Rig: A simple, secure and flexible design for Password Hashing

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

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

Abstract. Password Hashing, a technique commonly implemented by a server to protect passwords of clients, by performing a one-way transformation on the password, turning it into another string called the hashed password. In this paper, we introduce a secure password hashing framework *Rig* which is based on secure cryptographic hash functions. It provides the flexibility to choose different functions for different phases of the construction. The design of the scheme is very simple to implement in software and is flexible as the memory parameter is independent of time parameter (no actual time and memory trade-off) and is strictly sequential (difficult to parallelize) with comparatively huge memory consumption that provides strong resistance against attackers using multiple processing units. It supports client-independent updates, i.e., the server can increase the security parameters by updating the existing password hashes without knowing the password. *Rig* can also support the server relief protocol where the client bears the maximum effort to compute the password hash, while there is minimal effort at the server side. We analyze *Rig* and show that our proposal provides an exponential time complexity against the low-memory attack.

Keywords: Password, Password hashing, GPU attack, Cache-timing attack, Client-independent update, Server-relief technique

1 Introduction

A password is a secret word or string of characters which is used by a principal to prove her identity as an authentic user to gain access to a resource. Being secret, passwords cannot be revealed to other users of the same system. In order to ensure the confidentiality of the passwords even when the authentication data is somehow leaked from the server, passwords are never stored in clear, but transformed into an illegible form and then stored. Specifically, 'Password Hashing' is the technique which performs a one-way transformation on a password and turns it into another string, called the 'hashed' password. Strong password protection, i.e., a technique of password hashing that makes brute force attack on password guessing infeasible, either in software or by using GPUs (Graphics Processing Unit), is essential to protect the user security and identity. Thus any working password hashing scheme should be resistant to brute force attack.

Password hashing is an active topic of interest in cryptography community and a competition on password hashing is going on [1]. Currently, the significant constructions for password hashing are PBKDF2 [12], Bcrypt [14] and Scrypt [13]. All of these do not satisfy most of the necessary requirements mentioned at the competition page [1]. PBKDF2 (NIST standard) consumes very less memory as it was mainly designed to derive keys from a seed (password). Bcrypt uses fixed memory (4KB) for its implementation. Scrypt is not simple (different internal modules) and not flexible (time and memory parameters are dependent) and susceptible to cache timing attack (discussed in section 5).

Specifically, the rate at which an attacker can guess passwords is a key factor in determining the strength of the password hashing scheme. Current requirements [1] for a secure password hashing scheme are the following:

- The construction should be slow to resist password guessing attack but should have a fast response time to prove the authenticity of the user.
- It should have a simple design and should be easy to implement (coding, testing, debugging, integration), i.e., the algorithm should be simple in the sense of clarity and concise with less number of internal components and primitives.
- It should be flexible and scalable, i.e., if memory and time are not dependent then one would be able to scale any of the parameters to get required performance.
- Cryptographic security [1]: The construction should behave as a random function (random-looking output, one-way, collision resistant, immune to length extension, etc.).
- Resistant to GPU attack: A typical GPU has lots of processing cores but has limited amount of memory for each single core. It is quite efficient for an attacker to utilize all the available processing cores with limited memory to run brute-force attack over the password choices. Use of comparatively huge memory per password hash by the password hashing construction can restrict the use of GPU. Therefore, the design should have large memory consumption to force comparatively slow and costly hardware implementation that can resist the GPU attack.
- Leakage Resilience: The construction should protect against information extraction from physical implementation, i.e., the scheme should not leak information about the password due to cache timing or memory leakage, while supporting any length of password.
- The construction should have the ability to transform an existing hash to a different cost setting (client independent update, explained in section 5) without knowledge of the password.
- It is good if the construction provides server relief technique where the client performs most of the computations for password hashing and the server puts minimal effort with minimal use of resources, to reduce the load of the server. This property needs a secure protocol to maintain the security of the hash computation (discussed in section 5).

The most challenging threat faced by any password hashing scheme is the existence of cheap, massively parallel hardware such as Graphics Processing Units (GPUs), Application-Specific Integrated Circuits (ASICs) and Field-Programmable Gate Arrays (FPGAs). Using such efficient hardware, an adversary with multiple computing units can easily try multiple different passwords in parallel. To prevent such attempts we need to slow down password hash computation and ensure that there is little parallelism in the design. One way to achieve this is to use a 'Sequential memory-hard' algorithm, a term first introduced with the design of 'Scrypt' [13], a password hashing scheme. The main design principle of Scrypt is that it asymptotically uses almost as many memory locations as it uses operations to slow down the password-hash computation. Memory is *relatively* expensive, so, a typical GPU or other cheap massively-parallel hardware with lots of cores can only have a limited amount of memory for each single core. Hence an attacker with access to such hardware will still not be able to utilize all the available processing cores due to the lack of sufficient memory and will be forced to have an (almost) sequential implementation of the password hashing scheme.

In this document we propose *Rig*, a password hashing scheme which aims to address the above mentioned requirements. *Rig* is based on cryptographic (secure) hash functions and is very simple to implement in software. It is flexible as the memory parameter is independent of time parameter (no actual time and memory trade-off) and is strictly sequential (difficult to parallelize) with comparatively huge memory consumption that provides strong resistance against attackers using multiple processing units. It supports client-independent password hash up-gradation without the need of the actual password. This feature helps the server to increase the security parameters to calculate the password hash to reduce the constant threats of technological improvements, specifically in the field of hardware. *Rig* provides protection against the extraction of information from cache-timing attack and prevents denial-of-service attack if implemented to provide server-relief technique. We analyze *Rig* and show that our proposal provides an exponential time complexity against memory-free attack. It gives the flexibility to choose different functions for different phases of the construction and we denote the general construction of *Rig* as Rig [H_1, H_2, H_3]. In this work we provide two variants of Rig [H_1, H_2, H_3]. A strictly sequential variant, Rig [Blake2b, BlakeCompress, Blake2b] and the other variant, Rig [BlakeExpand, BlakePerm, Blake2b] which improves the performance by performing memory operations in larger chunks.

The rest of the document is organised as follows. In section 2 we present the important preliminaries necessary for understanding the specification. This is followed by the introduction of significant hardwares used as attack platform in section 3. The specification and design rationale of the scheme are presented in sections 4 and 5 respectively. Subsequently, the implementation aspects and performance analysis are presented in sections 6 and 7. Finally, in sections 8 and 9, we provide the security analysis of the scheme and the conclusions of the paper respectively.

2 Preliminaries

The techniques used in our construction are discussed below.

- **Binary 64-bit mapping:** It is a 64-bit binary representation of the decimal value. The binary number

$$a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + \cdots + a_0$$

  is denoted as $a_{n-1}a_{n-2}\cdots a_0$ where $a_i \in \{0, 1\}$ and n is the number of digits to the left of the binary (radix) point. In our construction we use $n = 64$ and we denote $\text{binary}_{64}(x)$ for 64-bit binary representation of the value x.
- **Bit reversal permutation [9,11] (br):** It is implemented to permute the indices of an array of $n = 2^k$ elements where $k \in \mathbb{N}$. We explain the steps of the permutation through Algoritm 1 below.
  The example of a bit reversal permutation applied on an array of $m = 2^3$ elements where $k = 3$ and indices are $0, 1, \cdots, 7$ is given below.
  $br[000, 001, 010, 011, 100, 101, 110, 111] = [000, 100, 010, 110, 001, 101, 011, 111]$
  $= br[0], br[1], br[2], br[3], \cdots, br[7]$.

Algorithm 1: Bit reversal permutation (br)

Input: Indices of an array A of $n = 2^k$ elements where $k \in \mathbb{N}$ and indices are: $0, 1, 2, \cdots, n-1$.

Output: Permuted indices of array A as: $br[0], br[1], br[2], \cdots, br[n-1]$

1. for $i = 0$ to $n - 1$
2. $\quad (i)_{bin_k} = i_{k-1}i_{k-2}\cdots i_1 i_0 = \sum_{j=0}^{k-1} 2^j i_j$
3. $\quad \triangleright\ (i)_{bin_k} = k$-bit binary representation of value i
4. $\quad br[i] = \sum_{j=0}^{k-1} 2^j i_{k-1-j}$
5. return $br[0], br[1], br[2], \cdots, br[n-1]$

3 Attack platforms: Significant hardwares

According to Moore's Law [15], the number of transistors on integrated circuits doubles approximately every two years. This has indeed been the case over the history of computing hardware. Following this law, hardware is becoming more and more powerful with time. This happens to be the most prominent threat for existing password hashing schemes. Consequently, there is a need to raise the cost of brute force attack by controlling the performance of the massively parallel hardware available.

An important electronic circuit, **Graphics Processing Unit** (GPU), and their highly parallel structure makes them more effective than general-purpose CPUs for algorithms where processing of large blocks of data is done in parallel. An **Application-Specific Integrated Circuit** (ASIC), is an integrated circuit (IC) which can be customized with memory chips to implement a dedicated design. An ASIC can not be altered after final design hence the designers need to be certain of their design when it is implemented in ASIC. On the other hand,

Field Programmable Gate Arrays (FPGAs) are programmable integrated circuits and consist of an array of logic elements together with an interconnected network and memory chips, providing high-performance. A designer can test her design on an FPGA before implementing it on an ASIC.

Both ASIC and FPGAs can be configured to perform password hashing with highly optimized performance. The cost of implementation on FPGA is cheaper than ASICs if the number of units of the hardware required is small. Therefore, one can easily use parallel FPGAs to increase the rate of password guessing. RIVYERA FPGA cluster is an example of a very powerful and cost optimized hardware. It can hash 3,56,352 passwords per second by using PBKDF2 (NIST standard, Password Based Key Derivation Function 2) with SHA-512 and 512-bit derived key length [8]. This high performance is possible on the FPGA because PBKDF2 does not consume high memory for password hashing. Comparing FPGAs with GPUs (Graphics processing units), the authors of [8] provide results of the same implementation on 4 Tesla C2070 GPUs as 1,05,351 passwords per second. ASIC is better than FPGA purely on performance in terms of number of hashes per second. However, FPGA is preferable when cost is considered with the speed of hashing. Following Moore's law, the speed of hardware is likely to increase by almost a factor of two in less than two years. However, as processor speeds continue to outpace memory speeds [10], the gap between processor and memory performance increases by about 50 % per year [5]. Thus, there is a need to minimize the effects of such high performance hardware. Hence, we need a password hashing algorithm which consumes comparatively large memory to prevent parallel implementation.

4 Specification

Our construction is described in Fig. 1. Following is the step-by-step description of Algorithm 2 which explains our construction *Rig*.

1. First we need to fix the following parameters:
   - pwd = The user password of any length.
   - s = The salt value of any length.
   - n= The number of iterations required to perform iterative transformation phase.
   - m_c= The memory count from which the memory-cost is defined as: $m = 2^{m_c}$, i.e., m denotes the number of items to be stored in the memory. The value of m is updated as: $m_{i+1} = 2 \times m_i$ at each round.
   - r= The number of rounds for the setup phase followed by iterative transformation phase and output generation phase.
   - l= The output length of the password hash.
   - t= The number of bits retained from hash output after truncation. Used with a function $\text{trunc}_t(x) = x \gg (|x| - t)$, where x is the hash output.
2. **Initialization phase:** We map the parameters, namely the values: password length pwd_l, salt length s_l, n and the output length, l to a 64-bit binary value using binary_{64} mapping. We create the value x as the concatenation ($||$) of the above mentioned parameters as

$$x = \text{pwd} \parallel \text{binary}_{64}(pwd_l) \parallel s \parallel \text{binary}_{64}(s_l) \parallel \text{binary}_{64}(n) \parallel \text{binary}_{64}(l)$$

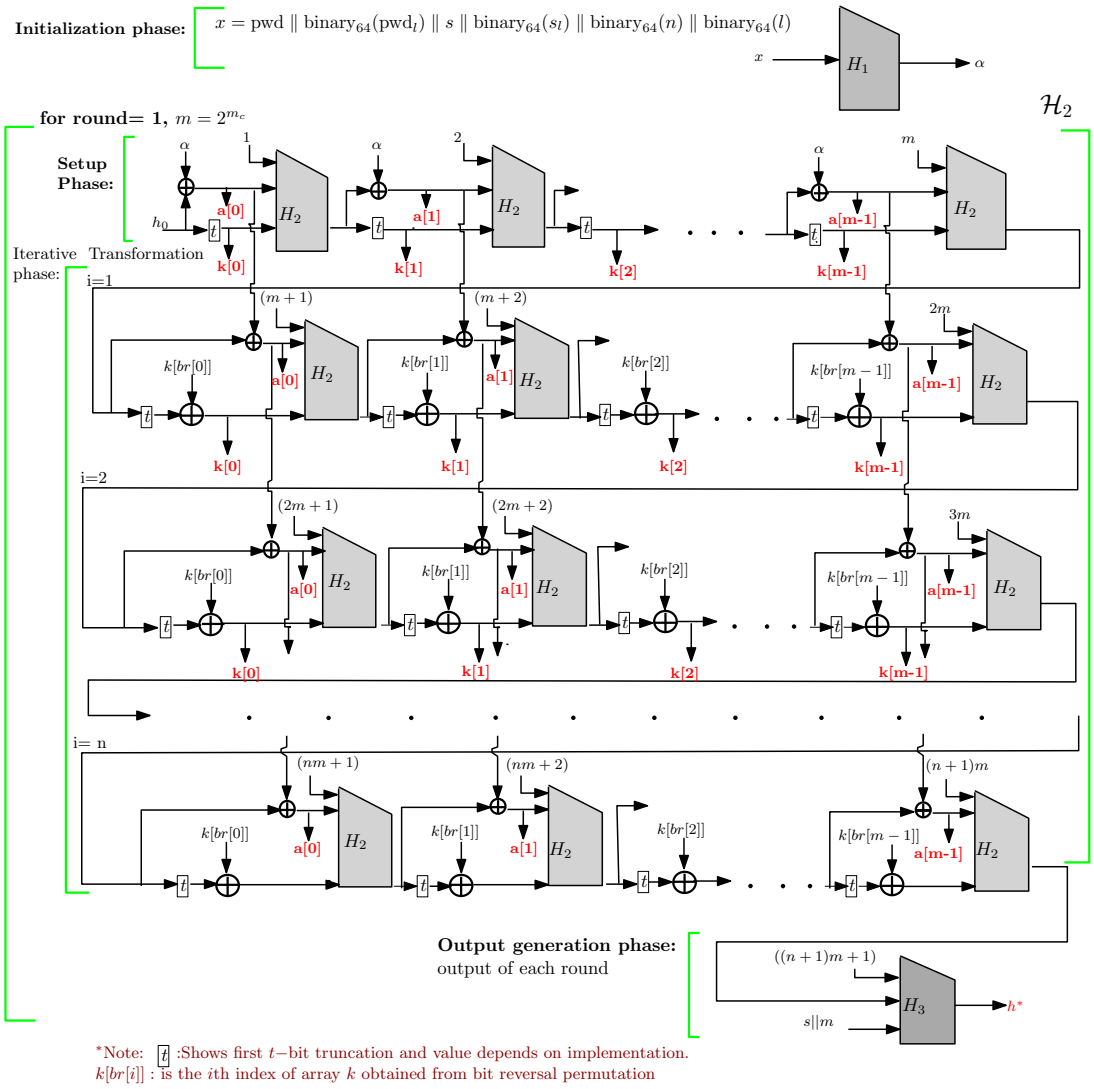


Fig. 1: Graphical representation of the proposed construction.

Rig: A password hashing scheme

Algorithm 2: Rig [H_1, H_2, H_3] Construction

Input: Password (pwd), Password length (pwd_l), Salt (s), Salt length (s_l), No. of iterations (n), Memory count (m_c), No. of bits to be retained from hash output of the setup phase (t), Output length (l), No. of rounds (r).

Output: l-bit hash value $h_r{}^*$ obtained after r rounds

▷ Initialization phase: generates α from password
Initialize: a random salt (s) of atleast 16-bytes, number of iterations (n), value of memory count m_c where $m = 2^{m_c}$, value t
$x =$ pwd $\parallel$ binary$_{64}(pwd_l) \parallel s \parallel$ binary$_{64}(s_l) \parallel$ binary$_{64}(n) \parallel$ binary$_{64}(l)$ ▷ concatenation: $\parallel$
$\alpha = H_1(x)$ ▷ H_1 : underlying hash function
for round 1 to r
  ▷ Initialization of Setup phase: Creates two arrays k and a where $|k| = |a| = m$ where $m = 2^{(round-1)} \times 2^{m_c}$
  $h_0 =$ initialized with the value of π after decimal, and $|h_0| = |\alpha|$.
  $a[0] = \alpha \oplus h_0$, $k[0] = trunc_t(h_0)$
  for $i = 1$ to m
    $h_i = H_2(i \parallel a[i-1] \parallel k[i-1])$ ▷ H_2 : underlying hash function
    if $i \neq m$
      $a[i] = \alpha \oplus h_i$
      $k[i] = trunc_t(h_i)$ ▷ retains the first $t-$bits of the hash output
  ▷ Initialization of Iterative Transformation phase
  for $i = 1$ to n
    for $j = 1$ to m
      $a[j-1] = a[j-1] \oplus h_{\{im+j-1\}}$
      $br[j-1] =$ index value of array k obtained using bit reversal permutation
      ▷ initialize a temporary array $|k_{\text{temp}}| = |k|$
      $k_{\text{temp}}[j-1] = k[br[j-1]] \oplus \text{trunc}_t(h_{im+j-1})$
      $h_{im+j} = H_2((im+j) \parallel a[j-1] \parallel k_{\text{temp}}[j-1])$
    $k = k_{\text{temp}}$
  ▷ Output generation phase
  $h_{round}{}^* = (H_3((n+1)m+1) \parallel h_{(n+1)m} \parallel s \parallel \text{binary}_{64}(m))$
  if round $< r$
    $\alpha = h_{round}{}^*$

and compute $H_1(x) = \alpha$ where H_1 is the underlying hash function. We use α for further calculations in the setup phase.

3. **Setup phase** We initialize h_0 with the value of π after the decimal point. We take as many digits of π as desired to ensure that $|h_0| = |\alpha|$. The values h_0 and α are used to initialize two arrays k and a and further $m-1$ values of the arrays are iteratively calculated as shown in the fig. 1. First t-bits of each hash output are stored in the array k.
   The large number of calls to the underlying hash function are guaranteed to have different inputs by the use of different counter values. H_2 denotes the underlying hash function.
4. **Iterative transformation phase** This phase is designed to make constant use of the stored array values and to update them. Here we modify each

element of the arrays a and k, n-times where n is the number of iterations. Array a is accessed sequentially where values of array k are accessed using bit reversal permutation explained in Algorithm 1. We denote the index of array k obtained applying bit-reversal permutation as: $br[j]$, $0 \leq j \leq m-1$.

5. **Output generation phase** After execution of the setup phase and iterative transformation phase sequentially, we calculate one more hash, denoted by H_3 to get the output of each round. If round=1, this output is considered as the password hash.
   Note: The output is an l-bit value. The algorithm stores the output as the hashed password. Our construction allows for storing a truncated portion of the hash output as well. If this is desired we can take one of the following two approaches.
   (a) The user may run the complete algorithm as described above and truncate the final output after r rounds to the desired length. This approach does not support client-independent update.
   (b) To support client-independent updates the user can choose a length for truncation which is sufficient to claim brute-force security. Then append some constant value, we suggest the hexadecimal value of π after first 64-bytes of decimal point. Take as many digits as desired to make the output length of each round equal to the length of α of the setup phase. So this way one can reduce the storage requirement for password hashes at the server.

5 Design rationale

Existing password hashing schemes are not simple and do not fulfill the necessary requirements as discussed in section 1. We have tried to design a solution which overcomes the known disadvantages of existing schemes (PBKDF2 [12], Bcrypt [14] and Scrypt [13]). The primary concerns against existing proposals are their complex design and their inefficiency to prevent hardware threats. We have tried to strengthen our design by considering the necessary requirements as mentioned in section 1.

1. **Initialization phase** We have used concatenation of password, salt, 64-bit value of pwd_l, s_l, n and l as input to increase the size of input. This resists brute force dictionary attack.
2. **Setup phase** In this phase we initialize h_0 with π, as we want to have a random sequence and π is not known to have any pattern in the sequence of digits after the decimal point. We generate the values that are required to be stored and repeatedly accessed throughout the remaining phases. This ensures that a large memory requirement criteria for a password hashing scheme is satisfied which neutralizes the threat of using recent technological trends, such as GPUs, ASICs etc. We use different counter values for each hash computation to make all hash inputs different. This reduces collisions and hence makes the output different.

For array k there is a flexibility to vary the bit storage by taking first t-bits of the hash output where t is taken to be close to the hash-length but not equal to the hash-length. This fulfills the demand of huge memory while at the same time ensures sequential hash calculation and forces an attacker to compute the hash at run-time thus slowing him down. Further, it also allows to extend the scope of implementation in that a low memory device may keep very few bits of the hash values stored but may increase the number of iterations. This ensures that *Rig* can be implemented in resource constrained devices.

3. **Iterative transformation phase** To make the storage requirement compulsary, this phase progresses sequentially, accessing and updating all stored values at each iteration. Here again, we use different counters for hash input for the same reason as mentioned above. In this phase the memory access pattern is made password independent to reduce the chance of cache timing attack which we have explained later in this section.
4. **Output generation phase** This is the last phase of each round. We reuse the salt value as input to make the collision attack difficult.
   Apart from that the output of each round can be truncated to a desired length. This is optionally mentioned to handle the situations when it is required to reduce the server storage per password.

The other important criteria taken into account in the design of the scheme are the following:

5. **Simplicity and flexibility** Symmetry (as setup phase and iterative transformation phase follows similar structure) in the design of *Rig* enhances the overall clarity of the scheme. An earlier password hashing scheme Scrypt [13] uses PBKDF2 (internally calls HMAC-SHA256) and ROMix (internally calls BlockMix and uses Salsa20/8). Unlike Scrypt, *Rig* uses only a single primitive (a cryptographically secure hash function). This makes our scheme easier to understand and easy to implement (coding, testing, debugging and integration).
   In our scheme the memory parameter is independent of the time parameter. This flexibility in design choice allows a user to scale any of these parameters to get the required performance. On the other hand we have the flexibility for the choice of the functions H_1, H_2 and H_3 (see Fig. 1), but proper selection of the primitives are required to maintain the overall design properties and security. Therefore there can be multiple variants of *Rig* aimed at different implementations or scenarios.
6. **Random output** Our scheme calls a hash function repeatedly. We use different counters for each of these hash calls to ensure that no input to the hash function is repeated. The security of *Rig* relies on the prevention of preimage and collision attacks against the underlying hash function. Use of any state-of-the-art hash function (e.g. any finalist of SHA-3 competition) ensures the security of our scheme. We use Blake2b [3] in demonstrating the performance of our scheme later in this paper, although any other hash function could easily be used instead.

With the property of different input, different output and same input, same output, our scheme mimics the Random Oracle Model. This provides theoretical justification of the security of *Rig*.

7. **Client-independent update [9]** Our design supports client independent update, i.e., server can increase the security parameter without knowing the password. This is possible if we fix the value of n (number of iterations) and increase the number of rounds r. Each round of the algorithm doubles the memory consumption m from the previous round and hence increases the security parameter. This is possible because the output of each round can be treated as the value α at the next round and then can easily follow the Algorithm 2 to produce the output of the next round. The idea of client independent update of the security parameter m is fulfilled by the following way. The value of m is updated at each next round $i + 1$ (say) from its previous round i as: $m_{i+1} = 2 \times m_i$.
   The overall procedure is: output of each round is the input to the next. Each round gives full hash maintaining all requirements of a good password hashing technique. By increasing the number of rounds, the scheme increases the required memory and time hence increases the security parameter without the interference of the client.
8. **Resistance against cache-timing attack** In our construction, to access the memory which is stored in an array k, we use bit-reversal permutation, which is independent of the password used. If a password dependent permutation is used and if the array can be stored in the cache while accessing the values, an attacker can trace the access pattern observing the time difference in each access of the array index. This helps the adversary to filter the passwords that follows similar memory access pattern and to make a list of feasible passwords. Therefore, a password hashing scheme should have password-independent memory access patterns and to follow this requirement we use bit reversal permutation as in [9].
9. **Server-relief hashing** Current requirement of a password hashing technique is that it should be slow and should demand comparatively large memory to implement. But this requirement may put extra load on server. Therefore we need a protocol to divide the load between the client and the server. The idea is provided in [9] and our construction supports this property following the protocol as mentioned below:
   First the authentication server provides the salt to the client. The client performs the initialization phase, setup phase and iterative transformation phase (see Algorithm 2), and sends the end result to the server. The server computes the output generation phase and produces the final hash. This way we can easily reduce the load of the server.
   Note: In this case, an attacker acting as a client, can repeatedly send some random data without following the computations of the proposed algorithm to the server. Here, the attacker can easily get the access with a correct guess. But, the complexity of the random guess will be equivalent to the brute-force complexity i.e. 2^n, where n is the output length of the underlying hash function. Therefore this can not be a feasible attack.

6 Implementation aspects

This proposed construction for password hashing can be implemented efficiently on a wide range of processors. However, the same implementation will require huge number of computations if dedicated hardware such as ASIC or FPGA is used with limited memory.

Our design allows the flexibility to utilize less storage with increased number of calculations if we retain few bits of the intermediate hash computation after truncation and increase the number of iterations n. This way, *Rig* can be efficient on low memory devices.

We designed Rig to have a highly flexible structure. By changing the functions H_1, H_2 and H_3 (see Figure 1) we can completely change the overall design properties. From side channel resistance to GPU or ASIC/FPGA resistance, any property can be achieved by the proper selection of the above primitives. Therefore there can be multiple variants aimed at different implementations or scenarios. As mentioned before, we describe the general construction of *Rig* as Rig [H_1,H_2,H_3], where we can design/choose the functions H_1, H_2, H_3 for implementing different variants of *Rig*.
We have designed and implemented two versions of *Rig* as follows:

1. **Rig [Blake2b, BlakeCompress, Blake2b]** This variant is strictly sequential. Full Blake2b is used for H_1 and H_3 while the first round of the compression function of Blake2b is used for H_2 (and we call it *BlakeCompress*). We have removed the constants in the 'G' function of Blake2b as suggested by the Blake authors in [3] to improve the overall performance. This version does a large number of random reads and writes and as a result it is strictly bounded by memory latency.
2. **Rig [BlakeExpand, BlakePerm, Blake2b]** This variant is designed to improve the performance by performing memory operations in larger chunks. The functions H_1 and H_2 are parallelized internally and the idea of handling larger chunk size improves the performance significantly without changing the overall sequential nature and memory-hardness of *Rig*. It also makes this variant of *Rig* much more difficult to execute efficiently in GPUs and FPGA/ASIC (explained in sections 6.4 and 6.5). We implemented the functions H_1 as 'BlakeExpand' and H_2 as 'BlakePerm'. These functions are explained later. The function H_3 uses full Blake2b.

6.1 Design of Rig [Blake2b, BlakeCompress, Blake2b]

This strictly sequential variant follows the general construction of *Rig* as explained in section 4. The functions H_1 and H_3 implements Blake2b (full hash). The function H_2 is implemented using first round of Blake2b compression function.

6.1.1 Design of the BlakeCompress function In Figure 2 we graphically show the implementation of H_2 as BlakeCompress. H_2 takes 1024 bits input and

initializes the initial state of Blake2b with this input. Then the eight G-functions as defined in Blake2b are applied with an all zero input message of length 1024 bits. RF_i denotes the ith call to these eight G-functions. As we use the first round only, we show the round as RF_1 in Figure 2. The modified state after RF_1 is then split in two equal halves and xor'ed together to produce 512-bits of output. This choice of implementation is different from the actual Blake2b in many ways. The actual Blake2b construction, shown in Figure 3, has an initialization phase which initializes the starting state. A permutation of the message m^i is supplied as the input to each RF_i for round $i = 1$ to 12. After 12 rounds, the finalization phase performs feedforward xor'ing with output of RF_{12}. This preserves the onewayness of the Blake2b function. This feedforward xor'ing is omitted in our BlakeCompress implementation. This choice of implementation reduces the time of hash computation and improves the performance. The overall security is not compromised by this implementation (see section 8.2.1).

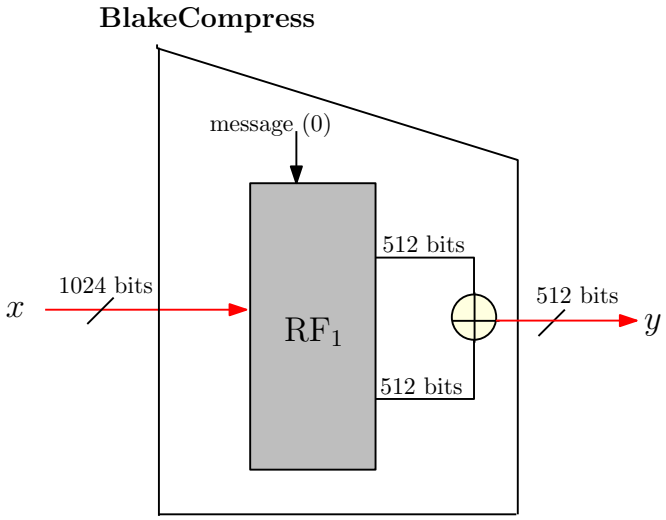
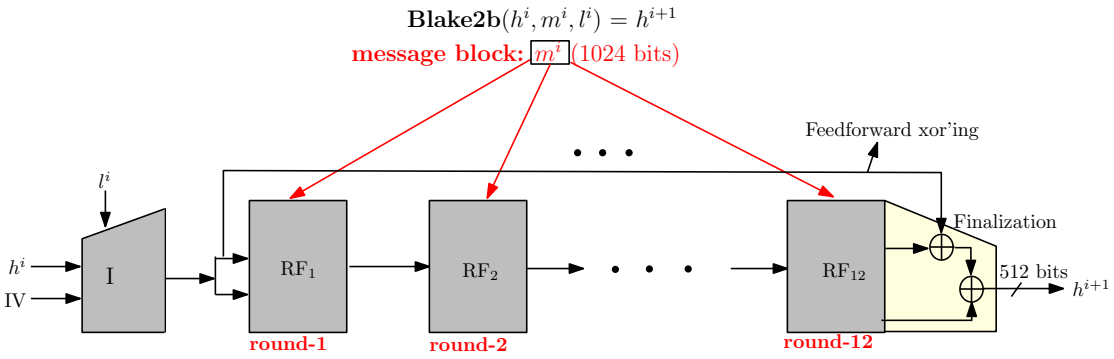


Fig. 2: Function H_2 implemented as function BlakeCompress$[x] = y$, where input length of x= 1024 bits and output length of y = 512 bits. RF_1 is the first round of Blake2b compression function. Input size of RF_1 = 1024 bits.

6.2 Design of Rig [BlakeExpand, BlakePerm, Blake2b]

The optimized variant of Rig uses an expansion function BlakeExpand to expand the state and a compression function BlakePerm to compress the state. Full Blake2b is used to hash the output state after the iterative-transformation phase to obtain the final hash. The design aspects are described below.



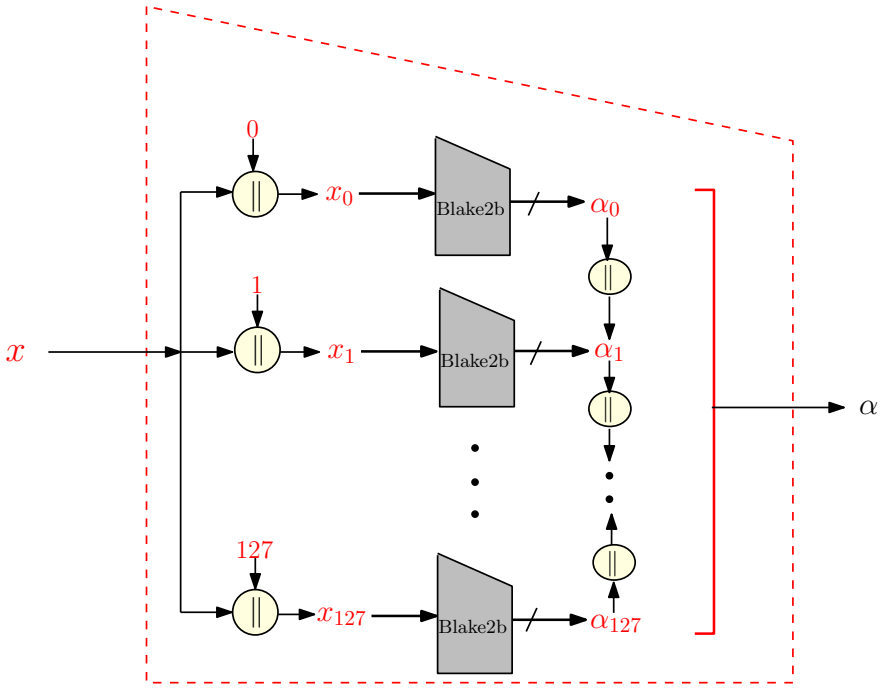
Blake2b compression funtion defined as **Blake2b**$(h^i, m^i, l^i) = h^{i+1}$ where
- h^i is the 512 bits chaining value,
- m^i is the ith message block and each m^i is of length1024 bits
- l^i denotes the number of data bytes in $m^0, m^1, \cdots, m^i$
- I denotes the initialization phase of Blake2b
- RF$_i$ denotes the ith call to the eight G-functions

Fig. 3: Blake2b compression function

6.2.1 Design of the BlakeExpand expansion function The BlakeExpand function is a very simple function which expands the input x to a fixed size of 8KiB. The function BlakeExpand is an instantiation of H_1. The input x passes through 128 individual instances of Blake2b (full hash) each appended by a counter as $x_i = x \parallel i$, for $0 \leq i \leq 127$ and produces the output $\alpha = \alpha_0 \parallel \alpha_1 \parallel \cdots \parallel \alpha_{127}$ where each α_i is of length 512 bits, i.e., 64 bytes. This construction ensures that the output of the function is random and the randomness depends solely on the cryptographic strength of Blake2b. Since this function needs to be executed only once, it has negligible impact on the performance of the overall *Rig* construction.

6.2.2 Design of BlakePerm function We provide the design considerations for the function BlakePerm before the description of the design.

Design considerations for BlakePerm function The DRAM memory latency is the limiting factor for the entire design of *Rig*. Initialization and copying data takes over 70 percent of the total run-time. In order to improve the overall performance one trivial optimization would be to increase the size of chunks in which the read and write operations are performed. The latest high performance processor offerings from Intel and AMD influenced many of the design decisions as they would be the most common target platform. There are several design considerations like:



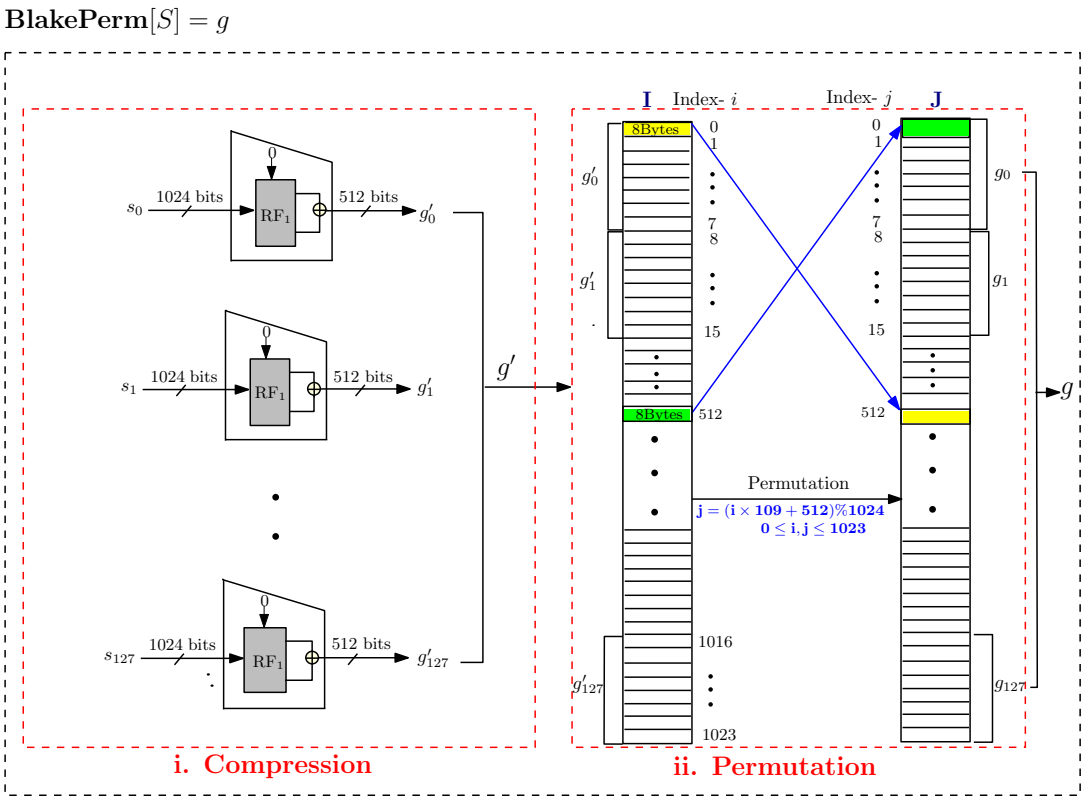
BlakeExpand $[x] = \alpha$ **where** $\alpha = \alpha_0 \parallel \alpha_1 \parallel \cdots \parallel \alpha_{127}$, $x_i = (x \parallel i)$ for $0 \leq i \leq 127$
and $x = \text{pwd} \parallel \text{binary}_{64}(\text{pwd}_l) \parallel s \parallel \text{binary}_{64}(s_l) \parallel \text{binary}_{64}(n) \parallel \text{binary}_{64}(l)$

Fig. 4: Function BlakeExpand

- **L1 cache size** This is one of the major factors because the L1 cache has the lowest latency of around 1-1.5ns (as few as 3 clocks). Therefore it is important that the work piece (chunk) fits within this size for high performance in case of a compute intensive task.
- **L1/L2/L3 cache line size** A typical modern processor has cache line size of 64 bytes. Therefore working in multiple of 64 bytes with preferably aligned memory access is the best strategy. The problem with working with other non-multiple sizes is that there would be a lot of extra accesses and split loads and stores, which will dramatically reduce performance in computation intensive tasks. The Blake2b compression function nicely fits this requirement as it compresses 128 bytes to 64 bytes. The only other requirement is an implementation detail for setting the proper aligned memory access while memory allocation. As a result, in our implementation we have zero split load/stores.
- **DRAM Latency** The memory latency is the primary limiting factor in algorithms having random reads and writes. DRAM generally has latency values of 250+ clock cycles. One strategy to get around this problem would

be to perform reads and writes in larger chunks. If the chunk size is large enough, the performance hit due to latency can become significantly small. We tested against various sizes from 2 KiB to 64 KiB and observed that 16 KiB is a good size; and it also fits the L1 cache. We have, as a result chosen chunk size of 16 KiB for the H_2 function.

Design of the BlakePerm function The BlakePerm function is a compression function which compresses 16 KiB of data to 8 KiB. It is a two step function as described below.



BlakePerm $[S] = g$, where input $S = s_0 \parallel s_1 \parallel \cdots \parallel s_{127}$ and output $g = g_0 \parallel g_1 \parallel \cdots \parallel g_{127}$
16 KiB input is equally divided in each 128 byte s_i.
RF_1 denotes the first round of Blake2b compression function

Fig. 5: Function H_2 implemented as function BlakePerm.

1. **Compression** It compresses the data using a single round Blake2b compression function. A single such compression function compresses 128

bytes to 64 bytes, as a result we need 128 such functions to compress 16 KiB to 8 KiB.

2. **Permutation** A permutation layer is needed to mix the compressed data so that the bit-relations are spread evenly among the output bits over several rounds. Even though, any random permutation can be chosen in such a scenario, a permutation of the form: output $O_i = (I_i \times A + B) \bmod C$ was chosen, where $0 \leq I_i \leq 1023$. The values $A = 109$, $B = 512$ and $C = 1024$ were chosen carefully after a series of experiments and diffusion tests. The permutation works on words of 8 bytes at a time, as a result the total addresses would be $8192/8 = 1024$. The value B = 512 is the number by which the overall permutation is cyclically rotated, it is half of the total size. The value of A is the most critical, even though the only requirement for a permutation is that A should be co-prime with C. The value of A strongly affects the overall characteristics and it needs to be carefully selected. It takes around 5 rounds for all the output bits to be fully affected by a single change in the input data. Since H_2 function is sequentially applied hundreds of times, the function BlakePerm produces complete avalanche and is cryptographically strong.

6.3 Parallelization

The design of *Rig* is sequential and therefore it is impossible to parallelize the overall implementation. As a result we chose to parallelize the H_1 and H_2 functions. The most critical function which affects the performance of the overall design is the H_2 function. The optimized variant Rig [BlakeExpand, BlakePerm, Blake2b] takes an input of 16 KiB and compresses it using 128 Blake2b compression functions. These 128 operations can be done in parallel to improve the performance without affecting the overall sequential nature of *Rig*. H_1 can similarly be parallelized but it has negligible effect on the overall performance.

6.4 GPU resistance

We designed *Rig* to have side-channel resistance, in pursuit of which we had to choose password-independent memory access patterns. Such memory-access patterns are harder to protect against GPU attacks. Modern GPUs have very strict requirements for memory accesses and very small cache sizes per core, as a result small random reads and writes dramatically reduce performance.

While designing H_2 of Rig [BlakeExpand, BlakePerm, Blake2b], we chose a permutation which causes reads and writes at significantly varying distances. Combined with the bit-reversal permutation used in *Rig* at the iterative transformation phase, the overall design is hard to parallelize efficiently.

As the H_2 function is pluggable, a new function can be easily added which performs small password-dependent memory accesses and make the design significantly GPU resistant. But, any such function would break the strict side-channel resistance.

6.5 ASIC/FPGA resistance

The *Rig* construction is strictly sequential and is therefore non-parallelizable. The compression function H_2 (BlakePerm) as explained above, can be parallelized. But, the size of the inputs and outputs (16 KiB to 8 KiB) which needs 128 parallel instances of Blake2b compression function is too large for implementations with a large number of simultaneous *Rig* instances.

Even though there can be a lot of possibilities of implementations with varying numbers of compression functions, the overall space requirement still remains high.

The biggest problem in case of ASIC resistance would however come from the memory latency and bandwidth of the DRAM needed for storage of the extremely large state (several hundred megabytes to a few gigabytes). Even though the compression functions consume less power because of their simplicity, the latency and very high memory bandwidth requirements would make parallel implementations on ASIC prohibitively expensive. For example, for a single instance of Rig [BlakeExpand, BlakePerm, Blake2b] having $n = 4$ (5 memory passes), and 1 GiB of state, the bandwidth on a standard PC exceeds 7.37 GiB/s as shown in Table 2.

7 Performance analysis

The reference implementation of *Rig* has been done in C language on an Intel Core i7-4770 CPU with 16GB RAM at 2400 MHz. For the implementation of the first round of the compression function of Blake2b[1] for the function H_2, we use AVX2 instructions. Specifically these AVX2 instructions are used to parallelize the implementation of first round G-function of Blake2b. The following tables (Table 1 and Table 2) show the performance figure in terms of 'memory hashing speed' and 'DRAM bandwidth' for different values of parameter n (number of iterations).
The reference implementation is available at: https://github.com/arpanj/Rig.

7.1 Suggested parameters

The performance figures provided in Table 1 and Table 2 show that, as expected, the memory hashing speed for Rig [BlakeExpand, BlakePerm, Blake2b] is significantly higher than that of the strictly sequential variant.

Due to the wide spectrum of possible uses it is very difficult to suggest optimal values for parameters which suits every possible implementation scenario. However, we can suggest values for common applications. For the parameter 'n' (number of iterations) we suggest values higher than 3. This means that one should have at least four passes over memory (including setup phase). For some scenarios this may be increased to make low memory attacks prohibitively expensive.

[1] The idea of using reduced-round Blake2b is inspired from [2, 6].

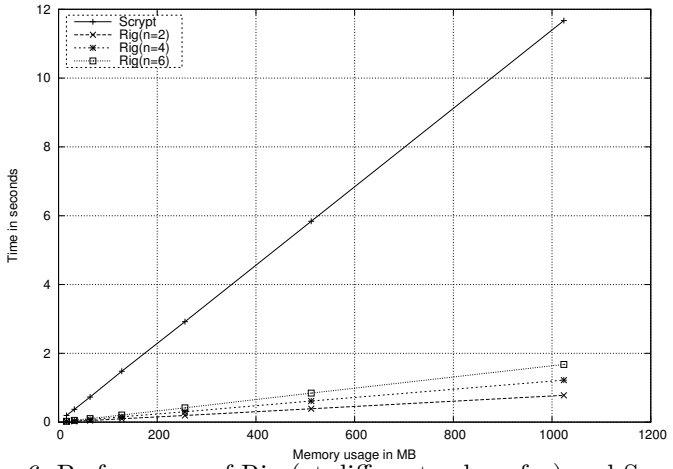


Fig. 6: Performance of Rig (at different value of n) and Scrypt.

The memory count value (m_c) would depend strongly on the requirement and the actual use-case. For a server client architecture where the clients are expected to have enough free RAM, the value can be set to use few tens of megabytes to a few hundred megabytes. In a mobile environment, this can be further reduced to allow for clients with smaller memories. In the case the algorithm is to be used as a proof-of-work test, large memory requirements of a few gigabytes combined with a large 'n' value can be set. It is important to keep 'n' high (as high as 6-10) in case the overall memory cost is very small.
The performance of Scrypt (with suggested parameters [13]) and the results from Table 2 are depicted in Figure 6. The graph shows the memory processing rate when consumable memory to compute the password hash is fixed. The comparison shows that the memory consumption of Scrypt is comparatively small with time. Scrypt takes approximately 6 seconds for 512 MB memory while *Rig* at $n = 2$ and $n = 4$ takes approximately 0.389 seconds and 0.613 seconds respectively.

8 Security analysis

Rig satistifies the basic requirement of a non-invertible design for password hashing because of the following reasons: (i) the iterative use of underlying primitive, the (secure) cryptographic hash function and (ii) the initial hashing of password with random salt and other parameters and the final use of salt with chaining data. This makes recovering password from the hashed output quite challenging.

Table 1: Performance of RIG [Blake2b, BlakeCompress, Blake2b]

1)RIG[Blake2b, BlakeCompress, Blake2b]:Memory Hashing Speed(MiB/s)							
M =>	15 M	30 M	60 M	120 M	240 M	480 M	960 M
n = 2	681	684	663	675	681	674	659
n = 4	383	383	381	388	388	391	392
n = 6	268	270	267	275	272	258	270
n = 8	211	210	210	212	210	201	209
n = 10	174	173	170	173	170	174	170
Memory Bandwidth (GiB/s)							
n = 2	3.552	3.566	3.458	3.517	3.551	3.515	3.436
n = 4	3.596	3.595	3.575	3.642	3.640	3.666	3.677
n = 6	3.638	3.665	3.624	3.737	3.693	3.500	3.666
n = 8	3.740	3.731	3.736	3.758	3.728	3.562	3.707
n = 10	3.826	3.789	3.721	3.796	3.737	3.811	3.736

Table 2: Performance of RIG [BlakeExpand, BlakePerm, Blake2b]

2)RIG[BlakeExpand, BlakePerm, Blake2b]:Memory Hashing Speed(MiB/s)							
M =>	64 M	128 M	256 M	512 M	1 GiB	2 GiB	4 GiB
n = 2	1377	1345	1307	1326	1315	1312	1318
n = 4	858	846	829	845	835	838	833
n = 6	621	621	617	618	606	610	619
n = 8	500	485	481	490	485	489	489
n = 10	403	398	399	402	404	404	402
DRAM Memory Bandwidth (GiB/s)							
n = 2	6.728	6.575	6.389	6.478	6.429	6.412	6.439
n = 4	7.547	7.441	7.295	7.431	7.347	7.374	7.329
n = 6	7.888	7.898	7.847	7.858	7.709	7.750	7.873
n = 8	8.309	8.070	8.003	8.152	8.072	8.123	8.126
n = 10	8.282	8.179	8.205	8.251	8.291	8.293	8.265

n = no. of iterations, performance figures averaged over 20-iterations.

Another important point is the simple, sequential and symmetric design of the scheme. The simplicity makes it easy to understand and sequential design

makes the parallel implementation hard and prevents significant speed up by the use of multiple processing units.

Flexibility of the design and the independence of the selection of memory parameter from time parameter makes it unique from existing constructions.

8.1 Resistance against low memory attack

Attacker's approach An attacker running multiple instances of *Rig* may try to do the calculations using smaller part of the memory (low memory) or almost no-memory (memory-free) to reduce the memory cost per password guess. This approach may allow parallel implementations of independent password guesses, utilizing almost all the available processing cores. This may not give advantage over single password hash computation but may increase the overall throughput of password guessing as compared to the legitimate implementation of the algorithm. Next we explain how feasible the low-memory or memory-free attack approach is, from the attacker's point of view.

Attack scenario Varying the required storage values We emphasize that the goal of analyzing the complexity of low memory attack is to show the approximate impact on the overall processing cost to implement the algorithm *Rig*. Our construction needs to store two arrays a and k as shown in Figure 1. Therefore we try to compute the time complexity when most of these array values are not stored. The cost of calculation for the values of array a are dominated by the cost of array k. Therefore, for the simplicity of the evaluation we consider the calculation cost for array k.

To vary the required storage at each iteration, we assume that **we store t consecutive values,** $0 \leq t \leq m-1$**,** of both the arrays at iterative transformation phase. This assumption is without loss of generality as we can easily calculate the index value of array k from the bit reversal permutation explained in section 2. We also store the hash chaining values after each iteration.

Effect of bit-reversal permutation on low memory scenario We use the bit-reversal permutation to shuffle the access of the array k. The effect of this yields exponential complexity for the low memory scenario. This is because at every step we update the values of array k and each updated value depends on all previous values. Let at iteration i, $1 \leq i \leq n$, $k[j]$, $0 \leq j \leq m-1$, is the required value that is not stored. Then we need to compute the value $k[j]$ at all previous $i-1$ iterations and as the access was not sequential, it is difficult to calculate the exact complexity. Hence, we compute the expected time complexity of a password hashing for memory constrained scenario.

Low memory attack complexity: The algorithm *Rig* can be computed with time complexity $O((n+1)mr)$ and space complexity $O(m)$ where $2m$ is the required number of stored values, n is the number of iterations used and r is the number of rounds. An attacker using reduced memory storage (i.e., 0 to $m-2$ stored values) will require a time complexity of $O(r \times m^{n+1})$ for a single

password computation.

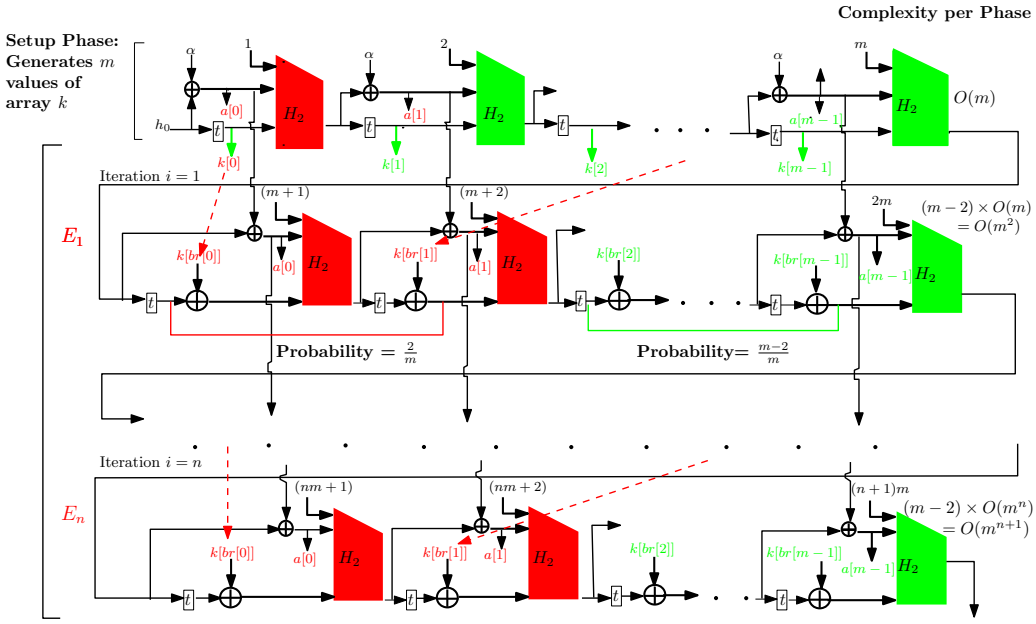
Analysis of low memory attack complexity For the legitimate implementation of the algorithm, *Rig*, we need to store $m = 2^{m_c}$ values of arrays a and k which are created at the setup phase and repeatedly accessed and updated at each iteration i of the iterative transformation phase (see Figure 1). Our goal is to analyse the extra cost incurred when we do not store the array values, and calculate them on the fly at each iteration i where $1 \leq i \leq n$. Specifically, we analyze the time complexity by varying the possible storage (0 to m) of the array values. At each iteration i of the iterative transformation phase, we apply bit reversal permutation (Algorithm 1) on m indices of the array k which we denote by $(1, 2, \cdots, m) \rightarrow (br[0], br[1], \cdots, br[m-1])$ and access the output of the permutation sequentially. It is easy to calculate this permutation for all n-iterations in advance. We calculate the overall cost of password hashing depending on whether a value of array k, let, $k[br[j]]$, $0 \leq j \leq m-1$ is stored or not. As mentioned before, to compute the complexity we store first t consecutive values of array k (where $0 \leq t \leq m-1$) that are required for corresponding hash calculations at iterative transformation phase. We also store the hash chaining values after each iteration except the last one, i.e., if the implementation uses n-iterations then we store $n-1$ such values. Figure 7 shows the graphical view of the low memory scenario with an example where we store two consecutive values of the arrays required for corresponding hash calculations, shown in red. Other $m-2$ values that are calculated on the fly and the corresponding hash calculations are shown in green. The general approach is explained below.

We apply the **law of total expectation** to estimate the expected running time for a password hashing, conditioning on the indices of the array k. This is because the calculation cost is very high when we do not store the values of required indices of k and is the most influential parameter in the overall attack complexity. Therefore, we calculate the probability of a value at a particular index of array k of being stored when we assume t consecutive values are stored. We also calculate the probability of a value at an index not being stored when $(m-t)$ values are not stored. Further, in case a specific index of array k is not stored, then we estimate the expected cost to evaluate this element at each iteration i.

We know that the total required indices are m for the array k. Out of these m values, we store t values while the remaining $m-t$ values are not stored. Therefore, the probability of a stored and not-stored index is given by

$$\mathbf{Pr}[\text{a value of an index is stored }] = \frac{t}{m}$$
$$\mathbf{Pr}[\text{ a value of an index is not stored }] = \frac{m-t}{m}$$

For $t = 2$ we store only 2 values and $m-2$ values are remained un-stored. Then, the above probabilities are $\frac{2}{m}$ and $\frac{m-2}{m}$ respectively as shown in Figure 7.



*Note: $k[br[i]]$: is the ith index of array k obtained from bit reversal permutation
Here in this example figure we store $t = 2$ consecutive memory for corresponding hash calculation shown in red
$(m-2)$ not stored values are shown in green

Fig. 7: Graphical representation of the calculation for the low memory complexity of *Rig*.

To apply the law of total expectation we apply the following concept.
At the setup phase we perform m hash calculations to store m values of array k. Therefore, for these m calculations the complexity of the setup phase is $O(m)$. In our construction, at each iteration i, $1 \leq i \leq n$, we access the values that are calculated at the previous iteration $i-1$, i.e., every next value is dependent on its previous value. Therefore at the first iteration of iterative transformation phase, if we need to access an element of array k that is not stored, we need to perform maximum m calculations of setup phase for that value. Therefore, at iteration $i = 1$ the total cost of calculation is $(m-t) \times O(m) = O(m^2)$ (as maximum m calculations are required for all $m-t$ not stored values). Similarly, at iteration $i = 2$, for all $m-t$ not stored values of array k it is required to calculate maximum m hashes of setup phase to generate the initial values. Then it is required to check when those values were updated at $i = 1$. At $i = 1$, the maximum calculations are: $(m-t) \times O(m) = O(m^2)$, and similarly at $i = 2$ the total calculations are: $(m-t) \times O(m^2) = O(m^3)$ and so on. This way we estimate the cost of calculation after each iteration of iterative transformation

phase shown in Figure 7 as 'Complexity per phase'. Now at each iteration we estimate the expected cost of calculation for each value $k[br[j]]$ of array k where $0 \leq j \leq m-1$. We denote the expected cost at iteration i as E_i. If a value is already stored then the expected cost of calculation, i.e., E(cost when value is stored)=1 but if the value is not stored then the expected cost will be the complexity till the previous iteration as the required value is dependent on all its previous values. So the expectation at each iteration is as follows:

$$
\begin{aligned}
E_1 = & \sum_{j=0}^{m-1} Pr[\text{ value at } k[br[j]] \text{ is stored }] \times [\text{ cost of calculation when } k[br[j]] \text{ is stored }] \\
& + Pr[\text{ value at } k[br[j]] \text{ is not stored }] \times E[\text{ cost of calculation } k[br[j]] \text{ is not stored }] \\
= & \sum_{j=0}^{m-1} \Big(\frac{t}{m} + \Big(\frac{m-t}{m} \Big) \times O(m) \Big) \\
E_2 = & \sum_{j=0}^{m-1} Pr[\text{ value at } k[br[j]] \text{ is stored }] \times [\text{ cost of calculation when } k[br[j]] \text{ is stored }] \\
& + Pr[\text{ value at } k[br[j]] \text{ is not stored }] \times E[\text{ cost of calculation } k[br[j]] \text{ is not stored }] \\
= & \sum_{j=0}^{m-1} \Big(\frac{t}{m} + \Big(\frac{m-t}{m} \Big) \times O(m^2) \Big) \\
& \cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots \\
E_n = & \sum_{j=0}^{m-1} Pr[\text{ value at } k[br[j]] \text{ is stored }] \times [\text{ cost of calculation when } k[br[j]] \text{ is stored }] \\
& + Pr[\text{ value at } k[br[j]] \text{ is not stored }] \times E[\text{ cost of calculation } k[br[j]] \text{ is not stored }] \\
= & \sum_{j=0}^{m-1} \Big(\frac{t}{m} + \Big(\frac{m-t}{m} \Big) \times O(m^n) \Big)
\end{aligned}
$$

The total cost E after $n-$iterations:

$$
\begin{aligned}
E &= m + E_1 + E_2 + \cdots + E_n \\
&= m + \sum_{j=0}^{m-1} \Big(\frac{t}{m} + \Big(\frac{m-t}{m} \Big) \times O(m) \Big) + \sum_{j=0}^{m-1} \Big(\frac{t}{m} + \Big(\frac{m-t}{m} \Big) \times O(m^2) \Big) + \cdots \\
&\quad \cdots + \sum_{j=0}^{m-1} \Big(\frac{t}{m} + \Big(\frac{m-t}{m} \Big) \times O(m^k) \Big) + \cdots + \sum_{j=0}^{m-1} \Big(\frac{t}{m} + \Big(\frac{m-t}{m} \Big) \times O(m^n) \Big) \\
&= m + nt + \Big(\frac{m-t}{m} \Big) \sum_{t=0}^{m-1} [O(m) + O(m^2) + \cdots + O(m^n)] \\
&= m + nt + \Big(\frac{m-t}{m} \Big) O(m^{n+1})
\end{aligned}
$$

Conditioning on the values of t we get the following complexities.

Case (i) : $t = 0$ implies the case of memory-free attack where the attacker does not use any memory.

$$\text{The expected cost is } = m + O(m^{n+1}) \equiv O(m^{n+1})$$

Repeating for r-rounds, the complexity of the attack is $O(r \times m^{n+1})$.
Case (ii) : When $1 \leq t \leq m-1$, i.e. the case when the attacker stores some of the memories.

$$\text{The expected cost is } = \Big(\frac{m-t}{m}\Big)O(m^{n+1}) \equiv O(m^{n+1})$$

Repeating for r-rounds, the complexity of the attack is $O(r \times m^{n+1})$
Case (iii) : $t = m$ implies the legitimate implementation of the algorithm.

$$\text{The complexity is } = m + nm \equiv O(n+1)m$$

Repeating for r-rounds, the complexity is $O(n+1)mr$.

Therefore, the memory-free attack complexity of *Rig* is $O(r \times m^{n+1})$ where r is the number of rounds.

8.2 Resistance against collision attack

In the design of *Rig* (see Figure 8) we define three different functions, H_1, H_2 and H_3. The input of H_1 is x where x is the concatenation of password, 64-bit value of password length, salt, 64-bit value of salt length, 64-bit value of n (number of iterations) and 64-bit value of the output length of password hash. The output of H_1 is α. The function $\mathcal{H}_2$ signifies the repetitive computation of function H_2 at setup phase and iterative transformation phase (see Figure 1) and generates the output c which is the output of iterative transformation phase. Therefore the inputs of $\mathcal{H}_2$ are α, m_c and n, where m_c is the number of memory count and n, the number of iterations used. Finally H_3 takes the concatenation of a 64-bit value which is the function of m_c and n, output of $\mathcal{H}_2$, the value salt and 64-bit value of 2^{m_c} and produces the output of password hash. Here we are considering round $r = 1$ (w.l.o.g). This is because, different values of round, say r and r' implies collision of H_3.

Theorem 1. *Collision for* Rig *means*

- ***i.*** *collision for H_1, or*
- ***ii.*** *collision for $\mathcal{H}_2$ when $\alpha \neq \alpha'$ and $(m_c, n) = (m'_c, n')$ for two different password hash computations, where $m_c = m'_c$ = the memory count and $n = n'$ = number of iterations, or*
- ***iii.*** *collision for H_3.*

Proof. We analyse the collision of *Rig* with five possible cases as shown in Figure 9. Specifically, we include all possible conditions that results in collision of H_1 or collision of $\mathcal{H}_2$ or collision of H_3 which implies the overall collision of *Rig*.

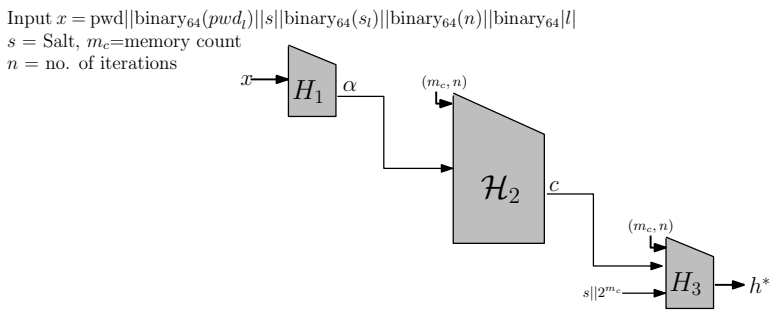


Fig. 8: Rig$[H_1,\ H_2,\ H_3](x, m_c, n, s)$, where $\mathcal{H}_2$ signifies repetitive use of H_2.

CaseI. Collision Rig $[H_1, H_2, H_3]$ if $(s, m_c, n) = (s', m'_c, n') \implies$ Collision H_1: The construction of *Rig* is such that if we get collision at H_1 for two different inputs say, x and x' and if $(s, m_c, n) = (s', m'_c, n')$ then it implies collision of $\mathcal{H}_2$ which implies collision at H_3, i.e., collision of Rig $[H_1, H_2, H_3]$.

CaseII. Collision Rig $[H_1, H_2, H_3]$ if $(s, m_c, n) = (s', m'_c, n')$ and $\alpha \neq \alpha' \implies$ Collision $\mathcal{H}_2$: For two different inputs x, x' if $\alpha \neq \alpha'$ then collision of $\mathcal{H}_2 \implies$ collision of H_3 when $(s, m_c, n) = (s', m'_c, n')$ for respective inputs.

CaseIII. Collision Rig $[H_1, H_2, H_3]$ if $(s, m_c, n) = (s', m'_c, n')$, $\alpha \neq \alpha'$ and $c \neq c' \implies$ Collision H_3: For two different inputs x, x' if $\alpha \neq \alpha'$ and $c \neq c'$ and if $(s, m_c, n) = (s', m'_c, n')$ then collision Rig $[H_1, H_2, H_3] \implies$ collision H_3, i.e., collision is due to H_3.

CaseIV. Collision Rig $[H_1, H_2, H_3]$ if $(s, m_c, n) \neq (s', m'_c, n') \implies$ Collision H_3: For two different inputs x, x' if $\alpha \neq \alpha'$ and $c \neq c'$ and if $(s, m_c, n) \neq (s', m'_c, n')$ then collision Rig $[H_1, H_2, H_3] \implies$ collision H_3.

CaseV. Collision Rig $[H_1, H_2, H_3]$ if $x = x'$ and $m_c \neq m'_c \implies$ Collision H_3: For two different password hash calculations if $m_c \neq m'_c$ for the same input x then collision of Rig $[H_1, H_2, H_3]$ is for collision at H_3.

Therefore these five cases describe how collisions of *Rig* implies collision of H_1 or $\mathcal{H}_2$ or H_3. Hence *Rig* is collision resistant if H_1 or $\mathcal{H}_2$ or H_3 are collision resistant. □

8.2.1 Instantiation of Rig $[H_1, H_2, H_3]$ for reference implementations

Collision resistance of Blake2b: For our reference implementation we use Blake2b. This choice is motivated for several reasons. Blake2b is an improved version of Blake which is one of the finalists of SHA-3 competition. Further, in terms of speed of hashing, it among the most efficient hash functions in the SHA-3 competition. The significant differences between Blake and Blake2b are that Blake2b uses fewer rounds (12 instead of 16), provides optimized (in terms of speed) rotations in G-function, uses fewer constants (8 word constants instead of 24). It is mentioned in [4] that, "Based on the security analysis performed so far, and on reasonable assumptions, it is unlikely that 16 rounds are meaningfully more secure than 12 rounds". Despite several years of cryptanalysis there are

no significant attack on Blake. A recent paper cryptanalyzing Blake2b is [7] shows that no collision or preimage attacks exist against Blake2b. In fact, the best differential against Blake2b covers 3.5 rounds and has a complexity of 2^{480}. Therefore Blake2b is quite secure against collision and preimage attacks, properties which we need in our design. For further details on attacks on compression function and permutation of Blake2b, we refer the reader to [7]. Hence, the use of full round Blake2b is enough to claim security of our construction against attacks utilizing collisions or preimages of the underlying hash function.

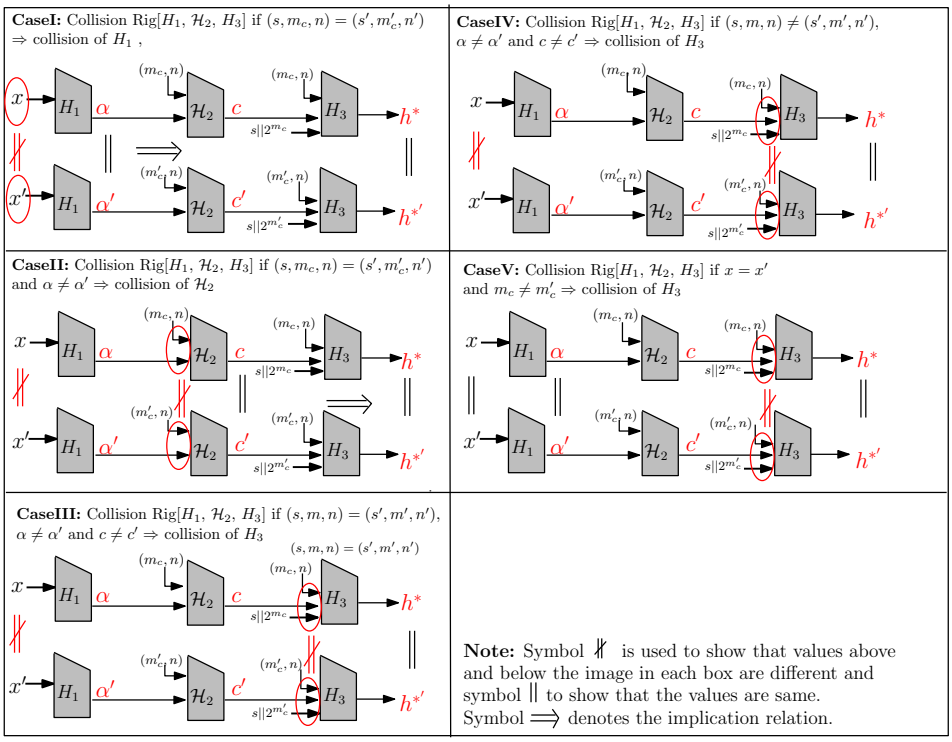


Fig. 9: Five cases showing collisions for Rig $[H_1, H_2, H_3]$.

– **Rig [Blake2b, BlakeCompress, Blake2b]**
Instantiation of H_1 and H_3 with Blake2b: We use full round Blake2b for the functions H_1 and H_3. We need collision resistance of both the functions for the security of *Rig*. Use of Blake2b allows us to claim collision resistance of H_1 and H_3 [7].

Instantiation of H_2 with BlakeCompress: The function $\mathcal{H}_2$ as shown in Figure 8 signifies the repetitive computations of the function H_2 and inputs of $\mathcal{H}_2$ are α, m_c, n. Theorem 1 shows that collision of $\mathcal{H}_2$ implies collision of *Rig* only if H_1 gives two distinct outputs say, α, α' for two different inputs $x \neq x'$ while corresponding values $(m_c, n) = (m'_c, n')$. We use first round

of Blake2b compression function for the implementation of H_2 as shown in Figure 2.

For finalization phase we omit the feed-forward xor'ing as explained in Blake2b (see Figure 3). This choice of implementation is to improve the performance, but it does not preserve the onewayness as there is no feed-forward xor'ing. In this case it would appear that backward calculation is easy as H_2 computation is then reversible. But due to the design of *Rig* it is very difficult to perform these backward computations as input values of H_2 are arrays a and k and depends on their values at previous iteration (see Figure 1). Therefore, we only have the freedom to guess the values of array a and k to move backward with reverse calculation for the last iteration, say $i = n$. Last iteration n will fix the values of iteration $(n-1)$ as values are dependent. So we lose the freedom of guessing the values from iteration $(n-1)$ and onwards and the backward computations become hard. Hence omitting the feed forwardness can compromise the onewayness but the design of the scheme provides no security loss. Now consider two distinct inputs of $\mathcal{H}_2$ as $\alpha = H_1(x)$ and $\alpha' = H_1(x')$ for $x \neq x'$.

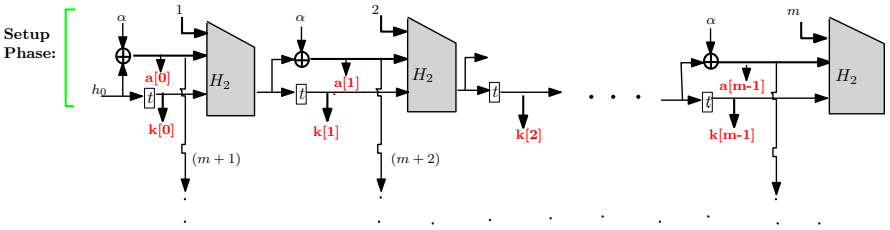


Fig. 10: Setup phase uses m inputs generated from α.

Figure 10 shows that $m = 2^{m_c}$ number of computations of H_2 of the setup phase are generated from these different α or α' and it will be difficult to find collision everywhere as values generated from two different inputs are usually expected to be different. The values of array a and k are generated from α or α' and influence further calculations. Now, consider the last iteration of iterative transformation phase, i.e., the last layer of 2^{m_c} computations of H_2 as shown in Figure 11. All these H_2 calls use single round Blake2b and are generated from the same value α or α'. The input values influenced by α or α' are shown in red color and most of them will have nonzero difference. We can visualize this scenario as similar to the Blake2b construction, instead of 12 rounds using 2^{m_c} rounds. Use of comparatively large number of rounds provides enough security. Since we expect the parameter $m_c \geq 10$, we will have 1024 rounds of Blake2b compression function. Therefore, we can expect $\mathcal{H}_2$ to be collision resistant.

Therefore the choice of reduced round Blake2b does not affect the collision resistance of the design.

– **Rig [BlakeExpand, BlakePerm, Blake2b]**

Instantiation of H_1 with BlakeExpand and H_3 with Blake2b: As explained in section 6.2 we are using 128 individual instances of Blake2b (full hash) appended by a counter for the function H_1. Therefore a collision for H_1 means collisions for each of the 128 calls to Blake2b. Therefore, collision resistance of H_1 is obtained from the collision resistance of Blake2b. For the function H_3 we implement Blake2b (full round), therefore H_3 is collision resistant if Blake2b is so.

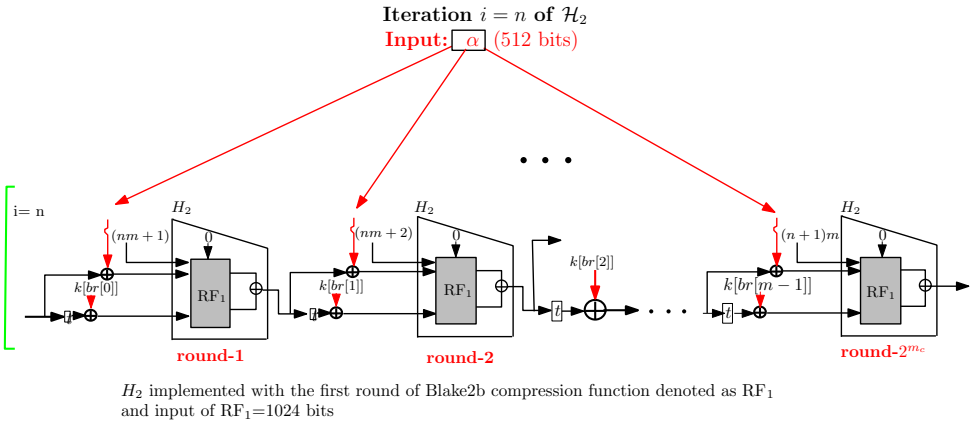


Fig. 11: H_2 implemented with RF$_1$ at iteration $i = n$.

Instantiation of H_2 with BlakePerm: We implement each H_2 with two step functions. The first function, 'Compression' executes 128 instances of first round Blake2b compression function and compresses total 16 KiB input to 8 KiB. The second function, 'Permutation' is a permutation that mixes the compressed 8 KiB output to spread the bit-relations evenly among the output bits (see Figure 5).
We can provide similar arguments for the security of $\mathcal{H}_2$ as explained for the sequential variant, Rig [Blake2b, BlakeCompress, Blake2b] above.
Hence, the above choice of implementation is expected to be secure against collision attack.

8.3 Resistance against length extension attack

The length extension attack on hash function works as follows. Hash functions work on blocks of data iteratively. Let the initial value of a hash be $IV{=}h_0$. A long message $m = m_0 \| m_1 \| \dots \| m_l$ is processed as follows. First $h_1 = H(h_0, m_0)$

is computed and then the recursion $h_{i+1} = H(h_i, m_i)$ is used for $1 \leq i \leq l$. The final output is h_{l+1}. An attacker, without knowing the intermediate message blocks, may simply append a new block m^* and compute the hash value of the message $m' = m_0 \| m_1 \| \ldots \| m_l \| m^*$, by using one call to the hash function as $H(h_{l+1}, m^*)$. In our construction the password is not the only input to the hash function. If an attacker attempts to append the password by any text, the value of α will change and it will affect all subsequent blocks. Thus the length extension attack is not feasible in our design.

8.4 Resistance against cache-timing attack

As discussed in section 5, our construction accesses array k (see section 4) using a bit reversal permutation which is password independent. Hence cache timing attack is not possible on our costruction. Further, since the only primitive used in our scheme is a secure hash function, the security of our scheme can be formulated in terms of the security of the underlying hash function. With the current state-of-the-art we have the possibility of using SHA-3 implementation, or even any of the other finalists of SHA-3 competition, which are resilient to side-channel attacks. Thus our scheme resists cache timing attacks.

8.5 Resistance against denial-of-service attack

In computing, a denial-of-service (DoS) attack is an attempt to make a machine or network resource unavailable to its intended users. This is possible by making the server busy injecting lots of request for some resource consuming calculation. It is quite easy if the server uses some slow password hashing technique for authentication. To handle such situations, the server-relief technique can provide some relief to the server from heavy calculations as the client will do the heavy part of the algorithm. This way we can reduce the chances of DoS attacks with slow password hashing schemes.

9 Conclusions

In this work, we have proposed a secure password hashing scheme *Rig* based on cryptographic hash functions. Besides supporting necessary and commonly used features of a password hashing scheme, *Rig* also supports client-independent updates and server relief technique. The flexibility in the choice of the two important parameters (memory count and number of iterations) enhances the scope of its implementation. *Rig* can be implemented in various software and hardware platforms including resource constraint devices.

10 Acknowledgments

We would like to thank the anonymous reviewers of Inscrypt and the contributors to the PHC mailing list (specially Bill Cox) whose comments helped improve the paper significantly.

References

1. Password Hashing Competition (PHC), 2014. `https://password-hashing.net/index.html`.
2. Leonardo C. Almeida, Ewerton R. Andrade, Paulo S. L. M. Barreto, and Marcos A. Simplício Jr. Lyra: Password-Based Key Derivation with Tunable Memory and Processing Costs. *IACR Cryptology ePrint Archive*, 2014:30, 2014.
3. Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. BLAKE2: Simpler, Smaller, Fast as MD5. In Michael J. Jacobson Jr., Michael E. Locasto, Payman Mohassel, and Reihaneh Safavi-Naini, editors, *ACNS*, volume 7954 of *Lecture Notes in Computer Science*, pages 119–135. Springer, 2013.
4. 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.
5. Carlos Carvalho. The gap between processor and memory speeds. In *Proc. of IEEE International Conference on Control and Automation*, 2002.
6. Joan Daemen and Vincent Rijmen. A new MAC construction ALRED and a specific instance ALPHA-MAC. In *Fast Software Encryption: 12th International Workshop, FSE 2005, Paris, France, February 21-23, 2005, Revised Selected Papers*, pages 1–17, 2005.
7. Jian Guo, Pierre Karpman, Ivica Nikolic, Lei Wang, and Shuang Wu. Analysis of BLAKE2. In Josh Benaloh, editor, *Topics in Cryptology - CT-RSA 2014 - The Cryptographer's Track at the RSA Conference 2014, San Francisco, CA, USA, February 25-28, 2014. Proceedings*, volume 8366 of *Lecture Notes in Computer Science*, pages 402–423. Springer, 2014.
8. Markus Dürmuth, Tim Güneysu, Markus Kasper, Christof Paar, Tolga Yalçin, and Ralf Zimmermann. Evaluation of Standardized Password-Based Key Derivation against Parallel Processing Platforms. In Sara Foresti, Moti Yung, and Fabio Martinelli, editors, *ESORICS*, volume 7459 of *Lecture Notes in Computer Science*, pages 716–733. Springer, 2012.
9. Christian Forler, Stefan Lucks, and Jakob Wenzel. The Catena Password Scrambler. Submission to Password Hashing Competition (PHC), 2014.
10. Jim Gray and Prashant Shenoy. Rules of Thumb in Data Engineering. Technical Report, MS-TR-99-100, Microsoft Research, Advanced Technology Division. December 1999, Revised March 2000.
11. 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.
12. William Burr Meltem Snmez Turan, Elaine Barker and Lily Chen. NIST: Special Publication 800-132, Recommendation for Password-Based Key Derivation. Computer Security Division Information Technology Laboratory. `http://csrc.nist.gov/publications/nistpubs/800-132/nist-sp800-132.pdf`.
13. Colin Percival. Stronger key derivation via sequential memory-hard functions. In *BSDCon*, 2009. `http://www.bsdcan.org/2009/schedule/attachments/87_scrypt.pdf`.
14. Niels Provos and David Mazières. A Future-Adaptable Password Scheme. In *USENIX Annual Technical Conference, FREENIX Track*, pages 81–91. USENIX, 1999.
15. Robert R. Schaller. Moore's law: past, present, and future. *IEEE Spectrum*, June 1997.