A Memory-Hard Function for Password Hashing and Key Derivation

Charles F. de Barros

[1] Departamento de Ciência da Computação
Universidade Federal de São João del-Rei
São João del-Rei – MG – Brazil

charlesbarros@ufsj.edu.br

Abstract. *Key derivation and password scrambling are crucial procedures in cryptographic applications, and the security of these methods against brute-force attacks is a critical concern in face of the increasing computational power available to perform these attacks. This paper proposes a candidate memory-hard function for password scrambling and key derivation, based on some design principles such as flexibility, variable-length output, adjustable parametrization to achieve high cache miss rates and dynamic update of the internal buffer.*

1. Introduction

Widely used password hashing/key derivation algorithms such as PBKDF and PBKDF2 [Josefsson 2011] aim at imposing a certain computational cost to the task of hashing a password or deriving a cryptographic key. As a consequence, any attempt of cracking a password or key will also have a cost, specially if we consider that a brute-force attack consists of a large number of attempts until the correct password/key is found. Usually, these algorithms are based on a huge number of iterations of a hash function, such as SHA256 [Hansen and 3rd 2006], which leads to an increasing amount of time needed to make a lot of attempts, but the storage complexity of these algorithms is usually low. With the increasing capacity of parallelism offered by dedicated hardware devices, time complexity has gradually become less of a problem, as an adversary may perform a large number of attempts simultaneously [Dandass 2008, Kini et al. 2015].

Memory-hard functions, on the other hand, aim at imposing a high storage cost to the task of hashing a password or deriving a key, mitigating attacks based on parallelization techniques. These functions are well known in the literature, including Scrypt [Percival and Josefsson 2016], Catena [Forler et al. 2013], balloon hashing [Boneh et al. 2016], among others. Despite the fundamental differences between each one of these functions, they are based on some general design principles. Two of these principles consist of filling a large array (also called an internal buffer) with pseudorandom values (usually obtained by iterating a hash function) and accessing the entries of this array in a non-sequential, pseudorandom fashion, which makes it mandatory to keep the whole array in memory during the function evaluation.

A typical attack against memory-hard functions, known as *time-space trade-off* [Hellman 1980], consists of trying to compute the function using less storage, while paying the price of a higher time complexity. In this kind of attack, the adversary usually avoids to keep the whole array in memory, computing the needed entries *on the fly*, thus

consuming more time. Some memory-hard functions, such as Lyra [Almeida et al. 2014] and Lyra2 [Jr. et al. 2015] try to increase the complexity of time-space trade-offs by dynamically updating the internal array, which leads to a huge cost to compute its entries on demand.

This paper shows some preliminary results on a work in progress regarding the construction of an efficient memory-hard function, mostly based on the same general principles that underlie existing constructions, but focusing on the idea of updating the internal array (in order to mitigate time-space trade-offs), flexibility (by allowing any desired output size and the use of any hash algorithm) and bandwidth hardness [Blocki et al. 2018], which is strongly dominated by the cache miss rate.

2. Preliminaries

In this section, we establish some basic definitions and notations that will be used throughout this paper. For our purposes, we denote by ℓ the length in bytes of a single word. This length is variable and architecture-dependent. The most typical values are $\ell = 4$ and $\ell = 8$. A cryptographically secure hash function shall be denoted by $\mathrm{H}(x)$, where the input x is an arbitrary-length array of bytes. The output of the hash function will be considered as an m-word array of ℓ-byte words, so that the total size in bytes of the output is equal to ℓm. We also consider concatenation of arrays, denoted by $||$, as an operation that yields a bigger array. Throughout this paper we make use of the function $\mathrm{wParity}$, which takes as input an array of m words and returns a single ℓ-byte word given by

$$\mathrm{wParity}(w_0, w_1, \cdots, w_{m-1}) = w_0 \oplus w_1 \oplus \cdots \oplus w_{m-1},$$

where $\oplus$ denotes the XOR operator.

3. Design Rationale

Before we describe the inner details of the proposed function, we provide some rationale for its design choices. We focused on dynamic updating of the internal buffer, flexibility and bandwidth hardness. We summarize below the main ideas behind the construction of the proposed function:

1. Dynamic update of the internal buffer: the buffer is constantly updated in order to increase the cost of low-memory attacks. This is a significant improvement in regards to most existing memory-hard functions. It also represents a step towards more efficiency, with respect to algorithms such as Lyra2, as the update mechanism replaces the *wandering phase* of Lyra2, that iteratively overwrites several entries, by a single update and creates a dependence of the internal state on the overwritten entries.
2. Flexibility: the proposed algorithm works with any hash construction, with any output size. As soon as a hash construction becomes obsolete, a new one can be used in the algorithm by fine-tuning the parameters. In this aspect the proposed function differs from Lyra2, for example, which essentially depends on the sponge construction. The output length is also variable.
3. High level of dependency between the entries of the internal buffer: the internal buffer has a high level of dependency between non-consecutive entries, including a cyclic dependency between the first and last rows, which significantly increases the cost of low-memory attacks.

4. Bandwidth hardness: the function is designed to produce a high rate of cache misses. The parameters can be adjusted so that consecutive references to the internal buffer never map to the same cache line.

4. Basic steps

The proposed function takes as input a user password P, a random salt S, a word length ℓ, a desired key length k, a cost parameter n and a hash function H whose output length in bytes is equal to $m\ell$. The value of k represents the number of ℓ-byte words of the desired key, so the total number of bytes of the output is equal to ℓk. The parameter n refers to the length of each row (in blocks of m words) of the internal buffer, as we shall see in details in the next subsection.

We followed the two main principles that underlie every memory-hard construction: filling an internal buffer with pseudorandom values, which depend on a user defined password and a random salt, and accessing its entries in a pseudorandom fashion. The dynamic update of the internal buffer was mainly inspired by Lyra and Lyra2, although the update mechanism used in this paper substantially differs from the one used in those functions. Instead of having a *wandering phase* that iteratively overwrites pseudorandom cells, the novelty of this work consists of updating the current entry being accessed, namely $B[i][j]$, and another entry $B[i][\phi(j)]$, which is guaranteed to be sufficiently distant from $B[i][j]$, ensuring the non-locality required to yield a large number of cache misses.

4.1. Filling the internal buffer

The internal buffer is a key component in the proposed function. It can be viewed as a bidimensional array with k rows and mn columns, where each entry is an ℓ-byte word. The i-th row is divided in blocks $B_i^{(j)}$, for $j = 0, 1, \cdots, n-1$, so that each block is a hash value, given by a sequence of m words of ℓ bytes. The general structure of the internal buffer is the following:

$$B = \begin{matrix} B_0^{(0)} & B_0^{(1)} & \cdots & B_0^{(n-1)} \\ B_1^{(0)} & B_1^{(1)} & \cdots & B_1^{(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ B_{k-1}^{(0)} & B_{k-1}^{(1)} & \cdots & B_{k-1}^{(n-1)} \end{matrix} \tag{1}$$

where each block $B_i^{(j)}$ is given by

$$B_i^{(j)} = B[i][jm] \,||\, B[i][jm+1] \,||\, \cdots \,||\, B[i][jm+m-1] \tag{2}$$

To fill the buffer, we followed the principle of creating dependencies between non-consecutive blocks, in order to increase the cost of computing entries on the fly. Even rows are filled in ascending order, while odd rows are filled in descending order. Algorithm 1 shows the inner details of the procedure to fill the internal buffer. Alternating between ascending and descending filling patterns aims at creating an extra difficulty to perform a time-space trade-off. Note that, after we exit the while loop, we perform an extra update on the first row, creating a cyclic dependency between the first and last rows. Hence, computing the first row of the internal buffer requires going through the computation of all other entries. This is intended to significantly increase the cost of computing entries on the fly during a low-memory attack.

Algorithm 1 fillBuffer(P,S,ℓ,m,n,k)

$B_0^{(0)} \leftarrow \mathrm{H}(\mathrm{P} \mid\mid \mathrm{S})$
$B_0^{(1)} \leftarrow \mathrm{H}(\mathrm{P} \mid\mid \mathrm{S} \mid\mid B_0^{(0)})$
for $j \leftarrow 2, \cdots, n-1$ **do**
    $B_0^{(j)} \leftarrow \mathrm{H}(B_0^{(j-1)} \mid\mid B_0^{(j-2)})$
$i \leftarrow 1$
while True **do**
    $B_i^{(n-1)} \leftarrow \mathrm{H}(B_{i-1}^{(n-2)} \mid\mid B_{i-1}^{(n-1)})$
    for $j \leftarrow n-2, \cdots, 1$ **do**
        $B_i^{(j)} \leftarrow \mathrm{H}(B_{i-1}^{(j-1)} \mid\mid B_i^{(j+1)})$
    $B_i^{(0)} \leftarrow \mathrm{H}(B_{i-1}^{(0)} \mid\mid B_i^{(1)})$
    $i \leftarrow i+1$
    if $i == k$ **then**
        Break
    $B_i^{(0)} \leftarrow \mathrm{H}(B_{i-1}^{(0)} \mid\mid B_{i-1}^{(1)})$
    for $j \leftarrow 1, \cdots, n-2$ **do**
        $B_i^{(j)} \leftarrow \mathrm{H}(B_{i-1}^{(j+1)} \mid\mid B_i^{(j-1)})$
    $B_i^{(n-1)} \leftarrow \mathrm{H}(B_{i-1}^{(n-1)} \mid\mid B_i^{(n-2)})$
    $i \leftarrow i+1$
$B_0^{(0)} \leftarrow B_0^{(0)} \oplus \mathrm{H}(B_{k-1}^{(0)} \mid\mid B_{k-1}^{(1)})$
for $j \leftarrow 1, \cdots, n-2$ **do**
    $B_0^{(j)} \leftarrow B_0^{(j)} \oplus \mathrm{H}(B_{k-1}^{(j+1)} \mid\mid B_0^{(j-1)})$
$B_0^{(n-1)} \leftarrow B_0^{(n-1)} \oplus \mathrm{H}(B_{k-1}^{(n-1)} \mid\mid B_0^{(n-2)})$
Return B

4.2. Updating The Internal State

The internal state is represented by an array of k words of size ℓ. Hence, its total number of bytes equals $k\ell$. It will be denoted by the vector

$$S = (s_0, s_1, \cdots, s_{k-1})$$

The desired key will be given by the final value of the internal state. Each round of the proposed function updates a single word of the internal state, iterating through a single row of the internal buffer. To update the i-th word of the internal state, we iterate through the i-th row of the buffer. One of the core design principles of the procedure consists of maximizing the cache miss rate. In order to achieve this goal, the function takes as input the size c (in words) of the cache line on the target architecture, so that two consecutive iterations never make reference to the same cache line, thus breaking spatial locality and increasing the computational cost of evaluating the function.

Algorithm 2 shows the details of the internal state update. It takes as input the internal buffer B, a hash algorithm H, the desired key length k, the size of the hash output m, the cost parameter n and the size c of the target cache line. The procedure to update the internal state makes use of an indexing function ϕ, which is a mapping over the set of indices $I = \{0, 1, \cdots, nm-1\}$ with the following properties:

- Non-locality: there must be a huge gap between x and $\phi(x)$, for all $x \in I$. Formally speaking, we must have $|x - \phi(x)| \geq c$ for all $x \in I$.
- Bipartiteness: it is an extension of non-locality, implying that $\phi(x) \geq nm/2$ if $x < nm/2$, and $\phi(x) < nm/2$ for all $x \geq nm/2$.
- Spreadedness: there is a huge gap between $\phi(x)$ and $\phi(x+c)$, which means that $|\phi(x) - \phi(x+c)| \geq c$ for all $x \in I$.
- Input-independence: the mapping is independent of the password and the salt, in order to avoid side-channel attacks.

Non-locality ensures that each iteration modifies two entries that are distant enough, which contributes to increase the cache miss rate and the cost of computing entries on demand in a time-space trade-off. Bipartiteness is just a specialization of non-locality,

Algorithm 2 updateState(B,H,k,m,n,c)

for $i \leftarrow 0, 1, \cdots, k-1$ **do**
  $S[i] \leftarrow \text{wParity}(\text{H}(B[i][0]))$
  for $w \leftarrow 0, 1, \cdots, c-1$ **do**
    $r \leftarrow \sigma(w)$ ▷ σ is a pseudorandom permutation
    for $l \leftarrow 0, 1, \cdots, nm/c-1$ **do**
      $j \leftarrow r + \pi(l)c$ ▷ π is another pseudorandom permutation
      $x \leftarrow B[i][j] \oplus B[i][\phi(j)]$
      $S[i] \leftarrow S[i] \oplus \text{wParity}(\text{H}(x))$
      $B[i][j] \leftarrow B[i][j] \oplus S[i]$
      $B[i][\phi(j)] \leftarrow B[i][\phi(j)] \oplus B[i][j]$
return $S[0] \,||\, S[1] \,||\, \cdots \,||\, S[k-1]$

making sure that the second half of the internal buffer is modified during the access of its first half. Finally, spreadedness is another strategy to maximize the cache miss rate. Input-independence aims to prevent side-channel attacks based on timing techniques, also allowing a quick discard of the password and the salt, which helps to avoid dumping attacks. It is worth mentioning that an adversary could try to take advantage of this input-independence by filling the buffer in a different order, in an attempt to increase the number of cache hits during the internal state update. However, we argue that this approach is infeasible. Firstly, the adversary could try to relocate the buffer entries $B[i][j]$ and $B[i][j+\pi(l)c]$ for consecutive values of l, in order to have more cache hits in consecutive iterations of the inner loop. However, because j is defined as a function σ which is a pseudorandom permutation of the possible values of w, and also because the increment of j is given by a pseudorandom value (given by the permutation π), the adversary has no way of knowing in advance which entries should be relocated in order to yield more cache hits. Furthermore, if the adversary tries to relocate the entries $B[i][j]$ and $B[i][\phi(j)]$, it would probably break the spatial locality of consecutive iterations of the inner loop, resulting in more cache misses.

Finally, in order to give an estimate on the cost of a low-memory attack, assume that an adversary wishes to compute the buffer entries on the fly during the internal state update. We note that the same block will be accessed multiple times, but its individual words will have been modified, which also contributes to increase the cost of a low-memory approach. Because each buffer entry is modified during the internal state update, the execution of any iteration of the internal state update procedure by an adversary requires the recomputation of the current buffer entry, which depends on previous entries that have been already modified in previous iterations. Hence, every iteration of the external loop requires the whole execution of all previous iterations, and this will have to be done for all the iterations of the inner loop (considering a *low-memory adversary*, with no extra storage available).

5. Future Work and Final Remarks

This paper presented a work in progress regarding the construction of a memory-hard function. The proposed function is based on some well-known design principles that underlie similar constructions, but focuses on flexibility, by allowing the use of any hash

algorithm and any key size, as well as bandwidth hardness by presenting high cache miss rates during its evaluation. Future works include practical instantiations of the mapping function ϕ and efficient implementations, as well as the finalization of the formal proofs of memory-hardness and bandwidth-hardness for the proposed construction.

Acknowledgments

We would like to thank to all the reviewers for their valuable comments, suggestions and recommendations, which significantly contributed to improve the quality and strength of this proposal.

References

Almeida, L. C., Andrade, E. R., Barreto, P. S. L. M., and Jr., M. A. S. (2014). Lyra: Password-based key derivation with tunable memory and processing costs. Cryptology ePrint Archive, Paper 2014/030. `https://eprint.iacr.org/2014/030`.

Blocki, J., Ren, L., and Zhou, S. (2018). Bandwidth-hard functions: Reductions and lower bounds. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1820–1836, New York, NY, USA. Association for Computing Machinery.

Boneh, D., Corrigan-Gibbs, H., and Schechter, S. (2016). Balloon hashing: A memory-hard function providing provable protection against sequential attacks. In Cheon, J. H. and Takagi, T., editors, *Advances in Cryptology – ASIACRYPT 2016*, pages 220–248, Berlin, Heidelberg. Springer Berlin Heidelberg.

Dandass, Y. S. (2008). Using FPGAs to parallelize dictionary attacks for password cracking. In *Proceedings of the 41st Annual Hawaii International Conference on System Sciences (HICSS 2008)*, pages 485–485.

Forler, C., Lucks, S., and Wenzel, J. (2013). Catena: A memory-consuming password-scrambling framework. Cryptology ePrint Archive, Paper 2013/525. `https://eprint.iacr.org/2013/525`.

Hansen, T. and 3rd, D. E. E. (2006). US Secure Hash Algorithms (SHA and HMAC-SHA). RFC 4634.

Hellman, M. (1980). A cryptanalytic time-memory trade-off. *IEEE Transactions on Information Theory*, 26(4):401–406.

Josefsson, S. (2011). PKCS #5: Password-Based Key Derivation Function 2 (PBKDF2) Test Vectors. RFC 6070.

Jr., M. A. S., Almeida, L. C., Andrade, E. R., dos Santos, P. C. F., and Barreto, P. S. L. M. (2015). Lyra2: Efficient password hashing with high security against time-memory trade-offs. Cryptology ePrint Archive, Paper 2015/136. `https://eprint.iacr.org/2015/136`.

Kini, N. G., Paleppady, R., and Naik, A. K. (2015). Password cracking on graphics processing unit based systems. *International Journal of Humanities and Social Sciences*, 9(12):2442 – 2445.

Percival, C. and Josefsson, S. (2016). RFC 7914: The scrypt password-based key derivation function.