfrg RG): draft-irtf-cfrg-argon2-00, 2016. `https://datatracker.ietf.org/doc/draft-irtf-cfrg-argon2/`.
18. Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. Argon2: the memory-hard function for password hashing and other applications , Version 1.3, 2016. `https://www.cryptolux.org/images/0/0d/Argon2.pdf`.
19. Memory-hard scheme Argon2, 2016. https://github.com/khovratovich/Argon2 (162 commits).
20. Reference Implementation of Catena, a memory-consuming password scrambler, 2015. https://github.com/medsec/catena (75 commits).
21. Reference and Optimized implementations of Rig, A simple, secure and flexible design for Password Hashing, 2016. https://github.com/arpanj/Rig.
22. A flexible implementation of Colin Percival's scrypt, 2016. https://github.com/floodyberry/scrypt-jane (58 commits).