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

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

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

March 31, 2014

Contents

List of Tables

List of Figures

1 Abstract

Password Hashing, a technique commonly implemented by the server to protect passwords of the client, performs a one-way transformation on the password turning it into another string, called the hashed password. In this report, we introduce a secure password hashing framework *Rig* which is based on cryptographic (secure) hash functions. The design of the scheme is very simple to implement in software, flexible as the memory parameter is independent of time parameter (no actual time and memory trade-off) and strictly sequential (i.e., not easily parallelizable) providing strong security even against attackers that use multiple processing units. It supports client-independent updates i.e., the server can increase the security parameters by updating the existing hashes without knowing the password. Rig can also be implemented as a server relief protocol where the client bears the maximum effort to calculate the password hash with a minimal effort from the server side.

We compare Rig and show that our proposal provides a high security level with exponential complexity against memory-free attack, while the complexity for scrypt algorithm is quadratic [1].

2 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 safety of the passwords even when the authentication data is somehow leaked from the server, passwords are never stored in clear. 'Password Hashing' is a 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 guessing the password infeasible in terms of brute force attack, either in software or by using GPU's (Graphics Processing Unit), is essential to protect the user security and identity.

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

- Fast to prove the authenticity of the user but comparatively slow to reduce the rate of password guessing.
- Simple and easy to implement (coding, testing, debugging, integration) design, i.e., the algorithm should be simple in the sense of clarity, concise with less number of components and primitives and should not require too much prior knowledge to understand.
- Flexible and scalable design, 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: the construction should behave as a random function (random-looking output, one-way, collision resistant, immune to length extension, etc.).
- Resistant to GPU attack: Slow and costly hardware implementation (requirement of comparatively huge memory) that can make GPU attack infeasible.
- Leakage Resilience: Protection against information extraction from physical implementation is also very important, i.e., the scheme should not leak information about the password due to cache timing or memory leakage, while supporting any length of password.
- Ability to transform an existing hash to a different cost setting without knowledge of the password.
- Server relief technique where client does the maximum calculation for password hashing and 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.

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' [2], a password hashing scheme which used this idea for the first time. The main design principle of Scrypt is that it asymptotically uses almost as many memory locations as it uses operations to make the password-hash computation process slow. Memory is 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 lack of sufficient memory. This will force the attacker to run an (almost) sequential implementation making the attack significantly slow.

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 very simple to implement in software, flexible as the memory parameter is independent of time parameter (no actual time and memory trade-off) and strictly sequential (i.e., not easily parallelizable), providing strong security even against attackers that use multiple processing units. It supports client-independent password hash upgradation without the need of the actual password. This feature helps to increase the security parameters with technological improvements. The design provides protection against the extraction of information from

cache-timing attack and prevents denial-of-service attack if implemented to provide server-relief. We analyze Rig and show that our proposal provides a high security level with exponential complexity against memory-free attack.

The rest of the document is organised as follows. First we present the cryptographic techniques and primitives necessary for understanding the specifications followed by the design rationale and the description of the scheme. Subsequently, the implementation aspects with performance analysis are presented. This is followed by the motivations of all design choices and the treatment of the resistance against known types of attacks. Finally, we provide the security analysis of the scheme.

3 Cryptographic Preliminaries

The simple techniques and cryptographic primitive that are explained below are the basis of this construction:

3.1 Technique

- **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.

- **Bit Reversal Permutation:** Let V be a vector of length $l = 2^k$, where $k \in \mathbb{N}$. A bit-reversal permutation is a permutation on the sequence of l items of the vector. The simple way to implement this bit-reversal permutation is by indexing the elements of the vector by the numbers from 0 to $l - 1$ and then reversing the binary representations of each of these numbers (padded so that each of these binary numbers has length exactly k). Each item is then mapped to the new position given by this reversed value. Example: $V = \{000, 001, 010, 011, 100, 101, 110, 111\}$ then after bit reversal we get the permutation as: $V_r = \{000, 100, 010, 110, 001, 101, 011, 111\}$. It is an involution, i.e., if we repeat the same permutation twice it returns to the original ordering of the items.

3.2 Cryptographic Primitive

Our scheme uses a cryptographic hash function. The basic idea of a cryptographic hash function is that it associates an almost unique, short and easy to compute value to an arbitrary length message.

Definition 3.1. *A cryptographic hash function is a function H that maps an input of arbitrary bit-length M, called message, to an output of fixed bit-length m, called image or hash, i.e.,* $H : \{0,1\}^* \rightarrow \{0,1\}^m$.

However, not every such function H is a good cryptographic hash function. There are three main properties which a good cryptographic hash function has to preserve.

1. Collision resistance: It should be computationally infeasible to find any two different messages M, M^* such that $H(M) = H(M^*)$. The security goal of collision attack is $2^{m/2}$.

2. Preimage resistance: Given $y \in \{0,1\}^m$, it is computationally infeasible to find a message M such that $H(M) = y$. The security goal of preimage attack is 2^{m-1}.

3. Second preimage resistance: Given a message M, it is computationally infeasible to find a second message $M^* \neq M$ such that $H(M) = H(M^*)$. The security goal of second preimage attack is 2^{m-1}.

4 Attack Platforms: Significant hardwares

According to Moore's Law, over the history of computing hardware, the number of transistors on integrated circuits doubles approximately every two years. 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 **application-specific integrated circuit** (ASIC), is an integrated circuit (IC) which can be customized with memory chips to implement a dedicated design. An ASIC cann't 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 a 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 to increase the rate of password guessing one can easily use parallel FPGAs. According to the result in [3] the RIVYERA FPGA cluster is very powerful and cost optimized hardware, which hashes 3,56,352 passwords per second implementing PBKDF2 with SHA-512 and 512-bit derived key length. This is possible because PBKDF2 does not consume memory for password hashing. Comparing FPGAs with GPUs (Graphics processing units), the authors of [3] provide results of the

same implementation on 4 Tesla C2070 GPUs as 1,05,351 passwords per second. Performance-wise ASIC is better than FPGA but cost-wise FPGA is preferable. Following Moore's law, the speed of hardware is likely to increase by almost a factor of two in less than two years, but not the memory. Hence, to minimize the effects of such high performance hardware, we need a password hashing algorithm which consumes comparatively large memory to prevent parallel implementation.

5 Design rationale

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

- **Simplicity and Flexibility:** Symmetry in the design of Rig enhances the overall clarity of the scheme. No internal use of different primitives or constructions makes it easy to understand and implement (coding, testing, debugging, integration).

  Flexibility is introduced as the memory parameter is independent of the time parameter and we are able to scale any of them to get required performance. Specifically, balancing time and memory is not a trade-off.

- **Random output:** Use of different counter values for calculating each hash of the scheme assures that no input is repeated. Implementing upon the best known hash function prevents pre-image and collision attacks. With the property of different input and different output and same input and same output the scheme mimics the Random Oracle model. This enhances the security of the scheme.

- **Client-independent update:** The design of Rig supports client independent update, i.e., server can increase the security parameters without knowing the password. This is possible if we fix the value of n (number of iterations) and increase the number of rounds. Each round of the algorithm doubles the memory count of the previous round and hence increases the security parameter. Output of each round will be the input to the next. As each round gives full hash maintaining all requirements of a good password hashing technique, increasing number of rounds will increase security. Therefore, we calculate the initialization phase followed by setup phase, iterative transformation phase and output generation phase for the first round, but for all further rounds we iteratively calculate setup phase, iterative transformation phase and output generation phase sequentially. Moreover, we take output of one round as the input to the next round.

- **Resistance against cache-timing attack:** In our construction, to access the memory which is stored in array we use bit-reversal permutation, which is independent of the password used. If we use the permutation which depends on the value of the password and if the array can be stored in the

cache while accessing the values, we can trace the access pattern observing the time difference in each access of the array index. This helps to filter out the passwords that follows similar memory access pattern. Hence it is recommended [4] to have password-independent memory access pattern for password hashing. We have attempted to follow this requirement using bit reversal permutation.

- **Server-relief hashing:** Current requirement of a password hashing technique is that it should be slow and should demand huge memory to implement. But this 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 [4] and our construction supports this property. The protocol runs as follows: first the authentication server provides the salt to the client and the client performs the heavy computations while consuming huge memory resources and sends the result before the last hash calculation, i.e., line no.22 of our Algorithm, to the server. The server calculates the last hash taking concatenation of the counter value, the value sent by the client, and the salt value as input and produces the final hash. This way we can easily reduce the load of the server.

6 Specification

Our scheme mainly iterates the hash function with the flexibility of choosing variable length password and produces the password hash. In this section we intended to explain the cipher structure without implementation details.

6.1 Structure of the password hashing scheme

Rig: A password hashing scheme

Algorithm: Rig Construction

Input: Password(pwd), Salt(s), No. of iterations(n), Memory count(m_c), Output length(l), first $t-$ bit truncated hash of setup phase, No. of rounds r.

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

▷ Initialization phase: generates α from password

Initialize: Memory $m = 2^{m_c}$, a random salt (s) of atleast 16-bytes, Arrays k and a both of size m, $c = 0$, value t

x=(pwd)$||s||$binary$_{64}(n)||$binary$_{64}|l|$

$\alpha = H_1(x)$ ▷ H_1 : underlying hash function

for round 1 to r

  ▷ Initialization of Setup phase: Creates two arrays of size m

  h_0 = first 64 byte representation of the value π after decimal

  $a[0] = \alpha \oplus h_0$, $k[0]$ = first $t-$bits of h_0 and $m = 2^{round-1}m$

  for $i = 1$ to m

    $h_i = H_2(i||a[i-1]||k[i-1])$ ▷ H_2 : underlying hash function

    $a[i] = \alpha \oplus h_i$

    if $i \neq m$

      $k[i] = trunc_t(h_i)$ ▷ truncates the first $t-$bits of the hash output

  ▷ Initialization of Iterative Transformation phase

  for $i = 1$ to n

    for $j = 1$ to m

      $a[j-1] = a[j-1] \oplus h_{\{im+j-1\}}$

      $br[j]$ = index value of array k obtained using bit-reversal permutation

      $k[br[j]] = k[br[j]] \oplus \text{trunc}_t(h_{im+j-1})$

      $h_{im+j} = H_2((im+j)||a[j-1]||k[br[j]])$

  ▷ Output generation

  ${h_{round}}^* = (H_3(((n+1)m+1)||h_{(n+1)m})||s||m)$

~

6.2 Description

Following are the step by step explanation of the construction:

1. First we need to fix following parameters:

   - r= number of rounds for the setup phase followed by iterative transformation phase.
   - m_c= memory count, i.e.,two times $m = 2^{m_c}$ items to be store in memory. The value of m increases at each round as $2^{round-1}m$ to increase the security parameter.
   - n= number of iterations required to perform Iterative Transformation phase,
   - t= output length obtained by the truncation of hash output using a function defined as: trunc$_t(x)$ which outputs first $t-$bits of the value x. This $t-$ bit value is used for each entry of array k.

2. **Initialization phase:** We map the above parameters, namely the values of n and length of l, i.e., $|l|$ to 64-bit binary value using the simple 64-bit binary mapping. We create the value x as the concatenation ($||$) of the above mentioned parameters as
   $x = pad(pwd)||s||binary_{64}(n)||binary_{64}|l|$
   We calculate the hash of x and store it as α and iteratively use this value for further calculations in setup phase. At this step, we mention the hash function as H_1.

3. **Setup phase:** Here we initialize the value of h_0 as first 64-bytes (hash output length) of the value π after the decimal point and then initialize the value of arrays k and a at index zero as $k[0]$ = first $t-$bits of h_0 and $a[0] = \alpha \oplus h_0$ respectively.
   In this phase we iteratively calculate m-hash values taking input as counter concatenated with $a[i]$ where $0 \leq i < m$ and $a[i] = (\alpha \oplus h_{i-1})$, i.e., α xor with previous hash chaining value. Each hash output is stored in array k as first $t-$bit truncation of hash output and this array k is accessed randomly at next iterative transformation phase, but values of array a are accessed sequentially. Starting with the setup phase to last output calculation, the algorithm starts the counter from value 1 and increments it by one at each further hash calculations. So here no counter value is repeated. At this and next phase, H_2 denotes the underlying hash function.

4. **Iterative Transformation phase:** This phase is mainly designed to make constant use of the stored values and to update them. It iterates $n-$times and at the beginning of each iteration it first updates the value of $a[i]$ as $a[i] = (a[i] \oplus h_{i-1})$, then it calculates the index value of array k by using bit reversal permutation, which is denoted by $br[i]$, $0 \leq i \leq m-1$, for $i-$th calculation. The value at index $br[i]$ of array k is then updated by taking xor of the last $t-$bit hash output and previous value at $k[br[i]]$, i.e., $k[br[i]] = k[br[i]] \oplus trunc_l\{h_{i-1}\}$ where h_{i-1} is the previous hash output. So at this phase input of each hash calculation is: {current counter$||$ $a[i-1]||$ $k[br[i]]$}.

5. **Final Output:** After execution of the above setup phase and iterative transformation phase sequentially, we calculate one more hash, denoted by H_3 taking input: the current counter value, previous chaining value, the salt and the value of number of memory m to produce the password hash as:
   Output= H_3(current counter value $||\alpha||$ (previous chaining value) $||s$, || memory count), a l-bit value.

   Note: The algorithm stores the whole hash output as the password hash, however it is possible to store some truncated bits of the final output. In such case, if the server wants to increase the security parameter it can append some constant at the end to make the truncated hash value equal to the actual hash output length.

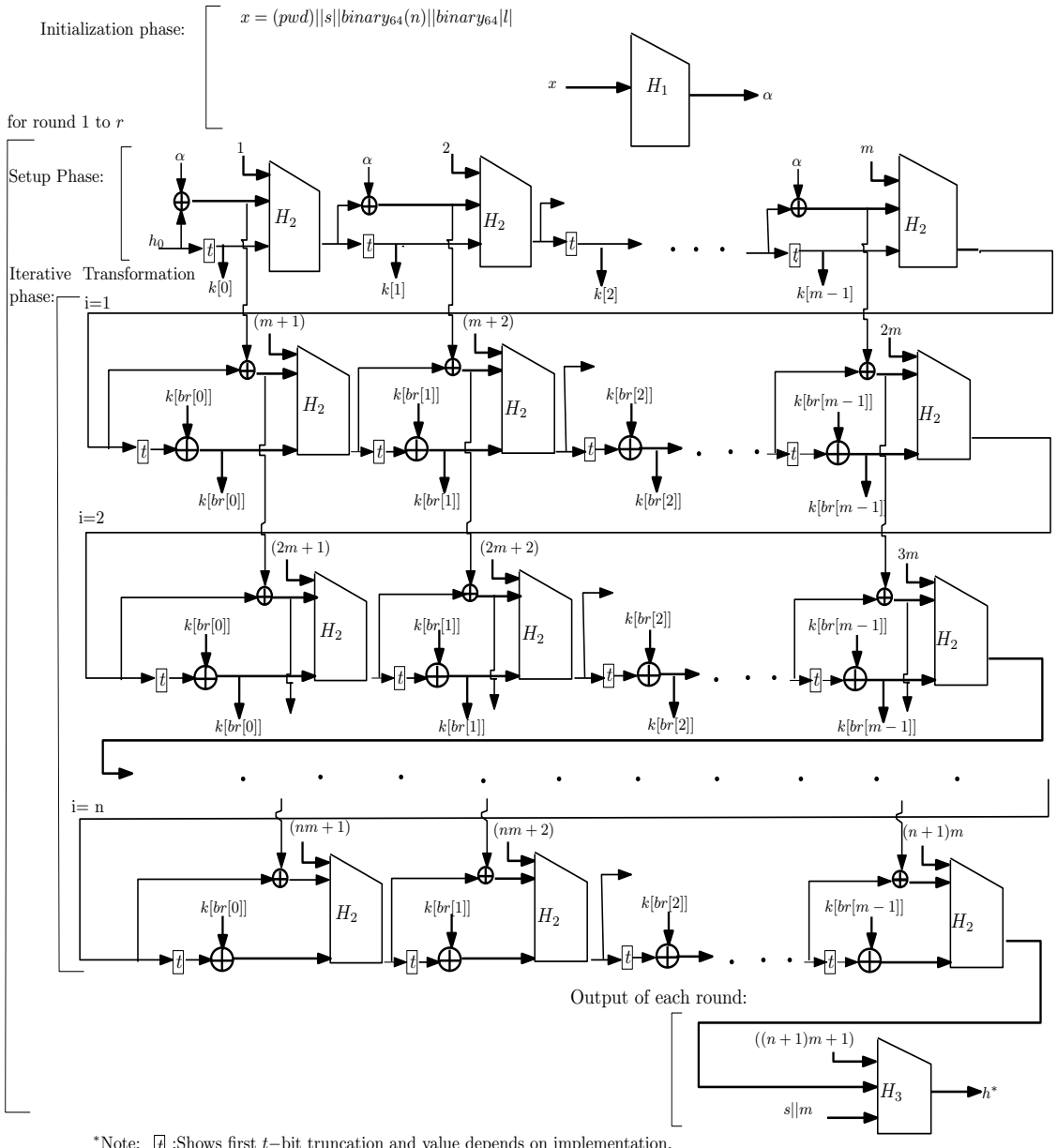

Figure 1: *Graphical representation of the proposed construction.*

7 Implementation aspects

This simple 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 truncate few bits of the intermediate hash computation to reduce the required storage while increasing the number of required iterations n. This way, Rig can be efficient on other low memory devices. We will concentrate on 32-bit or 64-bit processors, typical for PCs.

32-bit or 64-bit processor: To implement our scheme on a machine having 32-bit or 64-bit processor, we do not need any modification in the algorithm. The performance of the algorithm will be very much dependent upon the chosen value of m, n and round r. To get the hardware or software efficiency we can depend upon the existing best known method for calculating the hash and can vary the values of m, n and r accordingly. We describe some recommended values of various parameters later in this report.

Parallelism: All intermediate hash values can not be computed in parallel as they get updated at every iteration and this updation depends on every previous hash calculations. Therefore there is very less scope to parallelize the code maintaining the dependency of the steps.

Hardware suitability: Rig is not suited to be implemented in dedicated hardware if we want parallel execution of multiple instances each consuming comparatively low memory. Hardware implementation demanding huge memory is not economical.

8 Performance Analysis

Rig has been implemented in C# and $C++$ languages in an Intel Core i7 PC. We have used compression function of BLake2b [5] as the underlying hash function. For the setup phase and iterative transformation phase we use Blake2b with reduced number of rounds, but for the first hash calculation we use all 12-rounds of Blake2b compression function. As per the authors of [5], the $G-$ function of Blake2b can be simplified by omitting the constants in G. This change improves the performance and is the one we have implemented in our code. In Table 1 we present the performance analysis of our scheme. In this table we report the memory processing rate when consumable memory and the number of iterations to compute the password hash is fixed. For example, when consumable memory is 15 MB and number of iteration (n=1) the processing rate is 625 MB/sec. Similarly, under 15 MB memory, for ($n = 4$) and for ($n = 6$) the processing rate is 218 MB/sec and 180 MB/sec respectively. These values show that if we increase the number of iteration the processing rate will decrease and the algorithm will take more time for fixed

Memory (MB)	Memory Processing rate (MB/sec)		
	n=1	n=4	n=6
15	625	218	180
30	545	191	147
60	517	181	131
120	528	174	121
240	502	166	115
480	492	161	115
960	478	150	111
1920	460	149	107

Table 1: Performance Analysis of Rig.

memory consumption, which is what is required from a password hashing scheme.

If we fix the number of iterations and vary the consumable memory (in ascending order) then also the processing rate decreases correspondingly favoring our hashing requirement (see for n=1 in table 1)

Suggested Parameters: We propose the following parameters: memory count: 17, $n = 6$ and $r = 1$ for efficient results. We recommend these values because memory count 17 implies 15 MB memory consumption by our algorithm which is preferred.

9 Motivation for design choices

Existing password hashing schemes are not simple and do not fulfill the requirements to be considered as a standard. We tried to design a solution which overcomes the known disadvantages of existing schemes (Bcrypt [6], Scrypt [2] and PBKDF2). The primary concerns against existing proposals are their complex design and their inefficiency to prevent hardware threats. We also tried to strengthen the design by introducing flexibility in the choice of parameters.

9.1 The setup phase

This phase is designed to make use of comparatively huge memory to neutralize the effect of using recent technological trends, such as GPUs which have lots of processing core but limited amount of memory for each single core.

We store two arrays of size m but in one array we have the flexibility to vary the bit storage. In this case we store m number of $t-$bit values which we obtain as the first $t-$bits of the hash output. The value of t is taken to be close to the hash-length, but not equal to the hash-length, to fulfill the demand of huge memory while at the same time ensuring sequential calculation of the hash. Not storing the full hash value ensures that the attacker needs to compute the hash at run-time thus slowing him down.

This flexibility in the choice of length does not affect the security. This flexibility in the choice can reduce the total amount of storage than storing the whole value, but at the same time increases the scope of implementation. We can implement our design in resource constrainted devices. In such cases, we can increase the number of iterations to slow down the process. Another important point is that by storing less than hash-length bits forces each un-stored value (if we choose to store less than the required intermediate hash values) to be calculated from the beginning.

9.2 The iterative transformation phase

In this phase the constant modification of memory increases the dependency and makes the implementation almost sequential. Use of password independent permutation reduces the chance of cache timing attack.

9.3 Output generation phase

We take the $l-$bit hash value as output and for last hash calculation we take the extra input salt(s) to introduces extra security to prevent rainbow attack. Even if one can find collisions in the hash calculations, the attacker can't use it unless the salt is also same.

10 Expected strength

The best known algorithm for password hashing, Scrypt, is not flexible enough as its time complexity: $4Nr$ and memory complexity: Nr depend on each other, i.e., if we reduce the memory usage, it automatically reduces the time to calculate the password hash. This is not the case in our construction. In our construction the time complexity is: $O(mnr)$ and memory complexity is: $O(m)$. Here we have the flexibility that if we decrease the value of m we can increase the number of iterations n to make the process slow.

Our scheme is very simple. It is based on simple cryptographic primitive, hash functions. Like Scrypt it does not need different modules to calculate the password hash. It handles variable length passwords and prefers random salt value of atleast 16-byte of length. It basically does iterative hash calculations and for this it takes different counter values and makes each input different to all others. Different inputs operated on a good hash function intuitively provide different outputs and hence the construction behaves like a Random oracle producing random output.

It is comparatively fast on software and slow on hardware. The algorithm uses comparatively large memory and an attacker who tries to execute multiple instances in parallel requires many copies of such large memory, one for each instance. Hence it is not possible to efficiently utilize all of the processing cores of the hardware. This prevents brute force attacks for password recovery.

Using a decent length (preferable more than 16-bytes) of a unique salt per password makes the rainbow attack quite infeasible.

11 Security Analysis

The basic requirement for a password based design is to be non-invertible and our design follows this property because of two reasons: (i) the iterative use of underlying primitive, the (secure) cryptographic hash function and (ii) the design of the cipher due to the initial hashing of password with random salt and other parameters and final use of salt with chaining data. This shows recovering password from its output is quite challenging.

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 quite infeasible and prevents significant speed up by the use of multiple processing units.

Flexibility of the design makes it unique from existing constructions as time and memory parameters are independent of each other. By the initialization of the memory arrays (setup phase) and the constant need of the stored values throughout the processing, our construction falls in the category of sequential memory hard designs [1].

Attacker approach: An attacker running multiple instances of Rig, may try to do the calculations using smaller part of the memory or almost no-memory to reduce the memory costs 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 calculation but may increase the overall throughput of password guessing as compared to the legitimate implementation of the algorithm. Next we explain how feasible the memory-free approach, from the attacker's point of view.

Attack Scenario: Storing none of the required array values.

Theorem 11.1. *The algorithm can be computed with time complexity $O(mnr)$ 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. Storing none of the $2m$ values the time complexity of the algorithm is bounded by $O(r \times m^{n+1})$ for a single password calculation.*

Proof. The attacker tries the following steps to implement the constant memory attack. First, m- hash calculations for the initial setup phase but the attacker stores only the current chaining value and the value α. So any element in the first setup phase can be computed in effort $O(m)$, and then, at second step, i.e., at the first iteration of iterative transformation phase the effort is $O(m) + (1 + 2 + \cdots + m) \equiv O(m^2)$. Assuming the calculation for values a and k are similar we only consider the expression for getting values of k. Calculating each value at each iteration with $k[r]^i = q_{i-1} \times m + k[r]^{i-1}$ calculations, where q_{i-1} denotes the number of calculations

needed to reach i-th iteration and $k[r]^{i-1}$ is the number of calculations needed to calculate $k[r]$-th value till iteration $i-1$. This we obtain by following the approach that at each iterative transformation phase (for n iterations) whenever the attacker gets the index value of the array, it needs to calculate all the changes for each previous iterations. To check this the attacker needs to run the whole initialization step followed by the $i-1$th (say) step of iterative transformation phase for the calculation of its ith step. So we get the expression at $n-$th iteration $O(m^n) + m * O(m^n)$ which gives the time complexity of $O(r \times m^{n+1})$ where t is the number of rounds. □

11.1 Length extension attack

The length extension attack on hash function works as follows: Hash functions work on blocks of data. Most messages that are hashed will have a length that is not evenly divisible by the hash block length. Thus, the message must be padded to match a multiple of the block length. Let we know the hash of a data without knowing its value except the length of it. So we have: $h(m)$ which is calculated by $m||p$, where p is the padding and value of m is secret. Then take a message m', which begins with $m||p$, i.e., $m' = m||p||x$ where x is a sequence of bits that we can choose arbitrarily. Now we can extend the hash to get $h(m')$. So we get correct hash of $m||p||x$ without knowing the value m.

In our construction we are taking password and if we extend the password by any value, the value of α will change and it will affect all subsequent blocks. So in this case we will not get prefix hash to get the correct hash of the modified password. Thus length extension attack is not feasible in our construction.

11.2 Resilience to cache-timing attack

In our scheme we are storing two arrays, each of length m. Among these two arrays one array access is sequential and the other one uses bit reversal permutation for accessing the values. This permutation does not depend on the value of password. Therefore even if the whole value of the array is stored in cache by an attacker, tracing the access pattern will not yield any advantage to her. This is due to the fact that if the access sequence depends on the value of the password then by tracing the timing it is easy to filter out some of the passwords which are following similar access pattern. Therefore cache timing attack is not possible and hence the scheme is resilience to cache-timing attacks.

Further, if we choose a secure hash function to get the password hash, we can claim the security based on the hash function used. With the current state-of-the-art we have the possibility of using SHA-3, which is resilient to side-channel attacks. Using cryptographically strong hash functions, we can reduce the possibility of cache timing attacks.

11.3 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 use some slow password hashing technique for authentication. To handle such situations, the proposed server-relief technique can relief the server from heavy calculation as the client will do the heavy part of the algorithm. This way we can reduce the chance of Dos attack with slow password hashing schemes.

12 Intellectual property statement

We state that the scheme proposed in this report, Rig, is and will always remain available worldwide on a royalty free basis. Further, we are unaware of any patent or patent application that covers the use or implementation of the submitted algorithm.

References

[1] 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.

[2] Colin Percival. Stronger key derivation via sequential memory-hard functions. In *BSDCon*, 2009. `http://www.bsdcan.org/2009/schedule/attachments/87_scrypt.pdf`.

[3] 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.

[4] Christian Forler, Stefan Lucks, and Jakob Wenzel. Catena: A memory-consuming password scrambler. *IACR Cryptology ePrint Archive*, 2013:525, 2013.

[5] 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.

[6] Niels Provos and David Mazières. A future-adaptable password scheme. In *USENIX Annual Technical Conference, FREENIX Track*, pages 81–91. USENIX, 1999.