Lyra2: Password Hashing Scheme with improved security against time-memory trade-offs

Marcos A. Simplicio Jr., Leonardo C. Almeida, Ewerton R. Andrade,
Paulo C. F. dos Santos, and Paulo S. L. M. Barreto

Escola Politécnica – Universidade de São Paulo (Poli-USP), São Paulo, Brazil.
{mjunior,lalmeida,eandrade,psantos,pbarreto}@larc.usp.br

Abstract. We present Lyra2, a password hashing scheme (PHS) based on cryptographic sponges. Lyra2 was designed to be strictly sequential (i.e., not easily parallelizable), providing strong security even against attackers that uses multiple processing cores (e.g., custom hardware or a powerful GPU). At the same time, it is very simple to implement in software and allows legitimate users to fine tune its memory and processing costs according to the desired level of security against brute force password-guessing. Lyra2 is an improvement of the recently proposed Lyra algorithm, providing an even higher security level against different attack venues and overcoming some limitations of this and other existing schemes.

Keywords: Password hashing, processing time, memory usage, cryptographic sponges

Note 1. *This paper corresponds to version 1 of Lyra2, originally submitted to the Password Hashing Competition (https: // password-hashing. net/).*

1 Introduction

User authentication is one of the most vital elements in modern computer security. Even though there are authentication mechanisms based on biometric devices ("what the user is") or physical devices such as smart cards ("what the user has"), the most widespread strategy still is to rely on secret passwords ("what the user knows"). This happens because password-based authentication remains as the most cost effective and efficient method of maintaining a shared secret between a user and a computer system [1,2]. For better or for worse, and despite the existence of many proposals for their replacement [3], this prevalence of passwords as one and commonly only factor for user authentication is unlikely to change in the near future.

Password-based systems usually employ some cryptographic algorithm that allows the generation of a pseudorandom string of bits from the password itself, known as a password hashing scheme (PHS), or key derivation function (KDF) [4]. Typically, the output of the PHS is employed in one of two manners [5]: it can

be locally stored in the form of a "token" for future verifications of the password or used as the secret key for encrypting and/or authenticating data. Whichever the case, such solutions employ internally a one-way (e.g., hash) function, so that recovering the password from the PHS's output is computationally infeasible [5,6].

Despite the popularity of password-based authentication, the fact that most users choose quite short and simple strings as passwords leads to a serious issue: they commonly have much less entropy than typically required by cryptographic keys [7]. Indeed, a study from 2007 with 544,960 passwords from real users has shown an average entropy of approximately 40.5 bits [8], against the 128 bits usually required by modern systems. Such weak passwords greatly facilitate many kinds of "brute-force" attacks, such as dictionary attacks and exhaustive search [1,9], allowing attackers to completely bypass the non-invertibility property of the password hashing process. For example, an attacker could apply the PHS over a list of common passwords until the result matches the locally stored token or the valid encryption/authentication key. The feasibility of such attacks depends basically on the amount of resources available to the attacker, who can speed up the process by performing many tests in parallel. Such attacks commonly benefit from platforms equipped with many processing cores, such as modern GPUs [10,11] or custom hardware [11,12].

A straightforward approach for addressing this problem is to force users to choose complex passwords. This is unadvised, however, because such passwords would be harder to memorize and, thus, more easily forgotten or stolen due to the users' need of writing them down, defeating the whole purpose of authentication [1]. For this reason, modern password hashing solutions usually employ mechanisms for increasing the *cost* of brute force attacks. Schemes such as PBKDF2 [6] and bcrypt [13], for example, include a configurable parameter that controls the number of iterations performed, allowing the user to adjust the time required by the password hashing process. A more recent proposal, scrypt [5], allows users to control both processing time and memory usage, raising the cost of password recovery by increasing the silicon space required for running the PHS in custom hardware, or the amount of RAM required in a GPU. There is, however, considerable interest in the research community in developing new (and better) alternatives, which recently led to the creation of a competition with this specific purpose [14].

Aiming to address this need for stronger alternatives, our studies led to the proposal of Lyra [15], a mode of operation of cryptographic sponges [16,17] for password hashing. In this article, we propose an improved version of Lyra, called simply Lyra2. Lyra2 preserves the security, efficiency and flexibility of Lyra, including: (1) the ability to configure the desired amount of memory, processing time and parallelism to be used by the algorithm; (2) the capacity of providing a high memory usage with a processing time similar to that obtained with scrypt; In addition, it brings important improvements when compared to its predecessor: (1) it allows a higher security level against attack venues involving time-memory trade-offs; (2) it allows legitimate users to benefit more effectively

from the parallelism capabilities of their own platforms; (3) it includes tweaks for increasing the costs involved in the construction of dedicated hardware to attack the algorithm.

The rest of this paper is organized as follows. Section 2 outlines the concept of cryptographic sponges. Section 3 describes the main requirements of PHS solutions and discusses the related work. Section 4 presents the Lyra2 algorithm and its design rationale, while Section 5 analyses its security. Section 6 discusses some possible extensions of Lyra2, all of which can be integrated into the basic algorithm discussed in Section 4. Section 7 shows our preliminary benchmark results. Finally, Section 8 presents our final remarks.

2 Cryptographic Sponges

The concept of *cryptographic sponges* was formally introduced by Bertoni *et al.* in [16] and is described in detail in [17]. The elegant design of sponges has also motivated the creation of more general structures, such as the Parazoa family of functions [18]. Indeed, their flexibility is probably among the reasons that led Keccak [19], one of the members of the sponge family, to be elected as the new Secure Hash Algorithm (SHA-3).

In what follows and throughout this document, we use the following notation: $\oplus$ stands for the XOR operation; $\|$ denotes concatenation; $|x|$ represents the bit-length of x, i.e., the minimum number of bits required for representing it; similarly, $len(x)$ represents the byte-length of x; $truncL(x, y)$ and $truncM(x, y)$ denote, respectively, the truncation of x to its least and most significant y bits; and $Int(x, y)$ denotes the y-bit representation of number x. All operations are made considering the little-endian convention.

2.1 Cryptographic Sponges: Basic Structure

In a nutshell, sponge functions provide an interesting way of building hash functions with arbitrary input and output lengths. Such functions are based on the so-called sponge construction, an iterated mode of operation that uses a fixed-length permutation (or transformation) f and a padding rule `pad`. More specifically, and as depicted in Figure 1, sponge functions rely on an internal state of
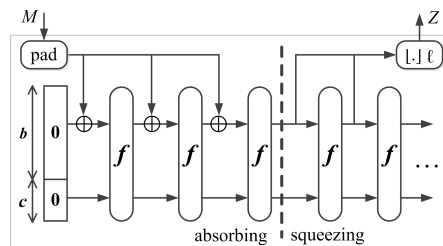

Fig. 1. Overview of the sponge construction $Z = [f, \mathtt{pad}, b](M, \ell)$. Adapted from [17].

$w = b + c$ bits, initially set to zero, and operate on an (padded) input M cut into b-bit blocks. This is done by iteratively applying f to the sponge's internal state, operation interleaved with the entry of input bits (during the *absorbing* phase) or the subsequent retrieval of output bits (during the *squeezing* phase). The process stops when all input bits consumed in the *absorbing* phase are mapped into the resulting ℓ-bit output string. Typically, the f transformation is itself iterative, being parameterized by a number of rounds (e.g., 24 for Keccak operating with 64-bit words [19]).

The sponge's internal state is, thus, composed by two parts: the b-bit long outer part, which interacts directly with the sponge's input, and the c-bit long inner part, which is only affected by the input by means of the f transformation. The parameters w, b and c are called, respectively, the *width*, *bitrate*, and the *capacity* of the sponge.

2.2 The duplex construction

A similar structure derived from the sponge concept is the *Duplex construction* [17], depicted in Figure 2.
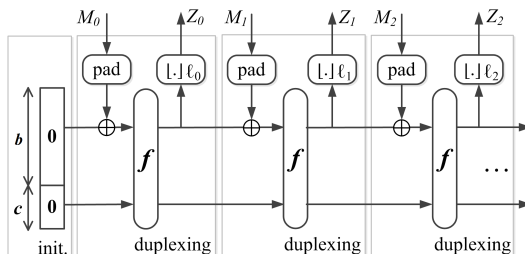

Fig. 2. Overview of the duplex construction. Adapted from [17].

Unlike regular sponges, which are stateless in between calls, a duplex function is stateful: it takes a variable-length input string and provides a variable-length output that depends on all inputs received so far. In other words, although the internal state of a duplex function is filled with zeros upon initialization, it is stored after each call to the duplex object rather than repeatedly reset. In this case, the input string M must be short enough to fit in a single b-bit block after padding, and the output length ℓ must satisfy $\ell \leqslant b$.

3 Password Hashing Schemes (PHS)

As previously discussed, the basic requirement for a PHS is to be non-invertible, so that recovering the password from its output is computationally infeasible. Moreover, a good PHS's output is expected to be indistinguishable from random bit strings, preventing an attacker from discarding part of the password space

based on perceived patterns [20]. In principle, those requirements can be easily accomplished simply by using a secure hash function, which by itself ensures that the best attack venue against the derived key is through brute force (possibly aided by a dictionary or "usual" password structures [7,21]).

What any modern PHS do, then, is to include techniques that raise the cost of brute-force attacks. A first strategy for accomplishing this is to take as input not only the user-memorizable password *pwd* itself, but also a sequence of random bits known as *salt*. The presence of such random variable thwarts several attacks based on pre-built tables of common passwords, i.e., the attacker is forced to create a new table from scratch for every different *salt* [20,6]. The *salt* can, thus, be seen as an index into a large set of possible keys derived from *pwd*, and need not to be memorized or kept secret [6].

A second strategy is to purposely raise the cost of every password guess in terms of computational resources, such as processing time and/or memory usage. This certainly also raises the cost of authenticating a legitimate user entering the correct password, meaning that the algorithm needs to be configured so that the burden placed on the target platform is minimally noticeable by humans. Therefore, the legitimate users and their platforms are ultimately what impose an upper limit on how computationally expensive the PHS can be for themselves and for attackers. For example, a human user running a single PHS instance is unlikely to consider a nuisance that the password hashing process takes 1 s to run and uses a small part of the machine's free memory, e.g., 20 MB. On the other hand, supposing that the password hashing process cannot be divided into smaller parallelizable tasks, achieving a throughput of 1,000 passwords tested per second requires 20 GB of memory and 1,000 processing units as powerful as that of the legitimate user.

A third strategy, especially useful when the PHS involves both processing time and memory usage, is to use a design with low parallelizability. The reasoning is as follows. For an attacker with access to p processing cores, there is usually no difference between assigning one password guess to each core or parallelizing a single guess so it is processed p times faster: in both scenarios, the total password guessing throughput is the same. However, a sequential design that involves configurable memory usage imposes an interesting penalty to attackers who do not have enough *memory* for running the p guesses in parallel. For example, suppose that testing a guess involves m bytes of memory and the execution of n instructions. Suppose also that the attacker's device has $100m$ bytes of memory and 1000 cores, and that each core executes n instructions per second. In this scenario, up to 100 guesses can be tested per second against a strictly sequential algorithm (one per core), the other 900 cores remaining idle because they have no memory to run.

Aiming to provide a deeper understanding on the challenges faced by PHS solutions, in what follows we discuss the main characteristics of platforms used by attackers and then how existing solutions avoid those threats.

3.1 Attack platforms

The most dangerous threats faced by any PHS comes from platforms that benefit from "economies of scale", especially when cheap, massively parallel hardware is available. The most prominent examples of such platforms are Graphics Processing Units (GPUs) and custom hardware synthesized from FPGAs [11].

Graphics Processing Units (GPUs). Following the increasing demand for high-definition real-time rendering, Graphics Processing Units (GPUs) have traditionally carried a large number of processing cores, boosting its parallelization capability. Only more recently, however, GPUs evolved from specific platforms into devices for universal computation and started to give support to standardized languages that help harness their computational power, such as CUDA [22] and OpenCL [23]). As a result, they became more intensively employed for more general purposes, including password cracking [10,11].

As modern GPUs include a few thousands processing cores in a single piece of equipment, the task of executing multiple threads in parallel becomes simple and cheap. They are, thus, ideal when the goal is to test multiple passwords independently or to parallelize a PHS's internal instructions. For example, NVidia's Tesla K20X, one of the top GPUs available, has a total of 2,688 processing cores operating at 732 MHz, as well as 6 GB of shared DRAM with a bandwidth of 250 GB per second [24]. Its computational power can also be further expanded by using the host machine's resources [22], although this is also likely to limit the memory throughput. Supposing this GPU is used to attack a PHS whose parametrization makes it run in 1 s and take less than 2.23 MB of memory, it is easy to conceive an implementation that tests 2,688 passwords per second. With a higher memory usage, however, this number is deemed to drop due to the GPU's memory limit of 6 GB. For example, if a sequential PHS requires 20 MB of DRAM, the maximum number of cores that could be used simultaneously becomes 300, only 11% of the total available.

Field Programmable Gate Arrays (FPGAs). An FPGA is a collection of configurable logic blocks wired together and with memory elements, forming a programmable and high-performance integrated circuit. In addition, as such devices are configured to perform a specific task, they can be highly optimized for its purpose (e.g., using pipelining [25,26]). Hence, as long as enough resources (i.e., logic gates and memory) are available in the underlying hardware, FPGAs potentially yield a more cost-effective solution than what would be achieved with a general-purpose CPU of similar cost [12]. When compared to GPUs, FPGAs may also be advantageous due to the latter's considerably lower energy consumption [27,28], which can be further reduced if its circuit is synthesized in the form of custom logic hardware (ASIC) [27].

A recent example of password cracking using FPGAs is presented in [11]. Using a RIVYERA S3-5000 cluster [29] with 128 FPGAs against PBKDF2-SHA-512, the authors reported a throughput of 356,352 passwords tested per

second in an architecture having 5,376 password processed in parallel. It is interesting to notice that one of the reasons that made these results possible is the small memory usage of the PBKDF2 algorithm, as most of the underlying SHA-2 processing is performed using the device's memory cache (much faster than DRAM) [11, Sec. 4.2]. Against a PHS requiring 20 MB to run, for example, the resulting throughput would presumably be much lower, especially considering that the FPGAs employed can have up to 64 MB of DRAM [29] and, thus, up to three passwords can be processed in parallel rather than 5,376.

Interestingly, a PHS that requires a similar memory usage would be troublesome even for state-of-the-art clusters, such as the newer RIVYERA V7-2000T [30]. This powerful cluster carries up to four Xilinx Virtex-7 FPGAs and up to 128 GB of shared DRAM, in addition to the 20 GB available in each FPGA [30]. Despite being much more powerful, in principle it would still be unable to test more than 2,600 passwords in parallel against a PHS that strictly requires 20 MB to run.

3.2 Scrypt

Arguably, the main password hashing solutions available in the literature are [14]: PBKDF2 [6], bcrypt [13] and scrypt [5]. Since scrypt is only PHS among them that explores both memory and processing costs and, thus, is directly comparable to Lyra2, its main characteristics are described in what follows. For the interested reader, a discussion on PBKDF2 and bcrypt is provided in the appendices.

The design of scrypt [5] focus on coupling memory and time costs. For this, scrypt employs the concept of "sequential memory-hard" functions: an algorithm that asymptotically uses almost as much memory as it requires operations and for which a parallel implementation cannot asymptotically obtain a significantly lower cost. As a consequence, if the number of operations and the amount of memory used in the regular operation of the algorithm are both $\mathcal{O}(R)$, the complexity of a memory-free attack (i.e., an attack for which the memory usage is reduced to $\mathcal{O}(1)$) becomes $\Omega(R^2)$, where R is a system parameter. We refer the reader to [5] for a more formal definition.

The following steps compose scrypt's operation (see Algorithm 1). First, it initializes p b-long memory blocks B_i. This is done using the PBKDF2 algorithm with HMAC-SHA-256 [31] as underlying hash function and a single iteration. Then, each B_i is processed (incrementally or in parallel) by the sequential memory-hard $ROMix$ function. Basically, $ROMix$ initializes an array M of R b-long elements by iteratively hashing B_i. It then visits R positions of M at random, updating the internal state variable X during this (strictly sequential) process in order to ascertain that those positions are indeed available in memory. The hash function employed by $ROMix$ is called $BlockMix$, which emulates a function having arbitrary (b-long) input and output lengths; this is done using the Salsa20/8 [32] stream cipher, whose output length is $h = 512$. After the p $ROMix$ processes are over, the B_i blocks are used as salt in one final iteration of the PBKDF2 algorithm, outputting key K.

Algorithm 1 Scrypt.

PARAM: h ▷ *BlockMix*'s internal hash function output length
INPUT: pwd ▷ The password
INPUT: $salt$ ▷ A random salt
INPUT: k ▷ The key length
INPUT: b ▷ The block size, satisfying $b = 2r \cdot h$
INPUT: R ▷ Cost parameter (memory usage and processing time)
INPUT: p ▷ Parallelism parameter
OUTPUT: K ▷ The password-derived key

1: $(B_0...B_{p-1}) \leftarrow \text{PBKDF2}_{HMAC-SHA-256}(pwd, salt, 1, p \cdot b)$
2: **for** $i \leftarrow 0$ **to** $p - 1$ **do**
3:     $B_i \leftarrow \text{ROMIX}(B_i, R)$
4: **end for**
5: $K \leftarrow \text{PBKDF2}_{HMAC-SHA-256}(pwd, B_0 \parallel B_1 \parallel ... \parallel B_{p-1}, 1, k)$
6: **return** K ▷ Outputs the k-long key

7: **function** ROMIX(B, R) ▷ Sequential memory-hard function
8:     $X \leftarrow B$
9:     **for** $i \leftarrow 0$ **to** $R - 1$ **do**   ▷ Initializes memory array M
10:       $V_i \leftarrow X$; $X \leftarrow \text{BLOCKMIX}(X)$
11:     **end for**
12:     **for** $i \leftarrow 0$ **to** $R - 1$ **do**   ▷ Reads random positions of M
13:       $j \leftarrow Integerify(X) \bmod R$
14:       $X \leftarrow \text{BLOCKMIX}(X \oplus M_j)$
15:     **end for**
16:     **return** X
17: **end function**

18: **function** BLOCKMIX(B) ▷ b-long in/output hash function
19:     $Z \leftarrow B_{2r-1}$ ▷ $r = b/2h$, where $h = 512$ for Salsa20/8
20:     **for** $i \leftarrow 0$ **to** $2r - 1$ **do**
21:       $Z \leftarrow Hash(Z \oplus B_i)$; $Y_i \leftarrow Z$
22:     **end for**
23:     **return** $(Y_0, Y_2, ..., Y_{2r-2}, Y_1, Y_3, Y_{2r-1})$
24: **end function**

Scrypt displays a very interesting design, being one of the few existing solutions that allow the configuration of both processing and memory costs. One of its main shortcomings is probably the fact that it strongly couples memory and processing requirements for a legitimate user. Specifically, scrypt's design prevents users from raising the algorithm's processing time while maintaining a fixed amount of memory usage, unless they are willing to raise the p parameter and allow further parallelism to be exploited by attackers. Another inconvenience with scrypt is the fact that it employs two different underlying hash functions, HMAC-SHA-256 (for the PBKDF2 algorithm) and Salsa20/8 (as the core of the *BlockMix* function), leading to increased implementation complexity. Finally, even though Salsa20/8's known vulnerabilities [33] are not expected to put the security of scrypt in hazard [5], using a stronger alternative would be

at least advisable, especially considering that the scheme's structure does not impose serious restrictions on the internal hash algorithm used by *BlockMix*. In this case, a sponge function could itself be an alternative. However, sponges' intrinsic properties make some of scrypt's operations unnecessary: since sponges support inputs and outputs of any length, the whole *BlockMix* structure could be replaced; in addition, sponges can operate in the stateful and sequential duplexing mode, meaning that the state variable X used by *ROMix* would become redundant.

Inspired by scrypt's design, Lyra2 builds on the properties of sponges to provide not only a simpler, but also more secure solution. Indeed, Lyra2 stays on the "strong" side of the memory-hardness concept: the processing cost of attacks involving less memory than specified by the algorithm grows much faster than quadratically, surpassing the best achievable with scrypt and effectively preventing any (useful) time-memory trade-off. This characteristic greatly discourages attackers from trading memory usage for processing time, which is exactly the goal of a PHS in which usage of both resources are configurable. In addition, Lyra2 allows for a higher memory usage for a similar processing time, increasing the cost of any possible attack venue beyond that of scrypt's.

4   Lyra2

As any PHS, Lyra2 takes as input a salt and a password, creating a pseudorandom output that can then be used as key material for cryptographic algorithms or as an authentication string [4]. Internally, the scheme's memory is organized as a matrix that is expected to remain in memory during the whole password hashing process: since its cells are iteratively read and written, discarding a cell for saving memory leads to the need of recomputing it from scratch, until the point it was last modified, whenever that cell is accessed once again. The construction and visitation of the matrix is done using a stateful combination of the absorbing, squeezing and duplexing operations of the underlying sponge (i.e., its internal state is never reset to zeros), ensuring the sequential nature of the whole process. Also, the number of times the matrix's cells are revisited after initialization is defined by the user, allowing Lyra2's execution time to be fine-tuned according to the target platform's resources.

In this section, we describe the Lyra2 algorithm in detail and discuss its design rationale and resulting properties.

4.1   Structure and rationale

Lyra2's steps are shown in Algorithm 2. As highlighted in the pseudocode's comments, the PHS's operation is composed by three sequential phases: Setup, Wandering and Wrap-up.

The Setup phase   The first part of the algorithm is the *Setup Phase* (lines 1 to 15). This phase comprises the construction of a $R \times C$ memory matrix whose

cells are b-long blocks, where R and C are user-defined parameters and b is the underlying sponge's bitrate (in bits).

Algorithm 2 The Lyra2 Algorithm.

PARAM: H ▷ Sponge with block size b (in bits) and underlying permutation f
PARAM: ρ ▷ Number of rounds of f during the Setup and Wandering phases
PARAM: W ▷ The target machine's word size (usually, 32 or 64)
INPUT: pwd ▷ The password
INPUT: $salt$ ▷ A salt
INPUT: T ▷ Time cost, in number of iterations
INPUT: R ▷ Number of rows in the memory matrix
INPUT: C ▷ Number of columns in the memory matrix ($C \geqslant 2$)
INPUT: k ▷ The desired key length, in bits
OUTPUT: K ▷ The password-derived k-long key

1: ▷ **Setup phase**: Initializes a ($R \times C$) memory matrix. The matrix's cells have b bits each
2:    ▷ Byte representation of input parameters (others can be added)
3: $basil \leftarrow len(k) \, \| \, len(pwd) \, \| \, len(salt) \, \| \, T \, \| \, R \, \| \, C$
4: $H.absorb(\mathtt{pad}(pwd \, \| \, salt \, \| \, basil))$ ▷ Padding rule: 10^*1.
5: $M[0] \, \| \, M[1] \leftarrow H.squeeze_\rho(2C \cdot b)$ ▷ Initializes the first and second rows
6: $row^* \leftarrow 0$; $prev \leftarrow 1$; $row \leftarrow 2$ ▷ The first and second rows will feed the sponge
7: **do**   ▷ **Filling Loop**: initializes remainder rows
8:     **for** $col \leftarrow 0$ **to** $C - 1$ **do**   ▷ **Columns Loop**: updates both $M[row]$ and $M[row^*]$
9:         $rand \leftarrow H.duplexing_\rho(M[prev][col] \oplus M[row^*][col], b)$
10:         $M[row][col] \leftarrow rand$
11:         $M[row^*][col] \leftarrow M[row^*][col] \oplus rotW(rand)$ ▷ $rotW()$: left rotation
12:     **end for**
13:     **if** $row^* \neq 0$ **then** $row^* \leftarrow row^* - 1$ **else** $row^* \leftarrow prev$ **end if**
14:     $prev \leftarrow row$; $row \leftarrow row + 1$ ▷ The next row in sequence will be initialized
15: **while** ($row \leqslant R - 1$)

16: ▷ **Wandering phase**: Iteratively overwrites cells of the memory matrix
17: $dir \leftarrow -1$; $prev \leftarrow 0$; $row \leftarrow R - 1$ ▷ Start visiting rows in reverse order
18: **for** $\tau \leftarrow 1$ **to** T **do**   ▷ **Time Loop**
19:     **do**   ▷ **Visitation Loop**: rows read in reverse order from before
20:         $row^* \leftarrow (truncL(rand, W) \oplus prev) \bmod R$ ▷ A random row will be visited
21:         **for** $col \leftarrow 0$ **to** $C - 1$ **do**   ▷ **Columns Loop**: updates both $M[row]$ and $M[row^*]$
22:             $rand \leftarrow H.duplexing_\rho(M[prev][col] \oplus M[row^*][col], b)$
23:             $M[row][col] \leftarrow M[row][col] \oplus rand$
24:             $M[row^*][col] \leftarrow M[row^*][col] \oplus rotW(rand)$
25:         **end for**
26:         $prev \leftarrow row$; $row \leftarrow row + dir$ ▷ The next row in sequence will be visited
27:     **while** ($0 \leqslant row \leqslant R - 1$)
28:     $dir \leftarrow -dir$; $prev \leftarrow R - row$; $row \leftarrow row + dir$
29: **end for**

30: ▷ **Wrap-up phase**: key computation
31: $H.absorb(M[row^*][0])$ ▷ Absorbs a final column with a full-round sponge
32: $K \leftarrow H.squeeze(k)$ ▷ Squeezes k bits with a full-round sponge

33: **return** K ▷ Provides k-long bitstring as output

The Setup phase starts when the sponge absorbs the (properly padded) password and salt, together with an optional *basil* bitstring, initializing a *salt-* and *pwd*-dependent internal state (line 4). The padding rule adopted by Lyra2 is the multi-rate padding pad10*1 described in [17], hereby denoted simply pad, which appends a single bit 1 followed by as many bits 0 as necessary followed by a single bit 1, so that at least 2 bits are appended. In this first absorb operation, the goal of the *basil* bitstring is basically to avoid collisions with some trivial combinations of salts an passwords: for example, for any $(u, v \mid u + v = \alpha)$, we have a collision if $pwd = 0^u$, $salt = 0^v$ and *basil* is an empty string; however, this should not occur if *basil* explicitly includes u and v. Therefore, *basil* can be seen as an "extension" of the salt, which can include any length of additional information such as: the list of parameters passed to the PHS (including the salt, password, and output lengths); a user identification string; a domain name toward which the user is authenticating him/herself (useful in remote authentication scenarios), among others. Anyhow, using a non-empty *basil* is made optional in the algorithm because in many real-world scenarios such collisions are not considered a serious problem, since the salts used in any given system are usually of fixed length and, even if they are not, they are long and random enough to avoid collisions by themselves.

Once the internal state of the sponge is initialized, its reduced squeezing operation $H.squeeze_\rho$ is called for initializing the first two rows of the memory matrix, $M[0]$ and $M[1]$, without any further input (line 5). Here, "reduced" means that the duplexing may actually be done with a reduced-round version of f, denoted f_ρ for indicating that ρ rounds are executed rather than the regular number of rounds ρ_{max}. This approach accelerates the duplexing operation and, thus, allows more memory positions to be covered in the same amount of time that with the application of a full-round f. Using reduced-round primitives in the core of cryptographic constructions is not unheard in the literature, as it is the main idea behind the ALRED family of message authentication algorithms [34,35,36,37]. As further discussed in Section 4.2, even though the requirements in the context of password hashing are different, this strategy does not decrease the security of the scheme as long as f_ρ is non-cyclic and highly non-linear, which should be the case for the vast majority of secure hash functions.

After $M[0]$ and $M[1]$ are initialized, the sponge's reduced duplexing operation $H.duplexing_\rho$ is then repeatedly called over the XOR of two rows (line 9): $M[prev]$, the one lastly initialized; and $M[row^*]$, a row from the window comprising every row initialized prior to $M[prev]$, taken in the reverse order of their initialization, the said window doubling in size whenever all of its rows are visited (i.e., right after $M[row^*] = M[0]$). More formally, in any iteration $i \geqslant 2$ of the Setup phase, the sponge's input is $M[prev] = M[i-1]$ XORed with $M[row^*] = M[2^{1+\lfloor \lg(i-1) \rfloor} - i]$. Therefore, these two rows must be available in memory before the algorithm can proceed. A similar effect could be achieved if the sponge's input was the concatenation of $M[prev]$ and $M[row^*]$, but XORing them instead is advantageous because then the duplexing operation involves a single call to the underlying f rather than two. The side-effect of this approach

is that, if $C = 1$, the first b bits of the sponge's state would be canceled with the value of $M[prev]$ fed to the sponge; this undesirable situation is avoided in Lyra2, though, due to the $C \geqslant 2$ restriction.

The corresponding output of the reduced duplexing operation, $rand$, then modifies two rows (lines 10 and 11): $M[row]$, which has not been initialized yet, receives the values of $rand$ directly; meanwhile, the columns of the already initialized row $M[row^*]$ have their values updated after being XORed with $rotW(rand)$, i.e., $rand$ rotated to the left by 1 word (e.g., 64 bits). The goal of this left rotation is to avoid the canceling of $rand$ if $M[row]$ and $M[row^*]$ are ever XORed together, which does not happen during the Setup phase but, as will be seen in what follows, may occur during the Wandering phase. Notice that any rotation would be similarly good for avoiding such situations, but a rotation by one word to any direction can be done basically for free, simply by rearranging words instead of actually executing shifts or rotations. More formally, representing $rand$ as an array of words $rand[0] \ldots rand[b/W - 1]$ (i.e., the first b bits of the outer state, from top to bottom as depicted in Figures 1 and 2), we have $M[row][i] \leftarrow rand[i]$ and $M[row^*][i] \leftarrow rand[(i - 1) \bmod (b/W)]$ $(0 \leqslant i \leqslant b/W - 1)$.

The Setup phase ends when all R rows of the memory matrix are initialized, which also means that some of them have been updated since their initialization.

The Wandering phase The most time-consuming of all phases, the *Wandering Phase* (lines 18 to 29), takes place after the Setup phase is finished, without resetting the sponge's internal state. Similarly to the Setup, the core of the Wandering phase comprises the reduced duplexing of $M[prev] \oplus M[row^*]$ for computing a random-like output $rand$ (line 22), which is then XORed with $M[row]$ (line 23) and with $M[row^*]$ after being rotated (line 24).

One important difference, however, is that the index row^* is not deterministic anymore, but is computed as "$(truncL(rand, W) \oplus prev) \bmod R$" (see line 20). Therefore, row^* depends on the most recently computed value of $rand[0]$ (the first word of the sponge's outer state, from top to bottom according to Figures 1 and 2) and, thus, corresponds to a pseudorandom value $\in [0, R - 1]$ that is only learned after the conclusion of the previous duplexing operation. We note that making "$row^* \leftarrow (truncL(rand, W) \bmod R)$" in line 20 would lead to a similar effect, but the additional "$\oplus\, prev$" avoids the situation in which $row^* \equiv prev$, at least in the case where R is a power of 2 (which is recommended because, then, the mod operation can be implemented with a simple logical AND). This is interesting because, when this happens, the algorithm ends up duplexing a string of zeros in its line 22 instead of enforcing the availability of both $M[row^*]$ and $M[prev]$ at that point of the execution. Although there is a low probability of such occurrences even without this trick (on the order of $1/R$), the additional $\oplus$ avoids this situation better and at a very low cost.

Another distinct aspect of the Wandering phase refers to how the row index is handled: instead of going from 0 to $R - 1$ as in the Setup phase, the order is the exact opposite when the Time Loop (lines 18–29) is executed for the first

time, and then is reversed once again for each iteration τ of this loop (as the *dir* variable is updated in line 28). The *prev* index, on the other hand, still corresponds to the previous value of *row*; the sole exception is during the first iteration of the Visitation Loop (lines 19–27), in which $prev = 0$ (when τ is odd) or $prev = R - 1$ (when τ is even).

The Wrap-up phase Finally, after $(T \cdot R)$ rows are iteratively duplexed in the Wandering phase, R rows per iteration of the *Time Loop*, the algorithm enters the *Wrap-up Phase*. This phase consists of a full-round absorbing operation (line 31) of a single cell of the memory matrix, followed by a full-round squeezing operation (line 32) for generating k bits, once again without resetting sponge's internal state to zeros. The goal of this final absorb operation is simply to ensure that the squeezing of the key bitstring will only start after the application of one full-round f to the sponge's state — notice that, as shown in Figure 1, the squeezing phase starts with b bits being output rather than passing by f, and at this point in Lyra2 the state was only updated by the reduced-round f. As a result, this last stage employs only the regular operations of the underlying sponge, building on its security to ensure that the whole process is both non-invertible and of sequential nature.

4.2   Strictly sequential design

Like with PBKDF2 and other existing PHS, Lyra2's design is strictly sequential, as the sponge's internal state is iteratively updated during its operation. Specifically, and without loss of generality, assume that the sponge's state before duplexing input $c_i = M[prev][col + i] \oplus M[row^*][col + i]$ is s_i; then, after c_i is processed, the updated state becomes $s_{i+1} = f_\rho(s_i \oplus c_i)$ and the sponge outputs $rand_i$, the first b bits of s_{i+1}. Now, suppose the attacker wants to parallelize the duplexing of multiple columns in lines 8–12 (Setup phase) or in lines 21–25 (Wandering phase), obtaining $\{rand_0, rand_1, rand_2\}$ faster than sequentially computing $rand_0 = f_\rho(s_0 \oplus c_0)$, $rand_1 = f_\rho(s_1 \oplus c_1)$, and then $rand_2 = f_\rho(s_2 \oplus c_2)$.

If the sponge's transformation f was affine, the above task would be quite easy. For example, if f_ρ was the identity function, the attacker could use two processing cores to compute $rand_0 = s_0 \oplus c_0$, $x = c_1 \oplus c_2$ in parallel and then, in a second step, make $rand_1 = rand_0 \oplus c_1$, $rand_2 = rand_0 \oplus x$ also in parallel. With dedicated hardware and adequate wiring, this could be done even faster, in a single step. However, for a highly non-linear transformation f_ρ, it should be hard to decompose two iterative duplexing operations $f_\rho(f_\rho(s_0 \oplus c_0) \oplus c_1)$ into an efficient parallelizable form, let alone several applications of f_ρ. It is interesting to notice that, if f_ρ has some obvious cyclic behavior, always resetting the sponge to a known state s after v cells are visited, then the attacker could easily parallelize the visitation of c_i and c_{i+v}. Nonetheless, any reasonably secure f_ρ is expected to prevent such cyclic behavior by design, since otherwise this property could be easily explored for finding internal collisions against the full f

itself. In summary, even though an attacker may be able to parallelize internal parts of f_ρ, the stateful nature of Lyra2 creates several "serial bottlenecks" that prevent duplexing operations from being executed in parallel.

Assuming that the above-mentioned structural attacks are unfeasible, parallelization can still be achieved in a "brute-force" manner. Namely, the attacker could create two different sponge instances, I_0 and I_1, and try to initialize their internal states to s_0 and s_1, respectively. If s_0 is known, all the attacker needs to do is compute s_1 faster than actually duplexing c_0 with I_0. For example, the attacker could rely on a large table mapping states and input blocks to the resulting states, and then use the table entry $(s_0, c_0) \mapsto s_1$. For any reasonable cryptographic sponge, however, the state and block sizes are expected to be quite large (e.g., 512 or 1,024 bits), meaning that the amount of memory required for building a complete map makes this approach unpractical.

Alternatively, the attacker could simply initialize several I_1 instances with guessed values of s_1, and use them to duplex c_1 in parallel. Then, when I_0 finishes running and the correct value of s_1 is inevitably determined, the attacker could compare it to the guessed values, keeping only the result obtained with the correct instantiation. At first sight, it might seem that a reduced-round f facilitates this task, since the consecutive states s_0 and s_1 may share some bits or relationships between bits, thus reducing the number of possibilities that need to be included among the guessed states. Even if that is the case, however, any transformation f is expected to have a complex relationship between the input and output of every single round and, to speed-up the duplexing operation, the attacker needs to explore such relationship *faster* than actually processing ρ rounds of f. Otherwise, the process of determining the target guessing space will actually be slower than simply processing cells sequentially. Furthermore, to guess the state that will be reached after v cells are visited, the attacker would have to explore relationships between roughly $v \cdot \rho$ rounds of f faster than merely running $v \cdot \rho$ rounds of f_ρ. Hence, even in the (unlikely) case that guessing two consecutive states can be made faster than running ρ of f, this strategy scales poorly since any existing relationship between bits should be diluted as $v \cdot \rho$ approaches ρ_{max}.

An analogous reasoning applies to the Filling / Visitation Loop, as well as to the Time Loop. The difference for the former is that, to parallelize the duplexing of two rows from consecutive iterations, r_i and r_{i+1}, the attacker needs to determine the sponge's internal state s_{i+1} that will result from duplexing r_i without actually performing the $C \cdot \rho$ rounds of f involved in this operation. For the latter, the state to be determined would be that resulting from the duplexing of several rows, which involves $C \cdot R \cdot \rho$ rounds of f.

Therefore, even if highly parallelizable hardware is available to attackers, it is unlikely that they will be able to take full advantage of this parallelism potential for speeding up the operation of any given instance of Lyra2.

4.3   Configuring memory usage and processing time

The total amount of memory occupied by Lyra2's memory matrix is $m = b \cdot R \cdot C$ bits. The value of b corresponds to the underlying sponge function's bitrate. With this choice of b, there is no need to pad the incoming blocks as they are processed by the duplex construction, which leads to a simpler and potentially faster implementation. The R and C parameters, on the other hand, can be defined by the user, thus allowing the configuration of the amount of memory required during the algorithm's operation.

Ignoring ancillary operations, the processing cost of Lyra2 is basically determined by the number of calls to the sponge's underlying f function. Its approximate total cost is, thus: $\lceil (|pwd| + |salt| + |basil|)/b \rceil + R \cdot C \cdot \rho/\rho_{max}$ calls in the Setup phase, plus $T \cdot R \cdot C \cdot \rho/\rho_{max}$ in the Wandering phase, plus $\lceil k/b \rceil$ in the Wrap-up phase, leading roughly to $(T + 1) \cdot R \cdot C \cdot \rho/\rho_{max}$ calls to f for small lengths of pwd, $salt$ and k. Therefore, while the amount of memory used by the algorithm imposes a lower bound on its total running time, the latter can be linearly increased without affecting the former by choosing a suitable T parameter. This allows users to explore the most abundant resource in a (legitimate) platform with unbalanced availability of memory and processing power. This design also allows Lyra2 to use more memory than scrypt for a similar processing time: while scrypt employs a full-round hash for processing each of its elements, Lyra2 employs a reduced-round, faster operation for the same task.

4.4   On the underlying sponge

Even though Lyra2 is compatible with any hash functions from the sponge family, the newly approved SHA-3, Keccak [19], does not seem to be the best alternative for this purpose. This happens because Keccak excels in hardware rather than in software performance [38]. Hence, for the specific application of password hashing, it gives more advantage to attackers using custom hardware than to legitimate users running a software implementation.

Our recommendation, thus, is toward using a secure software-oriented algorithm with low parallelism as the sponge's f transformation. One example is Blake2b [39], a slightly tweaked version of Blake [40]. Blake itself displays a security level similar to that of Keccak [41], and its compression function has been shown to be a good permutation [42,43] and to have a strong diffusion capability [40], while Blake2b retains most of these security properties [44].

4.5   Practical considerations

Lyra2 displays a quite simple structure, building as much as possible on the intrinsic properties of sponge functions operating on a fully stateful mode. Indeed, the whole algorithm is composed basically of loop controlling and variable initialization statements, while the data processing itself is done by the underlying hash function H. Therefore, we expect the algorithm to be easily implementable in software, especially if a sponge function is already available.

The adoption of sponges as underlying primitive also gives Lyra2 tremendous flexibility. For example, since the user's input (line 4 of Algorithm 1) is processed by an absorb operation, the length and contents of such input can be easily chosen by the user, as previously discussed. Likewise, the algorithm's output is computed using the sponge's squeezing operation, allowing any number of bits to be securely generated without the need of using another primitive (e.g., PBKDF2, as done in scrypt).

Another feature of Lyra2 is that its memory matrix was designed to allow legitimate users to take advantage of memory hierarchy features, such as caching and prefetching. As observed in [5], such mechanisms usually make access to consecutive memory locations in real-world machines much faster than accesses to random positions, even for memory chips classified as "random access". As a result, a memory matrix having a small R is likely to be visited faster than a matrix having a small C, even for identical values of $R·C$. Therefore, by choosing adequate R and C values, Lyra2 can be optimized for running faster in the target (legitimate) platform while still imposing penalties to attackers under different memory-accessing conditions. For example, by matching $b · C$ to approximately the size of the target platform's cache lines, memory latency can be significantly reduced, allowing T to be raised without impacting the algorithm's performance in that specific platform.

Besides performance, making $C \geqslant \rho_{max}$ is also recommended for security reasons: as discussed in Section 4.2, this parametrization ensures that the sponge's internal state is scrambled with (at least) the full strength of the underlying hash function after the execution of the Columns Loop (both in the Setup and Wandering phases). The task of guessing the sponge's state after the conclusion of any iteration of a Columns Loop without actually executing it becomes, thus, much harder. After all, assuming the underlying sponge can be modeled as a random oracle, its internal state should be indistinguishable from a random bitstring.

One final practical concern taken into account in the design of Lyra2 refers to how long the original password provided by the user needs to remain in memory. Specifically, the memory position storing *pwd* can be overwritten right after the first absorb operation (line 4 of Algorithm 2). This avoids situations in which a careless implementation ends up leaving *pwd* in the device's volatile memory or, worse, leading to its storage in non-volatile memory due to memory swaps performed during the algorithm's memory-expensive phases. Hence, it meets the general guideline of purging private information from memory as soon as it is not needed anymore, preventing that information's recovery in case of unauthorized access to the device [45,46].

5   Security analysis

Lyra2's design is such that (1) the derived key is non-invertible, due to the initial and final full hashing of *pwd* and *salt*; (2) attackers are unable to parallelize Algorithm 2 using multiple instances of the cryptographic sponge *Hash*, so they

cannot significantly speed up the process of testing a password by means of multiple processing cores; (3) once initialized, the memory matrix needs to remain available during most of the password hashing process, meaning that the optimal operation of Lyra2 requires enough (fast) memory to hold its contents.

For better performance, a legitimate user is likely to store the whole memory matrix in volatile memory, facilitating its access in each of the several iterations of the Wandering and Wrap-up phases. An attacker running multiple instances of Lyra2, on the other hand, may decide not to do the same, but to keep a smaller part of the matrix in fast memory aiming to reduce the memory costs per password guess. Even though this alternative approach inevitably lowers the throughput of each individual instance of Lyra2, the goal with this strategy is to allow more guesses to be independently tested in parallel, thus raising the overall throughput of the process. There are basically two methods for accomplishing this. The first is to trade memory for processing time, i.e., storing only the sponge's internal state after each row is processed and discarding (parts of) the matrix; then, the attacker can recompute the discarded information from scratch, when (and only when) it becomes necessary; we call this a *Low-Memory attack*. The second it to use low-cost (and, thus, slower) storage devices, such as magnetic hard disks, which we call a *Slow-Memory attack*.

In what follows, we discuss both attack venues and evaluate their relative costs, showing the drawbacks of such alternative approaches. Our goal with this discussion is to demonstrate how Lyra2's design discourages attackers from making such memory-processing trade-offs while testing many passwords in parallel. In other words, we show that they are more likely to pay the memory costs as configured by the legitimate user, which in turn limits the attackers' ability to take advantage of highly parallel platforms, such as GPUs and FPGAs, for password cracking.

In addition the above attacks, we also discuss the so-called *Cache-Timing attacks* [47], which employ a spy process collocated to the PHS and, by observing the latter's execution, could be able to recover the user's password without the need of engaging in an exhaustive search.

5.1   Low-Memory attacks

Before we discuss low-memory attacks against Lyra2, it is instructive to consider how such attacks can be perpetrated against scrypt's *ROMix* structure (see Algorithm 1), since its sequential memory hard design is mainly intended to provide protection against this particular attack venue. Specifically, as a direct consequence of scrypt's memory hard design, we can formulate Theorem 1 below:

Theorem 1. *Whilst the memory and processing costs of scrypt are both $\mathcal{O}(R)$ for a system parameter R, one can achieve a memory cost of $\mathcal{O}(1)$ (i.e., a memory-free attack) by raising the processing cost to $\mathcal{O}(R^2)$.*

Proof. The attacker runs the loop for initializing the memory array M (lines 9 to 11), which we call $ROMix_{ini}$. Instead of storing the values of $M[i]$, however,

the attacker keeps only the value of the internal variable X. Then, whenever an element $M[j]$ of M should be read (line 14 of Algorithm 1), the attacker simply runs $ROMix_{ini}$ for j iterations, determining the value of $M[j]$ and updating X. Ignoring ancillary operations, the average cost of such attack is $R + (R \cdot R)/2$ iterative applications of *BlockMix* and the storage of a single b-long variable (X), where R is scrypt's cost parameter. □

In comparison, an attacker trying to use a similar low-memory attack against Lyra2 would run into additional challenges. First, during the Setup phase, it is not enough to keep only one row in memory for computing the next one, as each row requires a pair of previously computed rows for its computation. For example, after using $M[0]$ and $M[1]$ for computing $M[2]$, $M[0]$ and $M[1]$ will still be used, respectively, in the computation of $M[4]$ and $M[3]$, meaning that they should not be discarded or they will have to be recomputed. Even worse: since $M[0]$ is modified when initializing $M[2]$, the value to be actually employed when computing $M[4]$ cannot be obtained directly from the password only. Instead, recomputing the updated value of $M[0]$ requires (a) running the Setup phase until the point it was last modified (i.e., when $M[2]$ was computed) or (b) using the value of $M[2]$ if it is already in memory, taking into account the value of *rand* that modified $M[0]$ can be obtained from $M[2]$. This creates a complex net of dependencies that grow in size as the algorithm's execution advances and more rows are modified, leading to several recursive calls. This effect is even more accentuate in the Wandering phase, due to an extra complicating factor: each duplexing operation involves a random-like (password-dependent)row index that cannot be determined before the end of the previous duplexing. Therefore, the choice of which rows to keep in memory and which rows to discard is merely speculative, and cannot be easily optimized for all password guesses.

Providing a tight bound on the complexity of such low-memory attacks against Lyra2 is, thus, an involved task, especially considering its non-deterministic nature. Nevertheless, aiming to give some insight on how an attacker could (but is unlikely to want to) explore such time-memory trade-offs, in what follows we consider some slightly simplified attack scenarios. We emphasize, however, that these scenarios are not meant to be exhaustive, since the goal of analyzing them is only to show the approximate (sometimes asymptotic) impact of possible memory usage reductions over the algorithm's processing cost. Formally proving the resistance of Lyra2 against time-memory trade-offs (e.g., using the theory of Pebble Games [48,49,50] as done in [47,51]) would be even better, but doing so, possibly building on the discussion hereby presented, remains as a matter for future work.

Preliminaries For conciseness, along the discussion we denote by CL the Columns Loop of the Setup phase (lines 8–12 of Algorithm 2) and of the Wandering phase (lines 21–25). In this manner, ignoring the cost of XORing, reads/writes and other ancillary operations, the cost of CL corresponds approximately to $C \cdot \rho/\rho_{max}$ executions of f, denoted simply by σ.
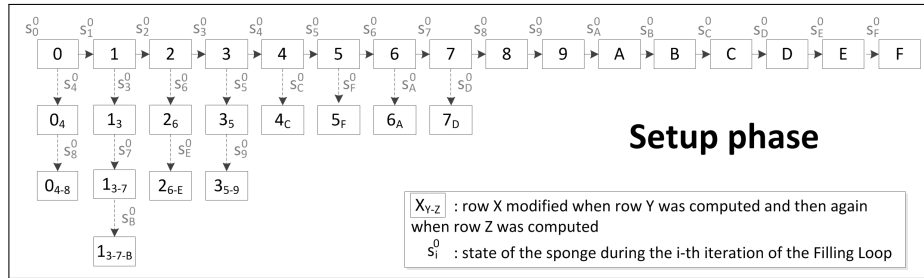
s_0^0 s_1^0 s_2^0 s_3^0 s_4^0 s_5^0 s_6^0 s_7^0 s_8^0 s_9^0 s_A^0 s_B^0 s_C^0 s_D^0 s_E^0 s_F^0

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |

s_4^0 s_3^0 s_6^0 s_5^0 s_C^0 s_F^0 s_A^0 s_D^0

| 0_4 | 1_3 | 2_6 | 3_5 | 4_C | 5_F | 6_A | 7_D |

Setup phase

s_8^0 s_7^0 s_E^0 s_9^0

| $0_{4\text{-}8}$ | $1_{3\text{-}7}$ | $2_{6\text{-}E}$ | $3_{5\text{-}9}$ |

s_B^0

| $1_{3\text{-}7\text{-}B}$ |

$X_{Y\text{-}Z}$: row X modified when row Y was computed and then again when row Z was computed
s_i^0 : state of the sponge during the i-th iteration of the Filling Loop

Fig. 3. The Setup phase.

We also denote by s_i^0 the state of the sponge during the i-th iteration of the Filling Loop, during the Setup phase, before the corresponding rows are effectively processed (i.e., the state in line 7 of Algorithm 2). Similarly, for the Wandering phase, we denote by s_i^τ the state of the sponge during the i-th iteration of the Visitation Loop and the τ-th iteration of the Time Loop, before the corresponding rows are effectively processed (i.e., the state in line 19 of Algorithm 2). Aiming to keep track of modifications made on those rows, we recursively use the subscript notation $M[X_{Y-Z}]$ to denote a row X modified when paired with row Y and then again when paired with row Z. Finally, we write V_1^τ and V_2^τ to denote, respectively, the first and the second half of the Visitation Loop during the τ-th iteration of the Time Loop.

The Setup phase. We start our discussion analyzing only the Setup phase. Aiming to give a more concrete view of its execution, along the discussion we use as example the scenario with 16 rows depicted in Figure 3, which shows the corresponding visitation order of such rows and also their modifications due to these visitations.

Storing only what is needed. Suppose that the attacker does not want to store all rows of the matrix during the algorithm's execution. One interesting approach for doing so is to store only what will be required in future iterations of the Filling Loop, discarding rows that will not be used anymore. Since the algorithm is purely deterministic, doing so is quite easy and, as long as the proper rows are kept in memory, incurs no processing penalty. This approach is illustrated in Figure 4 for our example scenario.

As shown in this figure, this simple strategy allows the execution of the Setup phase with a memory usage of approximately $R/2 + 1$ rows, since each half of the Setup phase requires all rows from the previous half and one extra row (the last one computed) to proceed. More precisely, $R/2 + 1$ corresponds to the peak memory utilization reached around the middle of the Setup phase, since (1) at the beginning of the phase part of the memory matrix has not been initialized yet and (2) rows computed near the end of the phase are only paired with the next row and, thus, can be discarded right after that. Even with this reduced
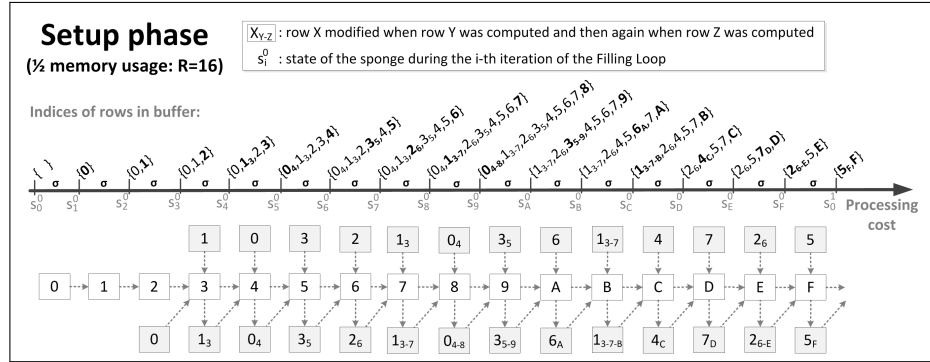
Setup phase
(½ memory usage: R=16)

X_{Y-Z} : row X modified when row Y was computed and then again when row Z was computed
s_i^0 : state of the sponge during the i-th iteration of the Filling Loop

Indices of rows in buffer:

{ } {0} {0,1} {0,1,2} {0,1,2,3} {0,1,2,3,4} {0,1,2,3,4,5} {0,1,2,3,4,5,6} {0,1,2,3,4,5,6,7} {0,1,2,3,4,5,6,7,8} {0,1,2,3,4,5,6,7,9} {1,2,3,4,5,6,7,A} {1,2,3,4,5,6,7,8} {2,4,5,7,C} {2,5,7,D} {2,5,E} {5,F}

s_0^0 s_1^0 s_2^0 s_3^0 s_4^0 s_5^0 s_6^0 s_7^0 s_8^0 s_9^0 s_A^0 s_B^0 s_C^0 s_D^0 s_E^0 s_F^0 s_0^0 **Processing cost**

| 1 | 0 | 3 | 2 | 1_3 | 0_4 | 3_5 | 6 | 1_{3-7} | 4 | 7 | 2_6 | 5 |

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |

| 0 | 1_3 | 0_4 | 3_5 | 2_6 | 1_{3-7} | 0_{4-8} | 3_{5-9} | 6_A | 1_{3-7-8} | 4_C | 7_D | 2_{6-E} | 5_F |

Fig. 4. Attacking the Setup phase: storing 1/2 of all rows. The most recently computed row in each iteration is marked as **r**.

memory usage, the processing cost of this phase remains at $R \cdot \sigma$, just as if all rows were kept in memory.

This attack can, thus, be summarized by the following lemma:

Lemma 1. *Consider that Lyra2 operates with parameters T, R and C. Whilst the regular algorithm's memory and processing costs of its Setup phase are, respectively, $R \cdot C \cdot b$ bits and $R \cdot \sigma$, it is possible to run this phase with a maximum memory cost of $(R \cdot C \cdot b)/2$ bits while keeping its total processing cost to $R \cdot \sigma$.*

Proof. The costs involved in the regular operation of Lyra2 are discussed in Section 4.3, while the mentioned memory-processing trade-off can be achieved with the attack described above. □

Storing less than what is needed. If the attacker consider that storing $R/2$ rows is too much, he/she may decide to discard additional rows, recomputing them from scratch only when they are needed. In that case, one appealing approach is to discard the rows that will take longer to be used. The reason is that this strategy allows the Setup phase to proceed smoothly for as long as possible and, thus, the arising missing rows will end up being those nearer the start of the matrix, which, in principle, are less expensive to compute.

The suggested approach is illustrated in Figure 5. As shown in this figure, at any moment we keep in memory only $R/4 + 1$ rows of the memory matrix. This allows the Setup phase to run without any recomputation until we need to compute row $M[D]$, at which moment $M[3_5]$ is required. Since $M[5]$ is available in memory at this point, we only need to recompute $M[3]$ from scratch and then use the fact that $M[3_5] = M[3] \oplus rotW(M[5])$ to obtain its value. However, if we want to keep $M[6]$ and $M[7]$ in memory for using an analogous trick when computing $M[2_6]$ and $M[1_{3-7}]$, we have to compute $M[3]$ itself while maintaining basically one single row in memory in addition to the four rows already kept by the main processing thread (thus still respecting the $R/4 + 1$ memory usage). As a result, computing $M[3]$ takes a total of 7σ rather than only 4σ as expected if
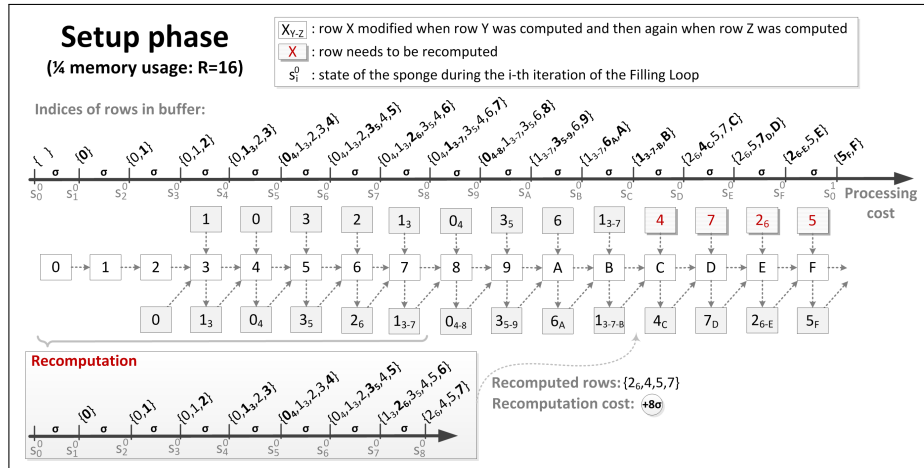
Fig. 5. Attacking the Setup phase: storing 1/4 of all rows. The most recently computed row in each iteration is marked as **r**.

$M[3]$ could be computed with an unbound amount of memory, because $M[0]$ and $M[1]$ have to be computed twice each. After that, as shown in bottom-right side of Figure 5, $M[2]$ and $M[1_3]$ can both be obtained with the storage of five rows if we (1) recompute $M[1]$ from scratch, with a cost of 2σ, XORing it to the value of $M[3]$ already available in memory, and (2) continue with the recomputation of $M[2]$ by using the space originally occupied by row $M[1]$, with an extra cost of 2σ. All in all, the cost of the attack in this example scenario almost doubles with this reduction of approximately half in memory usage.

Even though for the time being we are unable to provide a tight bound to the cost of recursively reducing memory usage by half, our simulations seem to indicate that the corresponding processing cost of such attacks approximately doubles with each reduction. Indeed, as shown in Figure 6, when reducing the memory cost to $\approx R/8 + 1$, the need of recomputations appears at the end of the first half of the memory matrix. The result is that the cost of computing this first half will already be approximately $R \cdot \sigma$, assuming that s_4^0 is stored by the attacker so it can be used when recomputing $M[0_{2-4}]$ from $M[0_2]$ and $M[3]$. After that, when computing the second half of the memory matrix, we are only able to keep in memory $R/8$ out of the $R/2$ values of $M[row^*]$ required, leading to the need of recomputing the remaining $3R/8$ from scratch, up to $R/8$ at a time and using no more than $R/8$ per recomputation. The consequence is that computing this second half of the memory matrix takes three times the cost of computing the first half if two extra sponge states (s_6^0 and s_7^0) are stored, leading to a total cost of approximately $4R \cdot \sigma$.

From the above observations, supposing that one can recursively use this strategy for reducing the algorithm's memory usage near to 1 row, the resulting
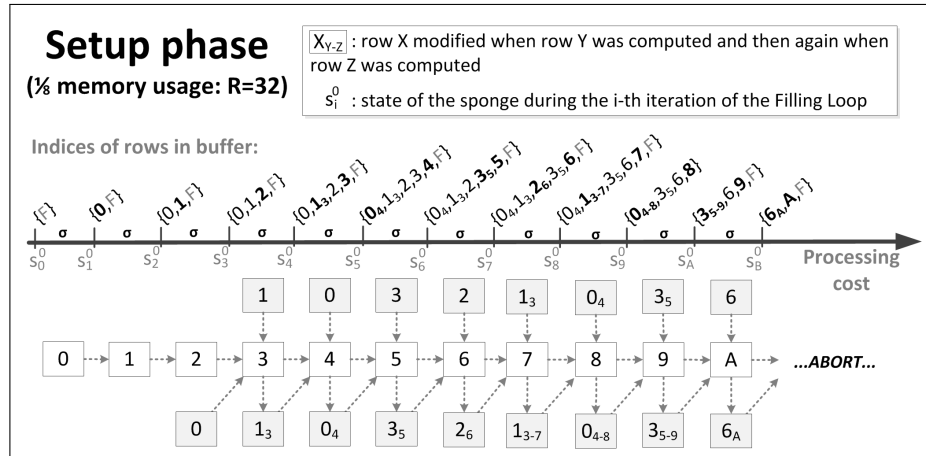
22

Fig. 6. Attacking the Setup phase: storing 1/8 of all rows. The most recently computed row in each iteration is marked as **r**.

processing cost of the Setup phase is expected to become approximately $(R^2/2) \cdot \sigma$, leading to the formulation of the following conjecture:

Conjecture 1. Consider that Lyra2 operates with parameters T, R and C. Whilst the regular memory and processing costs of its Setup phase's are, respectively, $R \cdot C \cdot b$ bits and $R \cdot \sigma$, one may be able to achieve a memory cost of $\mathcal{O}(1)$ bits by raising the processing cost to approximately $(R^2/2) \cdot \sigma$.

Adding the Wandering phase with $T = 1$. For analyzing the Wandering phase, it is useful to consider an "average", deterministic scenario, such as the execution of a single Time Loop depicted in Figure 7. This is a slightly simplified scenario, in which we ignore the fact that some rows are actually modified during the Setup phase. This simplification facilitates the analysis and, albeit unrealistic, it would actually benefit the attacker: whenever a row $M[i]$ needs to be recomputed from scratch, the processing cost for doing so can be as low as
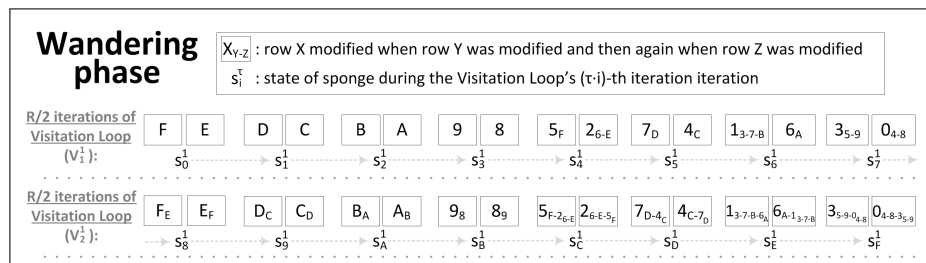

Fig. 7. An example of the Wandering phase's execution.

$i \cdot \sigma$, while recomputing a modified row $M[i_j]$ would in principle have a cost of at least $j \cdot \sigma$, since we always have $(j > i)$ during the Setup phase. Therefore, a real attack is expected to be at least as expensive as the attacks performed in this simplified scenario.

In addition to the above simplification, we consider that all rows are modified only once during the first half of the Visitation Loop, i.e., during V_1^1 every row from $M[F]$ to $M[8]$ is paired with a row from $M[7]$ to $M[0]$ when receiving the sponge's output. We argue that this is once again beneficial for the attacker, since any row required during this process can be obtained simply by running the Setup phase once again, instead of involving recomputations of the Wandering phase itself. We then apply the same principle to V_2^1, modifying each row only once more in a different (arbitrary) pseudorandom order.

The first half of the Visitation Loop with low memory usage. Figure 8 depicts in more detail the execution of V_1^1 in our example scenario, showing that any of its iterations involves three row indices: *prev*, *row** and *row*. Rows $M[prev]$ and $M[row^*]$ are needed for feeding the sponge (line 22 of Algorithm 2) and, thus, must be in memory for updating its internal state. In comparison, $M[row]$ corresponds to a row that only receives the sponge's output (line 23) and, thus, is not strictly necessary for updating the sponge's state. However, since $M[row]$ will certainly be used as input in the very next iteration (namely, as the new $M[prev]$), it makes more sense to have it in memory too for receiving the sponge's output. On the other hand, the other row receiving the sponge's output, $M[row^*]$, will only be useful in the next iteration if picked once again as the new $M[row^*]$ or $M[row]$, which happens with probability $2/R$. In the scenario depicted in Figure 7, for example, $M[row^*] = M[7_E]$ obtained as output of the Visitation Loop's iteration 0 is indeed not useful in the next iteration, which takes $M[R-2] = M[E]$ (deterministically picked) and $M[7]$ (randomly picked) instead.

From the above observations, its is reasonable to consider that the attacker always has $M[prev]$ in memory at the beginning of any iteration of the Visitation Loop, since he/she can simply keep the output of the previous iteration
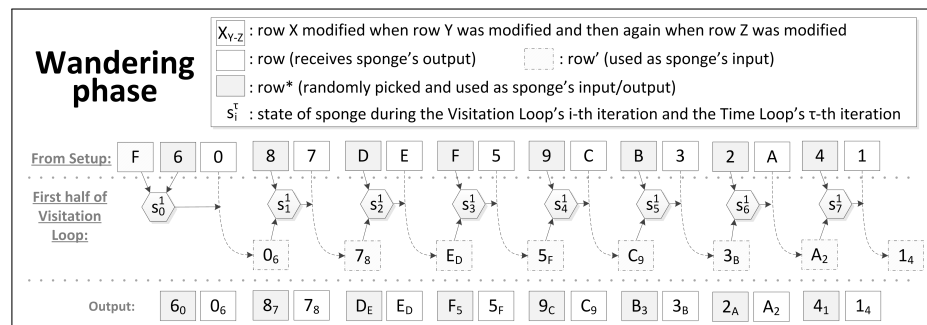

Fig. 8. Wandering phase's execution in detail: first half of Visitation Loop.

— or, for the very first iteration of this loop, have either $M[0]$ (when τ is odd) or $M[R-1]$ (when τ is even) in memory. $M[row]$ and $M[row^*]$, nevertheless, may have been discarded in previous iterations for keeping the algorithm's memory usage at a low level. In that case, those rows can be recomputed from scratch by running $\max(row^*, row)$ iterations of the Setup phase once again while storing $M[\min(row^*, row)]$ even after this row is not needed by the Setup phase itself. Going back to iteration 1 of the Visitation Loop in our example scenario, $M[row] = M[E]$ and $M[row^*] = M[7]$ can both be recomputed by running $\max(row^*, row) = E$ iterations the Setup phase while not discarding $M[\min(row^*, row)] = M[7]$ in the process.

Denoting the processing cost of j iterations of the Setup phase by $\lambda_s(j)$ and its memory cost by $\mu_s(j)$, the total cost for step i of V_1^1 ($0 \leqslant i < R/2$) whenever both $M[row]$ and $M[row^*]$ need to be recomputed is approximately $\lambda_s(R-i)$ executions of CL and the storage of $S_m(R-i) + 1$ rows. If the Setup phase is run with a very low amount of memory, we have $\lambda_s(R-i) \approx (R-i)^2/2$ for $S_m(R-i) \approx 1$. Therefore, since V_1^1 involves $R/2$ duplexing operations, its approximate total cost is $(R/2) \cdot \lambda_s(R - R/4) = (3R/4)^3/3$ executions of CL and the storage of up to two extra rows.

The first half of the Visitation Loop with the "store ahead" strategy. The processing costs of V_1^1 can be slightly reduced using the fact that $M[R-i-1]$ is known to be required during iteration i. Therefore, when running the Setup phase for iteration i, the attacker can very well keep in memory not only $M[R-i-1]$ but also $M[R-i-2]$, which will certainly be used in iteration $(i+1)$.

Using this "store-ahead" strategy, the memory cost of recomputing the Setup phase does not raise significantly, since $M[R-i-2]$ is still in memory right before $M[R-i-1]$ is computed and, at this point, most of the other rows may have already been discarded without impacting the Setup phase's processing cost. On the other hand, since there is no way to determine which $M[row^*]$ that will be used together with $M[R-i-2]$ before the end of iteration i, the corresponding $M[row^*]$ is likely to be discarded and will have to be recomputed during iteration $(i+1)$. For example, in the scenario depicted in Figure 7, the attacker could keep not only $M[F]$ and $M[5]$ but also $M[E]$ during the first iteration of the Visitation Loop. $M[E]$ could then be used in the second iteration, during which only $M[7]$ would have to be recomputed.

As a result of applying this store-ahead strategy for every pair of iterations i and $i+1$, the processing cost of iteration i remains basically the same, while the average cost of iteration $i+1$ becomes $\lambda_s(R/2)\sigma$ for recomputing any $M[row^*] \in [0, R-1]$. Hence, using the same low-memory approach as before, the total cost of V_1^1 becomes approximately $(R/4)(R - R/4)^2\sigma/2 + (R/4)(R/2)^2\sigma/2 \approx (3R/4)^3\sigma/4$ in total, which is only slightly faster than the $(3R/4)^3\sigma/3$ obtained without this strategy.

In addition, this store-ahead approach does not scale well for storing many rows. For example, in the extreme case of storing all rows known to be required during V_1^1, the attacker would have to store rows $M[R/2]$ to $M[R-1]$. Even with this high storage of $R/2$ rows, 50% of the time the randomly picked rows
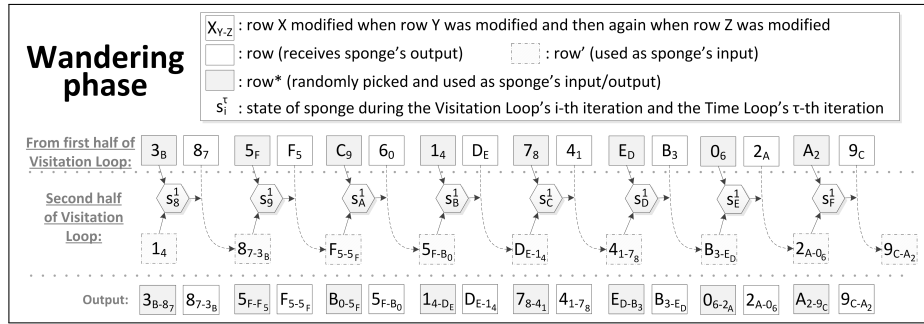
Fig. 9. Wandering phase's execution in detail: second half of Visitation Loop.

$M[row^*]$ would belong to the unavailable interval $M[0]$ to $M[R/2-1]$. Therefore, in V_1^1 the attacker would still have to recompute from scratch $R/4$ rows, with an average processing cost of $(R/4) \cdot \lambda_s(R/4)\sigma$ in total. If such rows are once again recomputed by running the Setup phase with a very low amount of memory, the total processing cost would become $(R/4)(R/4)^2\sigma/2 = (R/4)^3\sigma/2$, or $1/18$ of the cost of an attack not using this strategy.

The second half of the Visitation Loop. For V_2^1, the situation is different from what happens in V_1^1: since the rows required for any iteration of V_2^1 have been modified during the execution of V_1^1, it does not suffice to run the Setup phase once again to get their values. For example, in the scenario depicted in Figure 7, the rows required for iteration $i = 8$ of the Visitation Loop besides $M[prev] = M[8_3]$ are $M[7_E]$ and $M[A_2]$ (see Figure 9 for details). If those latter rows have not been kept in memory, their recomputation from scratch requires on average (see Figure 8): four rows whose values can be obtained by running the Setup phase (in our example, $M[7]$ and $M[E]$, $M[A]$ and $M[2]$); and two rows whose values can only be learned by running V_1^1 (values assumed by the $M[prev]$ variable), together with the corresponding sponge states (in our example, $M[F_5]$ and $M[B_0]$, as well as s_1^1 and s_5^1).

Obviously, given the probabilistic nature of the algorithm, the number of rows and states may actually be lower. For example, iteration $i = 9$ of the Visitation Loop requires $M[C_6]$ and $M[6_C]$, and both can be recomputed from scratch using two rows from the Setup phase ($M[C]$ and $M[6]$), as well as a single row from V_1^1 and sponge state ($M[prev] = M[D_4]$ and s_3^1). In addition, during iteration $i = 0$, if the randomly row picked to be XORed with $M[prev] = M[0]$ is $M[row^*] = M[0]$ itself, the attacker does not need to recompute anything, but can simply feed the sponge with a string of zeros. Since both situations only happen with probability $1/R$, though, we ignore such outliers along the discussion.

Whichever the iteration, the four rows required from the Setup phase can be computed by running (part of) it while storing those rows even after they are not necessary for the Setup phase itself, following the previously discussed

"store-ahead" strategy. For example, rows $M[E]$, $M[7]$, $M[A]$ and $M[2]$ can all be obtained with an approximate cost of $\lambda_s(R)\sigma = (R)^2\sigma/2$ and the storage of four rows in addition to the few rows required in any extremely low-memory execution of the Setup phase. This processing cost can be reduced further by aborting the computation after $M[E]$ is computed.

The previously modified rows $M[prev]$ and sponge states, on the other hand, can only be computed if parts of the Wandering phase are executed once again. Namely, suppose that s_0^1 and $M[0] \oplus M[5]$ are both known, either because they were kept in memory after the end of the Setup phase or because they were recomputed by running the whole Setup phase once again. Then, computing state s_i^1 from s_0^1 takes i iterations of the Visitation Loop, and this process already provides the corresponding $M[prev]$ to feed the sponge in this state. In addition, any pair of states (s_i^1, s_j^1) can be recomputed from s_0^1 and corresponding input ($M[0] \oplus M[5]$ in our example) with $\max(i,j)$ iterations of the Visitation Loop. For V_1^1, in which ($i, j \in [0, R/2 - 1]$), we have then an average of $3/4$ iterations, comprising a total of $3(R/2)/4 = 3R/8$ executions of CL for duplexing roughly $3R/4$ rows (i.e., $3R/8$ pairs used as input) whose values can be obtained directly from the Setup phase.

Combining the above observations, we can go back to our example for computing $M[7_E] \oplus M[A_2]$ as required at the start of V_2^1 (namely, during iteration $R/2 = 8$). Specifically, the attacker can do so by: (1) running the Setup phase once, while keeping in memory all rows required between the start of the Wandering phase and the computation of $M[A_2]$ (namely $M[0]$, $M[5]$, $M[F]$, $M[7]$, $M[E]$, $M[4]$, $M[D]$, $M[6]$, $M[C]$, $M[B]$, $M[2]$ and $M[A]$); and then (2) making 6 sequential executions of CL, the first using state s_0^1 and the last using state s_5^1. Alternatively, if the attacker wants to reduce the extra memory cost of this process in approximately half, he/she may execute the Setup phase twice: in the first run, only the set $\{M[0], M[5], M[F], M[7], M[E], M[4], M[D]\}$ is computed and the states s_0^1 to s_2^1 are employed in three sequential iterations of V_1^1; in the second, the attacker computes $\{M[6], M[C], M[0], M[B], M[2], M[A]\}$ and uses states s_3^1 to s_5^1 in three additional iterations of V_1^1. This approach can be extended until the number of additional rows stored in each run of the Setup phase is reduced to approximately two rather than $3R/4$, but then we would have to run roughly $3/4$ of the Setup for $3R/8$ times, once for each pair of rows employed during V_1^1. Whichever the case, the $R/2$-th iteration finishes with a final execution of the Columns Loop using the most current state s_8^1.

We are now in position to estimate the total cost of V_2^1, i.e., for iterations $(R/2)$ to $(R-1)$ of the Visitation Loop. Using the above strategies and assuming that s_0^1 and the input to be fed to the sponge in this state is known, the average cost of any single iteration i becomes approximately: from one (approximately full) execution of the Setup phase, with a processing cost of $\lambda_s(R)\sigma$ and the storage of $\mu_s(R) + 3R/4$ rows, to $3R/8$ partial executions of the Setup, each of which requiring $\lambda_s(3R/4)\sigma$ and the storage of approximately two rows; plus $3R/8 \cdot \sigma$ for computing from s_0^1 (assumed to remain in memory during the whole process) the two rows and corresponding states required during iteration i; plus

1σ using state s_i^1. Therefore, for all $R/2$ iterations of this second half of the Visitation Loop, the total processing cost of an almost memory-free execution becomes $(R/2)(3R/8) \cdot \lambda_s(3R/4)\sigma + R \cdot \sigma/2$, or approximately $(3R^2/16)((R - R/4)^2/2)\sigma = (3R/4)^4\sigma/6$.

Notice that this cost can be reduced by extending the "storage-ahead" strategy also to the second half of the Visitation Loop. For example, during iteration $i = 9$, the attacker knows that $M[6_C]$ will be required, since it is deterministically picked. Therefore, it is just natural to keep $M[6_C]$ in memory when $M[C_6]$ is recomputed during iteration $i = 8$, even if the recomputation is done using an almost memory-free (more computationally intensive) strategy. Nonetheless, since iteration $i = 9$ takes a random row as input (in the example, $M[C_6]$) just like any other iteration, in principle that row cannot be stored ahead because it is unknown to be needed. Hence, as in the case of the first half of the Visitation Loop, the acceleration brought by this approach is not expected to affect the asymptotic cost of the attack.

Adding the Wandering phase with $T > 1$. Given Lyra2's sequential structure, in any iteration τ of the Time Loop attackers can use the previous iteration $\tau - 1$ approximately like the first iteration of the Time Loop used the Setup phase. After all, right after the sponge's state becomes s_0^τ at the beginning of the Time Loop's iteration τ, proceeding with the algorithm's computation requires a random row $M[row^*]$ together with the rows at indices 0 and F (which alternate as $M[prev]$ and $M[row]$, depending on whether τ is even or odd), just like what happens when the sponge reaches the state s_0^1. The remainder of the τ-th execution of the Time Loop proceeds similarly, visiting rows in the reverse order in which they were computed during iteration $\tau - 1$, and combining them with random rows.

Therefore, as $Lyra2$'s execution progresses, it creates an inverted tree-like dependence graph like the one depicted in Figure 10, level $\ell = 0$ corresponding to the Setup phase and each half of the Visitation Loop raising the tree's depth by one. Each level $\ell > 0$ of this tree corresponds, thus, to $R/2$ iterations of the
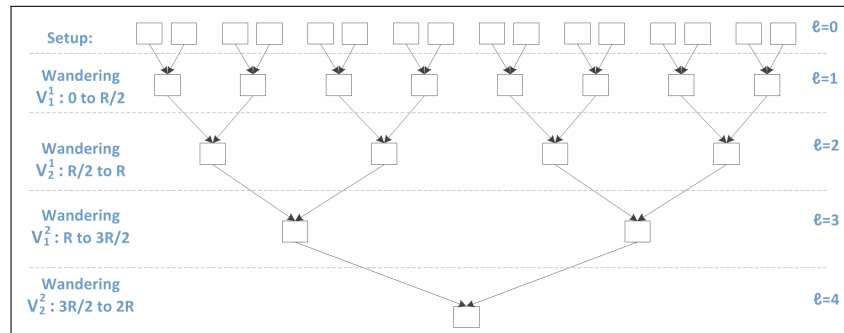

Fig. 10. Tree representing the dependences among rows in Lyra2.

Visitation Loop, each iteration requiring one (partial) execution of the previous level $(\ell - 1)$ for recomputing up to two rows thereby modified but discarded. If, as discussed in Section 5.1, the attacker saves memory by storing only the sponge state at the start of that level and the first input fed to the sponge at that point, the iterations at level ℓ requires $3/4$ iterations of level $(\ell - 1)$. The latter can then be run (1) with a single (approximately full) execution of level $(\ell - 2)$ for recovering at once all $\approx 3R/8$ pairs of rows involved in this partial execution, taking advantage of the fact that the rows' indices are already known; or (2) with the less memory-consuming approach of also executing only $3/4$ iterations of level $(\ell - 2)$ for recovering a single pair of rows at a time, leading to a total of $3R/8$ executions of level $(\ell - 2)$.

From the above discussion, we can estimate the cost of an extremely low-memory attack against any level $\ell \geqslant 2$ of the tree to be approximately $\lambda_\ell = (R/2)(3R/8) \cdot \lambda_{\ell-2}$, with $\lambda_0 \approx (3R/4)^2/2$. Hence, for last iteration of the Wandering phase, the second half of the Visitation Loop alone (i.e., for $\ell = 2T$) is expected to cost approximately $(3R^2/8)^T \cdot (3R/4)^2/2 = (3R/4)^{2T+2}/(2 \cdot 3^T)$, dominating Lyra2's running time. The total storage cost in this highly memory-constrained attack would then be roughly equivalent to $2T$ rows and sponge states (i.e., $2T(w + C \cdot b)$ bits). This can be summarized in the following Conjecture:

Conjecture 2. Consider that Lyra2 operates with parameters T, R and C. Whilst its regular memory and processing costs of are, respectively, $R \cdot C \cdot b$ bits and $R \cdot (T + 1) \cdot \sigma$, one can achieve a memory cost of $2T(w + C \cdot b)$ bits by raising the processing cost to approximately $(3R/4)^{2T+2}/(2 \cdot 3^T) \cdot \sigma$.

A remark on the row visitation order: reverse vs. bit-reversal During Lyra2's Setup phase, rows that have already been initialized ($M[row^*]$) are fed to the sponge and then updated with the latter's output, *rand*, forcing attackers to recompute those rows if they were previously discarded. A similar behavior is also observed in the algorithm's Wandering phase, during which the deterministically picked rows $M[row]$ are taken in the reverse order of the previous iteration of the Time Loop. This raises the natural question of "why reverse?". The main reasons for adopting this strategy is that (1) it allows rows with small indices to be XORed with rows with much higher indices, raising the cost of their recomputation during attacks, while (2) it is still quite simple to implement.

Nonetheless, it is reasonable to discuss why we do not adopt an approach based on a bit-reversal permutation, similarly to what is done in the Catena PHS [47]. After all, this is indeed a promising approach, especially considering the existence of tight proofs related the time-memory trade-offs of pebbling directed acyclic graphs (DAGs) built with this strategy [52]. However, our preliminary analysis shows that, besides being simpler, revisiting rows in a reverse order in the context of Lyra2 is not less effective than using bit-reversal with respect to resistance against low-memory attacks, as discussed in what follows.

During the Setup phase, the reverse visitation is not the only factor in play: there is also the fact that revisited rows are updated, making them depend on
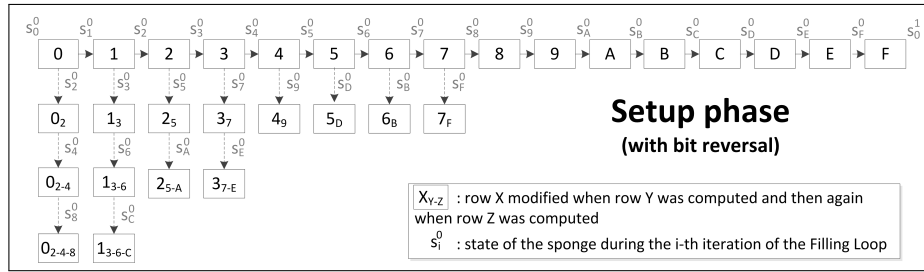
Fig. 11. The Setup phase with bit-reversal.

rows with higher indices and, thus, the cost of recomputing them raises as the Setup phase evolves. This is the main reason why, as discussed in Section 5.1, the processing cost of this phase approximately doubles when the amount of memory used for its execution goes from $R/2$ to $R/4$, and then again when it goes to $R/8$. In comparison, if we adopt a bit-reversal instead of a reverse visitation order in this phase (see Figure 11), the effect does not seem to be much better. Indeed, as shown in Figure 12, the additional processing cost of an attack with memory usage of $R/4$ in this bit-reversal experiment was only slightly higher than what is observed with Lyra2's reverse strategy, going from the 11σ originally obtained (see Figure 5) to 13σ. Conversely, for a total storage of $R/8$ as shown in Figure 13, the cost of the bit-reversal approach dropped to 34σ, a considerable reduction when compared with the 47σ obtained with the simple reverse visitation depicted in Figure 6.
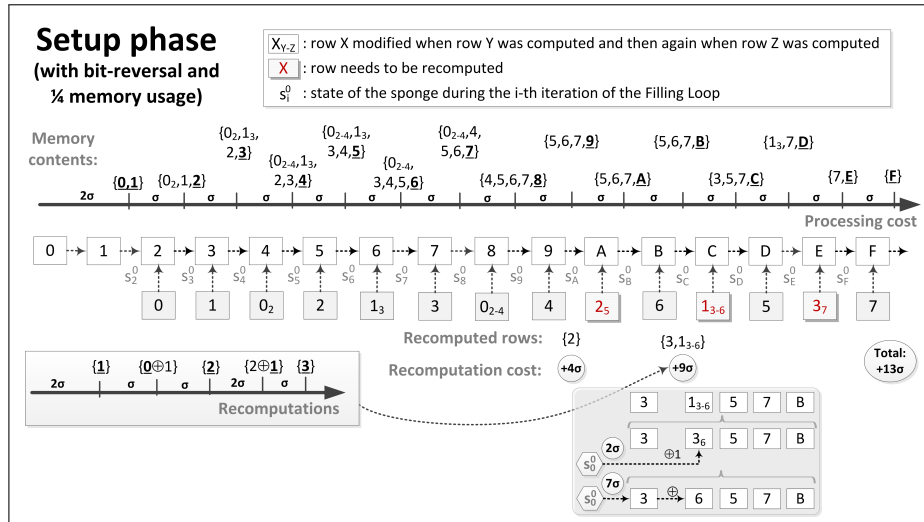

Fig. 12. Attacking the Setup phase with bit-reversal: storing 1/4 of all rows. The most recently computed row in each iteration is marked as **r**.
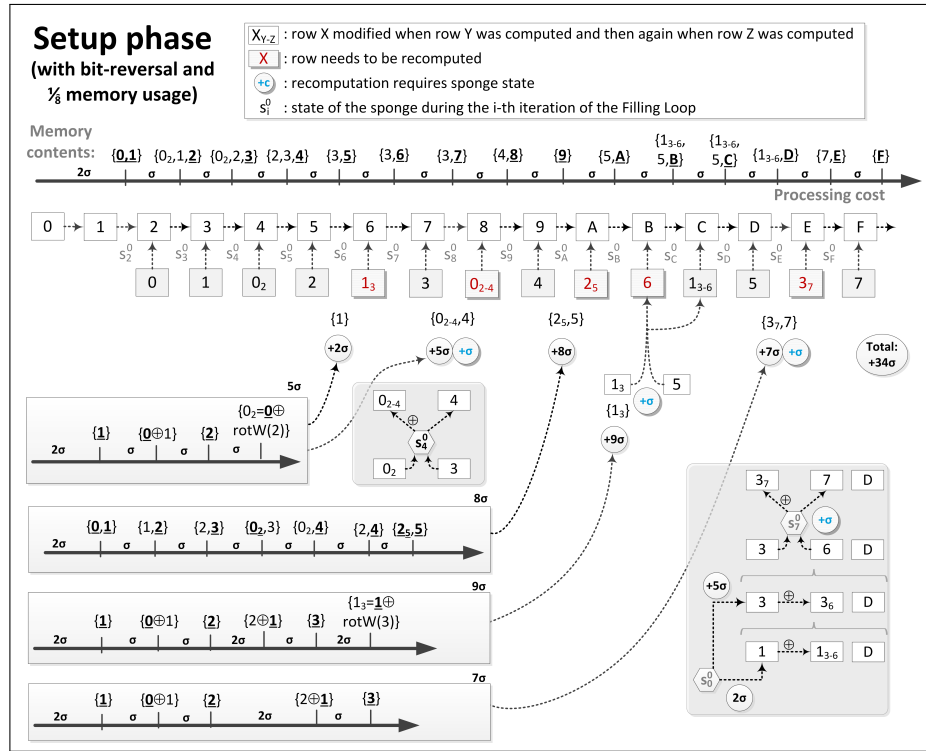
Fig. 13. Attacking the Setup phase with bit-reversal: storing 1/8 of all rows. The most recently computed row in each iteration is marked as **r**.

In the Wandering phase, we do not expect the effect to be much different, since the rows picked in a pseudorandom manner are more likely to be a burden to attackers than those picked in a deterministic manner.

These observations are by no means enough to allow any conclusion on which strategy is better, bit-reversal or simple reverse. Nevertheless, they indicate that, even though further analysis is required, the latter strategy is somewhat comparable to the former. Therefore, following Occam's Razor principle [53], we decided to adopt the simpler, reverse solution in the design of Lyra2.

5.2 Slow-Memory attacks

When compared to low-memory attacks, providing protection against slow-memory attacks is a more involved task. This happens because the attacker acts approximately as a legitimate user during the algorithm's operation, keeping in memory all information required. The main difference resides on the bandwidth and latency provided by the memory device employed, which ultimately impacts the time required for testing each password guess.

Lyra2, similarly to scrypt, explores the properties of low-cost memory devices by visiting memory positions following a pseudorandom pattern. In particular, this strategy increases the latency of intrinsically sequential memory devices, such as hard disks, especially if the attack involves multiple instances simultaneously accessing different memory sections. Furthermore, as discussed in Section 4.5, this pseudorandom pattern combined with a small C parameter may also diminish speedups obtained from mechanisms such as caching and pre-fetching, even when the attacker employs (low-cost) random-access memory chips. We notice that this strategy is particularly harmful against existing GPUs, whose internal structure is usually optimized toward deterministic memory accesses to small portions of memory.

When compared with scrypt, a slight improvement introduced by Lyra2 against such attacks is that the memory positions are not only repeatedly read, but also written. As a result, Lyra2 requires data to be repeatedly moved up and down the memory hierarchy. The overall impact of this feature on the performance of a slow-memory attack depends, however, on the exact system architecture. For example, it is likely to increase traffic on a shared memory bus, while caching mechanisms may require a more complex circuitry/scheduling to cope with the continuous flow of information from/to a slower memory level.

Another appealing aspect about Lyra2's design is the fact that the sponge's output is always XORed with the value of existing rows, preventing the memory positions corresponding to those rows from becoming quickly replaceable. This property is, thus, likely to hinder the attacker's capability of reusing those memory regions in a parallel thread.

Obviously, all features displayed by Lyra2 for providing protection against slow-memory attacks may also impact the algorithm's performance for legitimate user. After all, they also interfere with the legitimate platform's capability of taking advantage of its own caching and pre-fetching features. Therefore, it is of utmost importance that the algorithm's configuration is optimized to the platform's characteristics, considering aspects such as the amount of RAM available, cache line size, number of processing cores (see Section 6.1), etc. This should allow Lyra2's execution to run more smoothly in the legitimate user's machine while imposing more serious penalties to attackers employing platforms with different characteristics.

5.3   Cache-timing attacks

A cache-timing attack is a type of side-channel attack in which the attacker is able to observe a machine's timing behavior by monitoring its access to cache memory (e.g., the occurrence of cache-misses) [54,47]. This class of attacks has been shown to be effective, for example, against certain implementations of the Advanced Encryption Standard (AES) [55] and RSA [56], allowing the recovery of the secret key employed by the algorithms [54,57].

In the context of password hashing, cache-timing attacks may be a threat against memory-hard solutions that involve operations for which the memory visitation order depends on the password. The reason is that, at least in theory,

a spy process that observes the cache behavior of the correct password may be able to filter passwords that do not match that pattern after only a few iterations, rather than after the whole algorithm is run [47]. Nevertheless, cache-timing attacks are unlikely to be a matter of great concern in scenarios where the PHS runs in a single-user scenario, such as in local authentication or in remote authentications performed in a dedicated server: after all, if attackers are able to insert such spy process into these environments, they are more likely to insert a much more powerful spyware (e.g., a keylogger or a memory scanner) to get the password more directly. On the other hand, cache-timing attacks may be interesting in scenarios where the physical hardware running the PHS is shared by processes of different users, such as virtual servers hosted in a public cloud [58]. The reason is that such environments potentially create the required conditions for making cache-timing measurements [58], but are expected to prevent the installation of a malware powerful enough to circumvent the hypervisor's isolation capability for accessing data from different virtual machines.

In this context, the approach adopted in Lyra2 is to provide resistance against cache-timing attacks only during the Setup phase, in which the indices of the rows read and written are not password-dependent, while the Wandering and Wrap-up phases are susceptible to such attacks. As a result, even though Lyra2 is not completely immune to cache-timing attacks, the algorithm ensures that attackers will have to run the whole Setup phase and at least a portion of the Wandering phase before they can use cache-timing information for filtering guesses. Therefore, such attacks will still involve a memory usage of at least $R/2$ rows or some of the polynomial time-memory trade-offs discussed along Section 5.1.

The reason for this design decision providing partial resistance to cache-timing attacks is threefold. First, as discussed in Section 5.2, making password-dependent memory visitations is one of the main defenses of Lyra2 against slow-memory attacks, since it hinders caching and pre-fetching mechanisms that could accelerate such attacks. Therefore, resistance against low-memory attacks and protection against cache-timing attacks are somewhat conflicting requirements. Since low- and slow-memory attacks are applicable to a wide range of scenarios, from local to remote authentication, it seems more important to protect against them than completely prevent cache-timing attacks.

Second, for practical reasons (namely, scalability) it may be interesting to offload the password hashing process to users, distributing the underlying costs among client devices rather than concentrating them on the server, even in the case of remote authentication. This is the main idea behind the server-relief protocol described in [47], according to which the server sends only the salt to the client (preferably using a secure channel), who responds with $x = $ PHS($pwd, salt$), so the server only computes locally $y = H(x)$ and compares it to the value stored in its own database. The result of the approach is that the server-side computations during authentication are reduced to the computation of the hash, while the memory- and processing-intensive operations involved in the password hashing process are performed by the client, in an environment in which cache-timing is probably a less serious concern.

Third, as discussed in [59], recent advances in software and hardware technology are themselves likely to hinder the feasibility of cache-timing and related attacks due to the amount of "noise" conveyed by their underlying complexity. This technological constraint is also reinforced by the fact that security-aware cloud providers are expected to provide countermeasures against such attacks for protecting their users, such as (see [58] for a more detailed discussion): ensuring that processes run by different users do not influence each other's cache usage (or, at least, that this influence is not completely predictable); or making it more difficult for an attacker to place a spy process in the same physical machine as security-sensitive processes, in especial processes related to user authentication. Therefore, even if these countermeasures are not enough to completely prevent such attacks from happening, the added complexity brought by them may be enough to force the attacker to run a large portion of the Wandering phase, paying the corresponding costs, before a password guess can be reliably discarded.

6   Some possible extensions of Lyra2

In this section, we discuss some possible extensions of the Lyra2 algorithm described in Section 4, which could be integrated into its basic design for exploring different aspects, namely: taking advantage of parallelism capabilities potentially available on the legitimate user's platform; improving the algorithm's usage of the cache in the legitimate platforms for incresing the cost of slow-memory attacks; providing higher resistance against attacks low- and slow-memory attacks at the cost of more frequently cache misses; providing further protection against attacks based on hardware implementations; and allowing finer-grained control over the algorithm's processing time. We consider these modifications "extensions" mainly because they were not yet fully tested and throughly assessed as Lyra2's core, so the value added by them is still under evaluation, as well as their exact details.

Along the discussion, we explain how these changes can be independently integrated into the basic algorithm described in Section 4.1 rather than altogether. The reason for this approach is twofold: convenience for the reader, since each different extension can be plugged into Lyra2 either independently or altogether; and conciseness, given that the proposed extensions, although simple, result in a few additional lines of pseudo-code each. Nonetheless, for the interested reader, we provide in Appendix D. a pseudo-code integrating the first three proposed extensions into Lyra2's core.

6.1   Allowing parallelism on legitimate platforms: Lyra2$_p$

Even though a strictly sequential PHS is interesting for thwarting attacks, this may not be the best choice if the legitimate platform itself has multiple processing units available, such as a GPU, a multicore CPU or even multiple CPUs. In such scenarios, users may want to take advantage of this parallelism for (1) raising the PHS's usage of memory, abundant in a desktop or GPU running a single

PHS instance, while (2) keeping the PHS's total processing time within humanly acceptable limits, possibly using a larger value of T for improving its resistance against time-memory trade-offs.

Against an attacker making several guesses in parallel, this strategy instantly raises the memory costs proportionally to the number of cores used by the legitimate user. For example, if the key is computed from a sequential PHS configured to use 10 MB of memory and to take 1 second to run in a single core, an attacker who has access to 1,000 processing cores and 10 GB of memory could make 1,000 password guesses per second (one per core). If the key is now computed from the output of two instances of the same PHS parametrization, testing a guess would take 20 MB and 1 second, meaning that the attacker would need 20 GB of memory to obtain the same throughput as before.

Therefore, aiming to allow legitimate users to explore their own parallelism capabilities, we propose a slightly tweaked version of Lyra2. We call this variant Lyra2$_p$, where the $p \geqslant 1$ parameter is the desired degree of parallelism, with the restriction that $p|(R/2)$. Before we go into details on Lyra2$_p$'s operation, though, it is useful to briefly mention its rationale. Namely, the idea is to have p parallel threads working on the same memory matrix in such a manner that (1) the different threads do not cause much interference on each other's operation, but (2) everyone of the p slices of the shared memory matrix depends on rows generated from many threads. The first property leads to a smaller need of synchronism between threads, facilitating the algorithm's processing by highly parallel platforms, while the second makes it harder to run each thread separately with a reduced memory usage.

Structure and rationale First, during the Setup phase, p sponge copies are generated. This is done similarly to Lyra2, the difference being that each sponge S_i ($0 \leqslant i \leqslant p-1$), right after being bootstrapped in line 4 of Algorithm 2, must perform one additional full-round and stateful absorb operation on a b-long block containing the $b/2$-bit representation of p concatenated with the $b/2$-bit representation of i, i.e., the block $(Int(p, b/2) \, \| \, Int(i, b/2))$. For example, for $p = 2$, S_0 absorbs the bit-sequence $0^{b/2-2}10 \, \| \, 0^{b/2-1}0$, while S_1 does the same for $0^{b/2-2}10 \, \| \, 0^{b/2-1}1$. As another example, for $p = 4$, the blocks absorbed by sponges S_0 to S_3 would have the same $0^{b/2-3}100$ prefix, but their suffixes would be $0^{b/2-2}00$ to $0^{b/2-2}11$, respectively. This approach ensures that each of the p sponges is initialized with distinct internal states, even though they absorb identical values of *salt* and *pwd*. In addition, the fact that the value of the block absorbed by each sponge depends on p ensures that computations made with $p' \neq p$ cannot be reused in an attack against Lyra2$_p$, an interesting property for scenarios in which the attacker does not know the correct value of p.

The p sponges are evenly distributed over the memory matrix, becoming responsible for p contiguous slices of R/p rows each, hereby denoted M_i ($0 \leqslant i \leqslant p-1$). More formally, slice M_i corresponds to the interval $M[i \cdot R/p]$ to $M[(i+1) \cdot R/p-1]$ of the complete memory matrix, so that $M_i[x] = M[i \cdot R/p+x]$.

The Setup phase of each sponge S_i then proceeds as in the algorithm's non-parallelizable version, the only difference being that they remain limited to their own slices instead of sweeping the whole memory matrix. As a result, all sponges need to be synchronized only when they finish running their own Setup phases, but otherwise they can run in a completely independent manner (their computation is embarrassingly parallel).

After the whole memory matrix is initialized, the Wandering phase proceeds using a strategy very similar to Lyra2's: in each iteration of Time Loop, the order in which rows are visited is reversed, the visitation comprising the duplexing and updating of rows deterministically and pseudorandomly picked. There are, however, two small differences. First, for all S_i, the rows picked in a pseudorandom fashion during the first (resp. second) half of the Visitation Loop are limited to the first (resp. second) half of their own slices M_i. More formally, when the Visitation Loop control variable of S_i is row, the pseudorandom index row^* picked in that iteration is computed in line 20 as "$\texttt{offset} + truncL(rand, W) \oplus prev) \bmod R/2p$", with $\texttt{offset} = 0$ when $row < R/2p$ and $\texttt{offset} = R/2p$ otherwise.

Second, S_i's duplexing operation (line 22) is applied to $M_i[prev] \oplus M_i[row^*] \oplus M_j[row_p^*]$, where $M_j[row_p^*]$ corresponds to a pseudorandom row from a pseudorandom slice $j \neq i$. More precisely, we have $M_j[row_p^*] = M_j[(row^* + R/2p) \bmod R/p]$, meaning that row_p^* refers to an index in M_j that is at the same position as row^* in M_i, except for an offset of $R/2p$. This ensures that S_i reads only in the half of slice M_j that is currently not being processed by S_j. Therefore, as long as all sponges are synchronized every half of their own Visitation Loops, there is no interference between, allowing their processes to run independently. The slice index j, on the other hand, is computed by S_i as follows: set the value of j to the most significant word of $rand$ modulo p (i.e., make $j = (truncM(rand, W) \bmod p)$); if the value of j computed in this manner is such that $j = i$, make it $j = i + dir$ instead. As a result, S_i reads from other slices are expected to follow an approximately uniform distribution, with a small bias toward the slices that are M_i's immediate neighbors. Hence, after $p - 1$ iterations of the Visitation Loop, every sponge is expected to have read from roughly all other slices, obliging an attacker to have all slices in memory and duly updated, or to recompute them on demand and pay the corresponding memory and processing prices. Actually, these pseudorandom reads from other slices do not even have to be as frequent as once per iteration of the Visitation Loop, but the frequency itself could be configurable for reducing the number of reads on far-away regions of the memory by any sponge, accelerating the whole process. For example, if the frequency in which the S_i reads from slice $M_{j \neq i}$ is set to once every R/p^2 iterations of the Visitation Loop, each sponge is already expected to read from approximately all other sponges after one single iteration of the Time Loop.

Finally, the Wrap-up phase of Lyra2$_p$ is analogous to the one used in the algorithm's non-parallelizable version: each sponge S_i absorbs a single cell from its own slice M_i and squeezes k bits. When all sponges finish processing, the p

sub-keys generated in this manner are then XORed together, yielding Lyra2$_p$'s output K.

Preliminary security analysis The main advantage of Lyra2$_p$ when compared to plain Lyra2 is that the former allows the memory matrix to be processed, in theory, p times faster than the latter. In practice, this performance gain is unlikely to be as high as p due to the larger number of pseudorandom reads (and consequent cache misses) performed by the algorithm and need of eventual synchronization among threads. However, for the sake of the argument, consider that p is indeed the acceleration obtained. In what follows, we discuss some ways by which legitimate users may take advantage of this faster operation for raising the algorithm's resistance against attacks. Along the discussion, we use the p subscript to denote Lyra2$_p$ parametrization, while the omission of the subscript indicates the corresponding parameters used in Lyra2 (and, thus, in the security analysis carried out in Section 5).

On one extreme, legitimate users may then decide to use this fact to raise the password hashing memory usage p times while keeping its total processing time unchanged, using as parametrization $R_p = R \cdot p$ and $T_p = T$. Therefore, if the attacker wants to keep only R in memory, the resulting recomputation costs would be analogous to those of an attack against Lyra2 involving only R/p rows. For $p = 2$, for example, the attacker could store M_0, the half of the memory matrix that is known to be needed in any iteration of the Visitation Loop, and recompute rows from M_1 on the fly and with a reduced memory usage.

Unfortunately for the attacker, however, this approach is deemed to involve recomputations of rows from M_1 with a cost of $\mathcal{O}((3R/4)^{2T+2}/(3^T))$ in last iteration of the Time Loop. If, on the other hand, each thread keeps $R/2$ rows, the attacker could take advantage of the fact that each half of the Visitation Loop in Lyra2$_p$ only involves half of a slice: hence, it may appear that storing only the required half would allow the threads to run more smoothly. This strategy would fail, however, because if both S_0 and S_1 do so, each Visitation Loop iteration will require the recomputation of $M_1[row_p^*]$ (resp. $M_0[row_p^*]$) for the used of S_0 (resp. S_1). Alternative memory distributions (e.g., one that explores the "storage-ahead" strategy described in Section 5.1) would result in similar needs for recomputations, leading to similar asymptotic attack costs.

On the other extreme, legitimate users may use the multiple processing cores to raise Lyra2$_p$'s resistance against time-memory trade-offs, by making $R_p = R$ and $T_p = T \cdot p$. In a first analysis, assuming that the security properties of Lyra2 can be directly applied to any single sponge of Lyra2$_p$ operating on R/p rows (as discussed above), the cost of approximately memory-free attacks against any given sponge would become $\mathcal{O}((3R/4p)^{2T_p+2}/(3^T)) = \mathcal{O}((3R/4p)^{2T \cdot p+2}/(3^T))$. Hence, with a high enough value of R/p, the cost of low-memory attacks can be easily brought to unfeasible levels.

6.2 Higher resistance against slow-memory attacks

Another envisioned extension of Lyra2 refers to how the Columns Loop uses the cache lines of the legitimate user's machine. Specifically, while Lyra2's core makes a single deterministic pass over all columns of the Memory matrix in each iteration of Filling/Visitation Loop, performing reads and writes on each column, one could make $\chi \geqslant 0$ pseudorandom reads to those columns right after all of them are modified by the Columns Loop, further updating the sponge's internal state. As a result, a legitimate user who is able to fit in cache the three rows involved in the Columns Loop ($M[prev]$, $M[row]$ and $M[row^*]$) could perform those read operations quite fast, while an adversary using a device with lower cache size or slower memory would pay a penalty in terms of performance. This is analogous to introducing a "cache-oriented" scrypt to the Filling and Visitation Loop: several memory positions are read in a pseudorandom pattern after being modified, but those memory positions are all concentrated on a small area (the cache) for better performance on a legitimate user's platform.

In other to achieve this goal, all that is needed is a loop with χ iterations in which (1) a random column is picked as "$col^* \leftarrow truncL(rand, W) \bmod C$", and (2) an additional reduced-round duplexing operation "$rand \leftarrow H.duplexing_\rho(M[prev][col^*] \oplus M[row^*][col^*] \oplus M[row][col^*], b)$" is performed. Such loop can then be integrated into Lyra2's basic design described in Algorithm 2 right after the end of each Columns Loop, both in the Setup phase (i.e., after line 12) and in the Wandering phase (i.e., after line 25).

6.3 Higher resistance against time-memory trade-offs

One possible adaptation of the algorithm consists in raising the number of rows involved in each iteration of the Visitation Loop. This can be accomplished with the introduction of the following modifications into Algorithm 2:

1. Line 20: we obtain δ pseudorandom indices row_d^* (with $1 \leqslant \delta \leqslant b/W - 1$ and $0 \leqslant d \leqslant \delta - 1$), each of which is computed from different portions of $rand$. A simple but effective way achieving this is to make "$row_d^* \leftarrow (truncL(rotW^d(rand), W) \oplus prev) \bmod R$", where the operation $rotW^d$ corresponds to the iterative application of $rotW$, d times.
2. Line 22: instead of feeding the sponge with the result of XORing the previous row ($M[prev]$) with a single pseudorandomly chosen row ($M[row^*]$), the sponge's input is computed by XORing together $M[prev]$ and δ pseudorandom rows $M[row_d^*]$.
3. Line 24: every pseudorandom row row_d^* involved in the duplexing operation is XORed with a different rotation of $rand$. Specifically, we make "$M[row_d^*][col] \leftarrow M[row_d^*][col] \oplus rotW^{d+1}(rand)$".

This tweak is quite simple but it is also powerful, the main advantage being that it ends up increasing the security of the algorithm against low-memory attacks. The logic is as follows. First, the additional write operations accelerate

the modification of the memory matrix, raising the depth of the dependence tree discussed in Section 5.1. Moreover, the additional read operations raise the diffusion capability of the algorithm because each row of the memory matrix depends on a larger number of other rows, leading to a dependence tree with a higher branching factor. We can then use a reasoning analogous to that of Section 5.1 for computing the resulting cost of an almost memory-free attack against Lyra2 with this extra parameter: ignoring the fact that only a part of each layer ℓ needs to be processed during recomputations, we conjecture that this attack would involve $\mathcal{O}((R/2)^{(\delta+1)\cdot T+2})$ executions of CL and the storage of approximately $(\delta + 1) \cdot T$ rows and sponge states during the last $1/\delta$-th part of the Wandering phase, which should dominate Lyra2's running time.

The main disadvantage of this approach is that the the higher number of randomly picked rows potentially increases the number of cache misses observed during the algorithm's execution, raising the total processing time of Lyra2 for a same parametrization. This may oblige legitimate users to reduce the value of T to keep Lyra2's running time below a certain limit, which in turn would be beneficial to attack platforms able to mask the latency resulting from cache misses (e.g., using the idle cores that are waiting for input to run different password guesses). According to our tests, we observed slow downs from more than 100% to approximately 50% with each increment of δ. Therefore, the interest of such tweak depends on actual tests made on the target platform.

All things considered, it is important to emphasize that a moderately large value of T already leads to highly expensive low-memory attacks, which should be enough to avert them even without this tweak. Nonetheless, this extension might be interesting in many different contexts, such as: platforms that can only afford a very small value of T (e.g., $T = 1$); legitimate platforms that can afford cache misses better than potential attackers; scenarios in which the attacker model includes platforms with a large number of powerful processing cores, but in which memory is scarce; or a combination of such scenarios. In some extreme cases, it may even be appealing to have $\delta > b/W$, which can be easily accomplished if the computation of row_d^* (line 20) actually involves less than one full word of $rand$. After all, only $|R|$ are actually needed to ensure that each row_i^* is computed from a different set of bits. If $|R| < W/2$, for example, we could make "$row_d^* \leftarrow (truncL(rotH^d(rand), W/2) \bmod R)$", where $rotH$ denotes the bitwise left rotation by one half word, and then update each of those rows by making "$M[row_d^*][col] \leftarrow M[row_d^*][col] \oplus rotH^{d+1}(rand)$" In this case, the algorithm would support $1 \leqslant \delta \leqslant 2b/W - 1$.

6.4   Raising the cost of dedicated hardware

One final tweak that may be of interest when the goal is to achieve higher protection against dedicated hardware is to employ a different w-bit permutation f for each (group of) sponge operations executed during Lyra2's execution. Obviously, this would increase the code size in software, but this is unlikely to be a serious burden for a legitimate user in any modern machine, while the same

should not apply to hardware implementations. The reason is that, with this approach, implementing Lyra2 in hardware would require either a circuit with all minimum operations of each permutation or the separate construction of every permutation. Supposing that the permutations have few operations in common, the first approach would lead to a circuit that is approximately as effective as a software implementation on a generic processor The second approach, on the other hand, raises the area occupied by the algorithm in comparison with the scenario in which a single permutation is employed, thus increasing the hardware construction costs. To compensate for this inefficiency in terms of area, one possible strategy is to evaluate more than one password guess at a time with the same hardware, using pipelining strategies for scheduling each guess to different parts of the circuitry. However, if the order in which those permutations are employed by the algorithm is itself pseudorandom and password-dependent, building such pipelines and corresponding scheduling algorithms with a reduced area becomes a very difficult task. In addition, even if such optimized construction is possible, the large amount of memory that must be assigned to each guess for its efficient processing should limit the attacker's ability to use the same hardware for testing a large number of guesses simultaneously.

One simple way of implementing this tweak for selecting the order in which the permutations f_i will be applied, with $0 \leqslant i \leqslant \varphi-1$, is to make a (multi)bitwise evaluation of the φ least significant bits of $rand$. For example, we could consider that every bit position i of $rand$ is associated to an instance f_i, so that, while evaluating those bits in an ascending order, all permutations whose corresponding bit is 1 are applied before the instances associated with a bit 0. Then, for $\varphi = 4$, if the four least significant bits of $rand$ are 0101, this means that f_0, f_2, f_1, f_3 should be applied, in this order, in the next φ (group of) sponge operations. If $rand$ ends with 0100, then the order would be f_2, f_0, f_1, f_3. Notice that this single-bit approach does not lead to a uniform distribution for the order in which the permutations are applied: not only f_0 has a higher probability than $f_{\varphi-1}$ of being picked first, but it also cannot provide more than 2^φ combinations out of $\varphi!$ possibilities. Nonetheless, the pseudorandomness achieved may be already enough to prevent the construction of efficient pipelines or optimized scheduling algorithms, at least for small values of φ, while being very simple to implement (basically, it requires φ bitwise shifts and ANDs). If desired, however, one can obtain a more uniform distribution by associating each permutation with multiple bits of $rand$, so that permutations associated with higher values would be applied before those with lower values. The extreme case is to use at least $|\varphi|$-bit associations, which should lead to a uniform distribution at the cost of executing an algorithm for sorting the permutations from the largest to the smallest value associated to them.

Any of the strategies above can be integrated into Lyra2's code right after line 9 of Algorithm 2, when $rand$ is first initialized. More precisely, Algorithm 2 runs normally until line 9, employing only f_0 for absorbing the password and salt, for initializing rows $M[0]$ and $M[1]$, and for computing $rand$ for the very first time. Right after that, $rand$ is used to define the pseudorandom order in which

each f_i will be used, as explained above, so the next φ iterations of the Setup's Columns Loop use the φ different sponges. During the last of those φ iterations, the value of $rand$ thereby obtained is employed once again to obtain a (probably different) order in which the permutations will be used in the next φ iterations of the Columns Loop. If the cost of this ancillary "permutation selection" process is deemed too high, one could allow the same f_i picked in this manner to be used in the whole Columns Loop, employing the next permutation only in the subsequent iteration of the Filling Loop. Whichever the case, this process is iteratively repeated, covering not only the Setup but also the Wandering and Wrap-up phases of the algorithm.

It is important to notice that, even in cases where the different permutations display distinct processing times, the strategies described still provide a reasonably good resistance against timing attacks by ensuring that all permutations are employed once before any single permutation is executed once again. In addition, this characteristic also helps to keep synchronism between threads if this tweak is combined with the Lyra2$_p$ variant described in Section 6.1. As a last remark, notice that this extension of Lyra2 displays as an additional security the fact that it prevents attackers from determining, a priori, which is the actual permutation employed in the generation of key K during the final squeeze operation performed in line 33 of Algorithm 2.

6.5 Finer-grained processing time

With a simple modification of Algorithm 2, one can merge the Time and Visitation Loops using a single loop-controlling variable γ rather than two, T and R. As a result, the total number of duplexing operations does not have to be an integer multiple of R, but can be configured in a more fine-grained manner.

We notice that this approach would only slightly affect the algorithm's security against low-memory attacks, which happens because some rows of the memory matrix end up not being as required as others. For example, if $\gamma = R/2$, only the first half of the Visitation Loop would be executed, meaning that the entire second half of the memory matrix would certainly be visited and modified, while rows from its first half would be required with a probability of 50%. All in all, most of the analysis concerning the algorithm's resistance against attacks would still apply to this extension: it still carries the same features that thwart slow-memory and cache-timing attacks, while the cost of low-memory attacks against it would be at least as high as those associated with the plain version of Lyra2 for $T = \lfloor \gamma/(R/2) \rfloor$.

7 Performance for some recommended parameters

In our assessment of Lyra2's performance, we used a reference implementation of Blake2b's compression function [39] as the underlying sponge's f function of Algorithm 2 (i.e., without any of the extensions described in section 6). The

implementations employed, as well as test vectors, are available at `www.lyra-kdf.net`.

One important note about this implementation is that, even though sponges typically have their internal state initialized with zeros, in this case we initialize the state's least significant 512 bits to zeros, but the remainder 512 bits are set to to Blake2b's Initialization Vector. The reason is that Blake2b does not use the constants originally employed in Blake2 inside its G function [39], relying on the IV for avoiding possible fixed points. Indeed, if the internal state is filled with zeros, any block filled with zeros absorbed by the sponge will not change this state value. This should not be a critical issue for Lyra unless both the password and the salt are both strings of zeros and long enough to fill whole blocks, since otherwise the pad10*1 padding would already avoid the fixed point even if the input itself is filled with zeros. However, the adopted approach is both more cautious and more compliant with Blake2b's specification (which is not designed as a sponge).

The results obtained with an implementation having no SSE2 optimizations are illustrated in Figure 14. The results depicted correspond to the average execution time of Lyra2 configured with $C = 128$, $\rho = 1$, $b = 768$ bits (i.e., the inner state has 256 bits), and different T and R settings, giving an overall idea of possible combinations of parameters and the corresponding usage of resources. As shown in this figure, Lyra2 is expected to be able to execute in: less than 1 s while using up to 400 MB (with $R = 2^{15}$ and $T = 5$) or up to 1 GB of memory (with $R \approx 8.3 \cdot 10^4$ and $T = 1$); or in less than 5 s with 1.6 GB (with $R = 2^{17}$ and $T = 6$). All tests were performed on an Intel Core i5-2500 (3.30 GHz Dual Core, 64 bits) equipped with 8 GB of DRAM, running Ubuntu 13.04 64 bits. The source code was compiled using gcc 4.6.4 with `-O3` optimization.
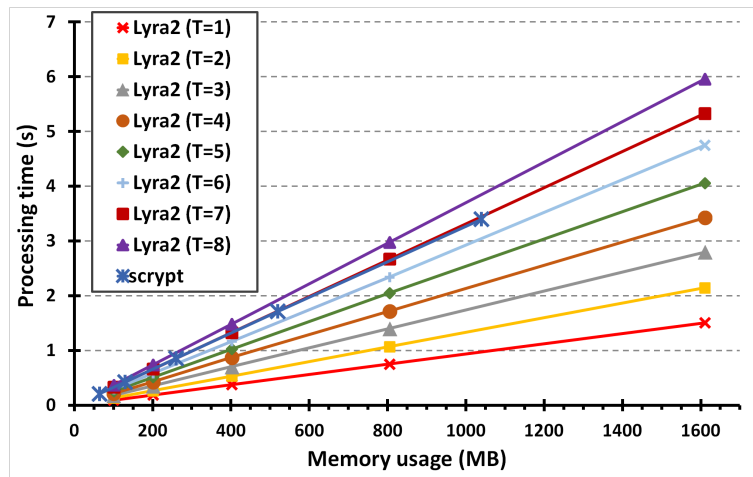

Fig. 14. Performance of our non-SSE Lyra2 implementation, for $C = 128$, $\rho = 1$, and different T and R settings, compared with non-SSE scrypt.
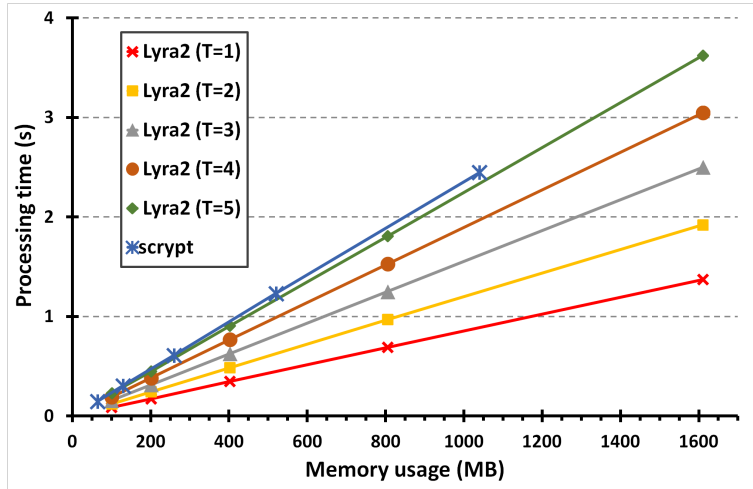
Fig. 15. Performance of our SSE-enabled Lyra2 implementation, for $C = 128$, $\rho = 1$, and different T and R settings, , compared with SSE-enabled scrypt.

Figure 14 also compares Lyra2 with the scrypt "optimized non-SSE2" implementation publicly available at `www.tarsnap.com/scrypt.html`, using the parameters suggested by scrypt's author in [5] (namely, $b = 8192$ and $p = 1$). The "non-SSE2" version of scrypt was chosen aiming at a fair comparison, since the particular Lyra2 implementation used in these tests do not explore SSE2 instructions either. The results obtained show that, in order to achieve a memory usage and processing time similar to that of scrypt, Lyra2 could be configured with $T \approx 7$.

We also used the same testbed for evaluating a very simple SSE2-enabled implementation of Lyra2, aiming to assess the potential of the algorithm in taking advantage of resources available in modern processors. Specifically, this version uses only quite obvious instructions for optimizing Lyra2, while the underlying SSE-enabled Blake2b code corresponds to the implementation by Samuel Neves available at [39]. Figure 15 compares this simple implementation of Lyra2 with the SSE2-enabled version of scrypt (also available at `www.tarsnap.com/scrypt.html`). As shown in this figure, those simple optimizations were enough to obtain a gain of 10% in Lyra2's execution time, allowing the algorithm to process 1.6 GB in less than 1.5 s (with $R = 2^{17}$ and $T = 1$). On the other hand, with the SSE-enabled scrypt code employed (which counts with a better SSE-oriented coding) the efficiency gain was considerably superior, reaching $\approx 25\%$ and approaching the SSE-enabled Lyra2's performance for $T = 5$. Developing and evaluating a similarly optimized SSE-oriented implementation of Lyra2 remains, however, as a matter of future work.

7.1   Expected attack costs

Considering that the cost of DDR3 SO-DIMM memory chips is currently around U\$10.00/GB [60], Table 1 shows the cost added by Lyra2 with $T = 5$ when an attacker tries to crack a password in 1 year using the above reference hardware, for different password strengths — we refer the reader to [7, Appendix A] for a discussion on how to compute the approximate entropy of passwords. These costs are obtained considering the total number of instances that need to run in parallel to test the whole password space in 365 days and supposing that testing a password takes the same amount of time as in our testbed. Notice that, in a real scenario, attackers would also have to consider costs related to wiring and energy consumption of memory chips, besides the cost of the processing cores themselves.

We notice that if the attacker uses a faster platform (e.g., an FPGA or a more powerful computer), these costs should drop proportionally, since a smaller number of instances (and, thus, memory chips) would be required for this task. Similarly, if the attacker employs memory devices faster than regular DRAM (e.g., SRAM or registers), the processing time is also likely to drop, reducing the number of instances required to run in parallel. Nonetheless, in this case the resulting memory-related costs may actually be significantly bigger due to the higher cost per GB of such memory devices. Anyhow, the numbers provided in Table 1 are not intended as absolute values, but rather a reference on how much extra protection one could expect from using Lyra2, since this additional memory-related cost is the main advantage of any PHS that explores memory usage when compared with those that do not.

Finally, when compared with existing solutions that do explore memory usage, Lyra2 is advantageous due to the elevated processing costs of attack venues involving time-memory trade-offs, effectively discouraging such approaches.

Indeed, considering the final Wandering phase alone and $T = 5$, the additional processing cost of a memory-free attack against Lyra2 is approximately $((3 \cdot 2^{15}/4)^{12})/(2 \cdot 3^5) \approx 2^{166} \cdot \sigma$ if the algorithm operates with 400 MB, or $((3 \cdot 2^{17}/4)^{12})/(2 \cdot 3^5) \approx 2^{190} \cdot \sigma$ for a memory usage of 1.6 GB. For the same

Password	Memory usage (MB) for $T = 1$				Memory usage (MB) for $T = 5$			
entropy (bits)	200	400	800	1,600	200	400	800	1,600
35	380.5	1.5k	6.1k	24.1k	985.4	4.0k	15.8k	63.6k
40	12.2k	48.7k	194.0k	770.0k	31.5k	127.1k	507.2k	2.1M
45	389.7k	1.6M	6.2M	24.6M	1.0M	4.1M	16.2M	65.1M
50	12.5M	49.9M	198.6M	788.4M	32.3M	130.1M	519.3M	2.1B
55	399.0M	1.6B	6.4B	25.2B	1.0B	4.2B	16.6B	66.6B

Table 1. Memory-related cost (in U\$) added by the SSE-enable version of Lyra2 with $T = 1$ and $T = 5$, for attackers trying to break passwords in a 1-year period using an Intel Core i5-2500 or equivalent processor.

memory usage settings, the total cost of a similar memory-free attack against scrypt would be approximately $(2^{15})^2/2 = 2^{29}$ and $(2^{17})^2/2 = 2^{33}$ calls to *Block-Mix*, whose processing time is approximately 2σ for the parameters used in our experiments. As expected, such elevated processing costs are prone to discourage attack venues that try to avoid the memory costs of Lyra2 by means of extra processing.

7.2 Preliminary GPU performance tests

Aiming to have a preliminary evaluation of Lyra2 in a GPU, we prepared a simple implementation of the algorithm in CUDA, which was built to run in a single thread and not use the device's shared memory. This is far from being a perfectly GPU-oriented setting, but at least it gives some insight on the performance of Lyra2 in a scenario involving multiple password guesses being performed in parallel with a limited amount of dedicated memory for each of them.

Basically, the code obtained is a direct port of the CPU code, with some small adaptations for ensuring compatibility and good performance (considering the hardware and the virtual machine instruction sets) with an NVIDIA GeForce GTX470 (Fermi architecture) [61]. This GPU board has 448 CUDA cores (14 Multiprocessors with 32 CUDA Cores each) operating at 1.22 GHz, and a total amount of memory of 1280 MB, operating at 1.67 GHz. We used the CUDA 5.0 driver and configured the architecture to 2.0 (the higher value allowed by the board).
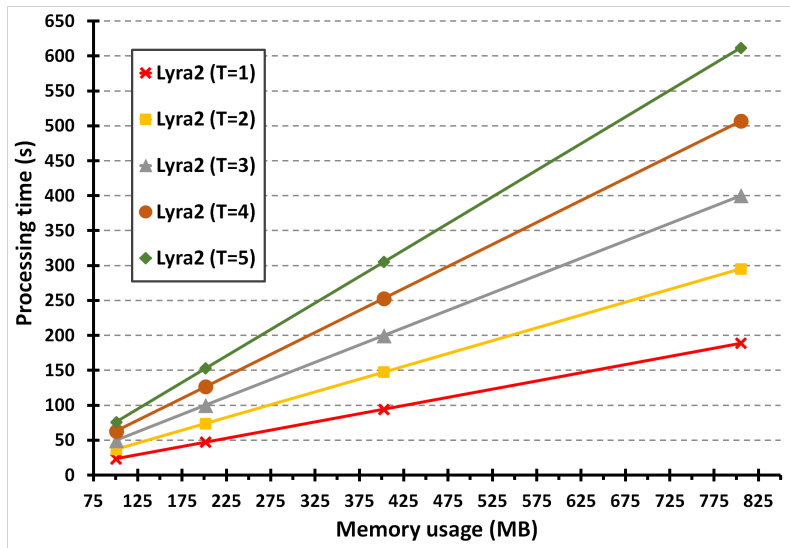

Fig. 16. Performance of a preliminary GPU-oriented Lyra2 implementation, for $C = 128$, $\rho = 1$, and different T and R settings, on NVIDIA GeForce GTX470.

The preliminary results obtained for an average of six executions of Lyra2 with $C = 128$ and different T and R settings are shown in Figure 16. From the numbers obtained, we can see that the performance of the GPU was very low, especially with higher values of T. This performance penalty is most likely due to the latency caused by the pseudorandom access pattern adopted in Lyra2, since GPUs are usually optimized to memory accesses in a certain interval or following a certain pattern. This latency could, in principle, be masked by the GPU if it was running several threads in parallel. However, if Lyra2 is configured to use a high enough amount of memory, the number of parallel threads are deemed to be low and, thus, GPUs are unlike to efficiently hide this latency for providing a considerable performance gain when compared with the results hereby obtained. Moreover, even in this case the pseudorandom pattern would still be a problem for the GPU, since it would oblige the board to frequently transfer memory from the global memory to the shared memory and vice-versa. This happens because the shared memory of the GPU used as testbed has only 48KiB, much less than what would be necessary to fit a considerable number of rows (each of which occupies 12KiB for $C = 128$), let alone the whole memory matrix. For a large enough memory matrix, this issue is likely to be similarly observed even in more powerful GPUs.

8   Conclusions

We presented Lyra2, a password hashing scheme (PHS) that allows legitimate users to fine tune memory and processing costs according to the desired level of security and resources available in the target platform. For achieving this goal, Lyra2 builds on the properties of sponge functions operating in a stateful mode, creating a strictly sequential process. Indeed, the whole memory matrix of the algorithm can be seen as a huge state, which changes together with the sponge's internal state.

The ability to control Lyra2's memory usage allows legitimate users to thwart attacks using parallel platforms. This can be accomplished by raising the total memory required by the several cores beyond the amount available in the attacker's device. In summary, the combination of a strictly sequential design, the high costs of exploring time-memory trade-offs, and the ability to raise the memory usage beyond what is attainable with similar-purpose solutions (e.g., scrypt) for a similar security level and processing time make Lyra2 an appealing PHS solution.

Finally, with the proposed (and possibly other) extensions, Lyra2 can be further personalized for different scenarios, including parallel legitimate platforms (with the p parameter). Assessing the interest of such tweaks and their potential integration into Lyra2's core remains, however, as a matter of future work.

Acknowledgements

References

1. Chakrabarti, S., Singbal, M.: Password-based authentication: Preventing dictionary attacks. Computer **40**(6) (june 2007) 68–74
2. Conklin, A., Dietrich, G., Walz, D.: Password-based authentication: A system perspective. In: Proc. of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04). Volume 7 of HICSS'04., Washington, DC, USA, IEEE Computer Society (2004) 170–179
3. Bonneau, J., Herley, C., van Oorschot, P.C., Stajano, F.: The Quest to Replace Passwords: A Framework for Comparative Evaluation of Web Authentication Schemes. In: IEEE Symposium on Security and Privacy. (2012) 553–567
4. NIST: Special Publication 800-18 – Recommendation for Key Derivation Using Pseudorandom Functions. National Institute of Standards and Technology, U.S. Department of Commerce. (October 2009) `http://csrc.nist.gov/publications/nistpubs/800-108/sp800-108.pdf`.
5. Percival, C.: Stronger key derivation via sequential memory-hard functions. In: BSDCan 2009 – The Technical BSD Conference. (2009)
6. Kaliski, B.: PKCS#5: Password-Based Cryptography Specification version 2.0 (RFC 2898). (2000)
7. NIST: Special Publication 800-63-1 – Electronic Authentication Guideline. National Institute of Standards and Technology, U.S. Department of Commerce. (December 2011) `http://csrc.nist.gov/publications/nistpubs/800-63-1/SP-800-63-1.pdf`.
8. Florencio, D., Herley, C.: A Large Scale Study of Web Password Habits. In: Proc. of the 16th International Conference on World Wide Web, Alberta, Canada (2007) 657–666
9. Herley, C., van Oorschot, P., Patrick, A.: Passwords: If We're So Smart, Why Are We Still Using Them? In: Financial Cryptography and Data Security. Volume 5628 of LNCS., Springer Berlin / Heidelberg (2009) 230–237
10. Sprengers, M.: GPU-based Password Cracking: On the Security of Password Hashing Schemes regarding Advances in Graphics Processing Units. Master's thesis, Radboud University Nijmegen (2011)
11. Dürmuth, M., Güneysu, T., Kasper, M.: Evaluation of Standardized Password-Based Key Derivation against Parallel Processing Platforms. In: Computer Security – ESORICS 2012. Volume 7459 of LNCS. Springer Berlin Heidelberg (2012) 716–733
12. Marechal, M.: Advances in password cracking. Journal in Computer Virology **4**(1) (2008) 73–81
13. Provos, N., Mazières, D.: A future-adaptable password scheme. In: Proc. of the FREENIX track: 1999 USENIX annual technical conference. (1999)
14. PHC: Password Hashing Competition. `https://password-hashing.net/` (2013)

15. Almeida, L., Andrade, E., Barreto, P., Simplicio, M.: Lyra: Password-Based Key Derivation with Tunable Memory and Processing Costs. Journal of Cryptographic Engineering **4**(2) (2014) 75–89 See also `eprint.iacr.org/2014/030`.

16. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: Sponge functions. (ECRYPT Hash Function Workshop 2007) (2007) Also available at `http://csrc.nist.gov/pki/HashWorkshop/Public_Comments/2007_May.html`.

17. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: Cryptographic sponge functions - version 0.1. http://keccak.noekeon.org/ (2011)

18. Andreeva, E., Mennink, B., Preneel, B.: The Parazoa family: Generalizing the Sponge hash functions. IACR Cryptology ePrint Archive **2011** (2011) 28

19. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: The Keccak SHA-3 submission. Submission to NIST (Round 3) (2011)

20. Kelsey, J., Schneier, B., Hall, C., Wagner, D.: Secure Applications of Low-Entropy Keys. In: Proc. of the 1st International Workshop on Information Security. ISW '97, London, UK, UK, Springer-Verlag (1998) 121–134

21. Weir, M., Aggarwal, S., Medeiros, B.d., Glodek, B.: Password Cracking Using Probabilistic Context-Free Grammars. In: Proc. of the 30th IEEE Symposium on Security and Privacy. SP'09, Washington, DC, USA, IEEE Computer Society (2009) 391–405

22. Nvidia: CUDA C programming guide. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/` (2012)

23. Khronos Group: The OpenCL Specification – Version 1.2. (2012)

24. Nvidia: Tesla Kepler family product overview. `http://www.nvidia.com/content/tesla/pdf/Tesla-KSeries-Overview-LR.pdf` (2012)

25. Dandass, Y.S.: Using FPGAs to Parallelize Dictionary Attacks for Password Cracking. In: Proc. of the 41st Annual Hawaii International Conference on System Sciences (HICSS 2008), IEEE (2008) 485–485

26. Kakarountas, A.P., Michail, H., Milidonis, A., Goutis, C.E., Theodoridis, G.: High-Speed FPGA Implementation of Secure Hash Algorithm for IPSec and VPN Applications. The Journal of Supercomputing **37**(2) (2006) 179–195

27. Chung, E.S., Milder, P.A., Hoe, J.C., Mai, K.: Single-Chip Heterogeneous Computing: Does the Future Include Custom Logic, FPGAs, and GPGPUs? In: Proc. of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO'43, Washington, DC, USA, IEEE Computer Society (2010) 225–236

28. Fowers, J., Brown, G., Cooke, P., Stitt, G.: A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications. In: Proc. of the ACM/SIGDA Internbational Symposium on Field Programmable Gate Arrays (FPGA'12), New York, NY, USA, ACM (2012) 47–56

29. SciEngines: Rivyera s3-5000. `http://sciengines.com/products/computers-and-clusters/rivyera-s3-5000.html`

30. SciEngines: Rivyera v7-2000t. `http://sciengines.com/products/computers-and-clusters/v72000t.html`

31. NIST: Federal Information Processing Standard (FIPS PUB 198) – The Keyed-Hash Message Authentication Code. National Institute of Standards and Technology, U.S. Department of Commerce. (March 2002) `http://csrc.nist.gov/publications/fips/fips198/fips-198a.pdf`.

32. Bernstein, D.: The Salsa20 family of stream ciphers. In Robshaw, M., Billet, O., eds.: New Stream Cipher Designs. Springer-Verlag, Berlin, Heidelberg (2008) 84–97

33. Aumasson, J.P., Fischer, S., Khazaei, S., Meier, W., Rechberger, C.: New features of latin dances: Analysis of Salsa, ChaCha, and Rumba. In: Fast Software Encryption. Volume 5084., Berlin, Heidelberg, Springer-Verlag (2008) 470–488

34. Daemen, J., Rijmen, V.: A new MAC construction ALRED and a specific instance ALPHA-mac. In: Fast Software Encryption – FSE'05. (2005) 1–17

35. Daemen, J., Rijmen, V.: Refinements of the ALRED construction and MAC security claims. Information Security, IET **4**(3) (2010) 149–157

36. Simplicio, M.A., Barbuda, P., Barreto, P., Carvalho, T., Margi, C.: The MARVIN Message Authentication Code and the LETTERSOUP Authenticated Encryption Scheme. Security and Communication Networks **2** (2009) 165–180

37. Simplicio, M.A., Barreto, P.: Revisiting the Security of the ALRED Design and Two of Its Variants: Marvin and LetterSoup. IEEE Transactions on Information Theory **58**(9) (2012) 6223–6238

38. Gaj, K., Homsirikamol, E., Rogawski, M., Shahid, R., Sharif, M.U.: Comprehensive Evaluation of High-Speed and Medium-Speed Implementations of Five SHA-3 Finalists Using Xilinx and Altera FPGAs. Cryptology ePrint Archive, Report 2012/368 (2012) `http://eprint.iacr.org/2012/368`.

39. Aumasson, J.P., Neves, S., Wilcox-O'Hearn, Z., Winnerlein, C.: BLAKE2: simpler, smaller, fast as MD5. `https://blake2.net/` (2013)

40. Aumasson, J.P., Henzen, L., Meier, W., Phan, R.: SHA-3 proposal BLAKE (version 1.3). `https://131002.net/blake/blake.pdf` (2010)

41. Chang, S., Perlner, R., Burr, W.E., Turan, M.S., Kelsey, J.M., Paul, S., Bassham, L.E.: Third-Round Report of the SHA-3 Cryptographic Hash Algorithm Competition. US Department of Commerce, National Institute of Standards and Technology (2012)

42. Aumasson, J.P., Guo, J., Knellwolf, S., Matusiewicz, K., Meier, W.: Differential and Invertibility Properties of BLAKE. In Hong, S., Iwata, T., eds.: Fast Software Encryption. Volume 6147 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2010) 318–332 See also `http://eprint.iacr.org/2010/043`.

43. Ming, M., Qiang, H., Zeng, S.: Security analysis of BLAKE-32 based on differential properties. In: 2010 International Conference on Computational and Information Sciences (ICCIS), IEEE (2010) 783–786

44. Guo, J., Karpman, P., Nikoli, I., Wang, L., Wu, S.: Analysis of BLAKE2. In: Topics in Cryptology (CT-RSA 2014). Volume 8366 of LNCS., Springer International Publishing (2014) 402–423 see also `https://eprint.iacr.org/2013/467`.

45. Halderman, J., Schoen, S., Heninger, N., Clarkson, W., Paul, W., Calandrino, J., Feldman, A., Appelbaum, J., Felten, E.: Lest we remember: cold-boot attacks on encryption keys. Commun. ACM **52**(5) (May 2009) 91–98

46. Yuill, J., Denning, D., Feer, F.: Using deception to hide things from hackers: Processes, principles, and techniques. Journal of Information Warfare **5**(3) (2006) 26–40

47. Forler, C., Lucks, S., Wenzel, J.: Catena: A Memory-Consuming Password Scrambler. Cryptology ePrint Archive, Report 2013/525 (2013) `http://eprint.iacr.org/2013/525`.

48. Cook, S.A.: An Observation on Time-storage Trade off. In: Proc. of the 5th Annual ACM Symposium on Theory of Computing (STOC'73), New York, NY, USA, ACM (1973) 29–33

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

50. Dwork, C., Naor, M., Wee, H.: Pebbling and Proofs of Work. In: Advances in Cryptology – CRYPTO 2005. Volume 3621 of Lecture Notes in Computer Science., Springer Berlin Heidelberg (2005) 37–54
51. Dziembowski, S., Kazana, T., Wichs, D.: Key-Evolution Schemes Resilient to Space-Bounded Leakage. In: Advances in Cryptology – CRYPTO 2011. Volume 6841 of Lecture Notes in Computer Science., Springer Berlin Heidelberg (2011) 335–353
52. Lengauer, T., Tarjan, R.E.: Asymptotically tight bounds on time-space trade-offs in a pebble game. J. ACM **29**(4) (oct 1982) 1087–1130
53. Ariew, R.: Ockham's Razor: A Historical and Philosophical Analysis of Ockham's Principle of Parsimony. University of Illinois press, Champaign-Urbana (1976)
54. Bernstein, D.J.: Cache-timing attacks on AES. Technical report, University of Illinois (2005) `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`.
55. NIST: Federal Information Processing Standard (FIPS 197) – Advanced Encryption Standard (AES). National Institute of Standards and Technology. (November 2001) `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`.
56. Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM **21**(2) (Feb 1978) 120–126
57. Percival, C.: Cache missing for fun and profit. In: Proc. of BSDCan 2005. (2005)
58. Ristenpart, T., Tromer, E., Shacham, H., Savage, S.: Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds. In: Proc.s of the 16th ACM Conference on Computer and Communications Security. CCS '09, New York, NY, USA, ACM (2009) 199–212
59. Mowery, K., Keelveedhi, S., Shacham, H.: Are AES x86 Cache Timing Attacks Still Feasible? In: Proc.s of the 2012 ACM Workshop on Cloud Computing Security Workshop (CCSW'12), New York, NY, USA, ACM (2012) 19–24
60. TrendForce: DRAM contract price (jan.22 2014). http://www.trendforce.com/price (visited on Mar.29, 2014) (2014)
61. GeForce: GeForce GTX 470: Specifications. `http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-470/specifications` (visited on Mar.29, 2014) (2014)
62. TrueCrypt: TrueCrypt: Free open-source on-the-fly encryption – documentation. `http://www.truecrypt.org/docs/` (2012)
63. Apple: iOS security. Technical report, Apple Inc. (2012) `http://images.apple.com/ipad/business/docs/iOS_Security_May12.pdf`.
64. Yao, F.F., Yin, Y.L.: Design and Analysis of Password-Based Key Derivation Functions. IEEE Transactions on Information Theory **51**(9) (2005) 3292–3297
65. Bellare, M., Ristenpart, T., Tessaro, S.: Multi-instance security and its application to password-based cryptography. In: Advances in Cryptology (CRYPTO 2012). Volume 7417 of LNCS., Springer Berlin Heidelberg (2012) 312–329
66. Schneier, B.: Description of a new variable-length key, 64-bit block cipher (Blowfish). In: Fast Software Encryption, Cambridge Security Workshop, London, UK, Springer-Verlag (1994) 191–204
67. Crew, B.: New carnivorous harp sponge discovered in deep sea. Nature (2012) Available online: `http://www.nature.com/news/new-carnivorous-harp-sponge-discovered-in-deep-sea-1.11789`.

Appendix A. PBKDF2

The Password-Based Key Derivation Function version 2 (PBKDF2) algorithm [6] was originally proposed in 2000 as part of RSA Laboratories' PKCS#5. It is

nowadays present in several security tools, such as TrueCrypt [62] and Apple's iOS for encrypting user passwords [63], and has been formally analyzed in several circumstances [64,65].

Basically, PBKDF2 (see Algorithm 3) iteratively applies the underlying pseudorandom function $Hash$ to the concatenation of pwd and a variable U_i, i.e., it makes $U_i = Hash(pwd, U_{i-1})$ for each iteration $1 \leqslant i \leqslant T$. The initial value U_0 corresponds to the concatenation of the user-provided $salt$ and a variable l, where l corresponds to the number of required output blocks. The l-th block of the k-long key is then computed as $K_l = U_1 \oplus U_2 \oplus \ldots \oplus U_T$, where k is the desired key length.

PBKDF2 allows users to control its total running time by configuring the T parameter. Since the password hahsing process is strictly sequential (one cannot compute U_i without first obtaining U_{i-1}), its internal structure is not parallelizable. However, as the amount of memory used by PBKDF2 is quite small, the cost of implementing brute force attacks against it by means of multiple processing units remains reasonably low.

Algorithm 3 PBKDF2.

INPUT: pwd ▷ The password
INPUT: $salt$ ▷ The salt
INPUT: T ▷ The user-defined parameter
OUTPUT: K ▷ The password-derived key

1: **if** $k > (2^{32} - 1) \cdot h$ **then**
2:     **return** $Derived\ key\ too\ long.$
3: **end if**
4: $l \leftarrow \lceil k/h \rceil$; $r \leftarrow k - (l - 1) \cdot h$
5: **for** $i \leftarrow 1$ **to** l **do**
6:     $U[1] \leftarrow PRF(pwd, salt \| INT(i))$ ▷ INT(i): 32-bit encoding of i
7:     $T[i] \leftarrow U[1]$
8:     **for** $j \leftarrow 2$ **to** T **do**
9:         $U[j] \leftarrow PRF(pwd, U[j-1])$; $T[i] \leftarrow T[i] \oplus U[j]$
10:     **end for**
11:     **if** $i = 1$ **then** $K \leftarrow T[1]$ **else** $K \leftarrow K \| T[i]$ **end if**
12: **end for**
13: **return** K

Appendix B. Bcrypt

Another solution that allows users to configure the password hashing processing time is bcrypt [13]. The scheme is based on a customized version of the 64-bit cipher algorithm Blowfish [66], called *EksBlowflish* ("expensive key schedule blowfish").

Both algorithms use the same encryption process, differing only on how they compute their subkeys and S-boxes. Bcrypt consists in initializing EksBlowfish's subkeys and S-Boxes with the salt and password, using the so-called EksBlowfish-Setup function, and then using EksBlowfish for iteratively encrypting a constant string, 64 times.

Algorithm 4 Bcrypt.

INPUT: *pwd*   ▷ The password
INPUT: *salt*   ▷ The salt
INPUT: *T*   ▷ The user-defined cost parameter]
OUTPUT: *K*   ▷ The password-derived key

1: $s \leftarrow InitState()$ ▷ Copies the digits of π into the sub-keys and S-boxes S_i
2: $s \leftarrow$ ExpandKey$(s, salt, pwd)$
3: **for** $i \leftarrow 1$ **to** 2^T **do**
4:    $s \leftarrow$ ExpandKey$(s, 0, salt)$
5:    $s \leftarrow$ ExpandKey$(s, 0, pwd)$
6: **end for**
7: $ctext \leftarrow "OrpheanBeholderScryDoubt"$
8: **for** $i \leftarrow 1$ **to** 64 **do**
9:    $ctext \leftarrow BlowfishEncrypt(s, ctext)$
10: **end for**
11: **return** $T \parallel salt \parallel ctext$

12: **function** EXPANDKEY$(s, salt, pwd)$
13:    **for** $i \leftarrow 1$ **to** 32 **do**
14:       $P_i \leftarrow P_i \oplus pwd[32(i-1)\ldots 32i-1]$
15:    **end for**
16:    **for** $i \leftarrow 1$ **to** 9 **do**
17:       $temp \leftarrow BlowfishEncrypt(s, salt[64(i-1)\ldots 64i-1])$
18:       $P_{0+2(i-1)} \leftarrow temp[0\ldots 31]$
19:       $P_{1+2(i-1)} \leftarrow temp[32\ldots 64]$
20:    **end for**
21:    **for** $i \leftarrow 1$ **to** 4 **do**
22:       **for** $j \leftarrow 1$ **to** 128 **do**
23:          $temp \leftarrow BlowfishEncrypt(s, salt[64(j-1)\ldots 64j-1])$
24:          $S_i[2(j-1)] \leftarrow temp[0\ldots 31]$
25:          $S_i[1+2(j-1)] \leftarrow temp[32\ldots 63]$
26:       **end for**
27:    **end for**
28:    **return** s
29: **end function**

EksBlowfishSetup starts by copying the first digits of the number π into the subkeys and S-boxes S_i (see Algorithm 4). Then, it updates the subkeys and S-boxes by invoking $ExpandKey(salt, pwd)$, for a 128-bit salt value. Basically, this function (1) cyclically XORs the password with the current subkeys, and then (2) iteratively blowfish-encrypts one of the halves of the salt, the resulting ciphertext being XORed with the salt's other half and also replacing the next two subkeys (or S-Boxes, after all subkeys are replaced). After all subkeys and S-Boxes are updated, bcrypt alternately calls $ExpandKey(0, salt)$ and then $ExpandKey(0, pwd)$, for 2^T iterations. The user-defined parameter T determines, thus, the time spent on this subkey and S-Box updating process, effectively controlling the algorithm's total processing time.

Like PBKDF2, bcrypt allows users to parameterize only its total running time. In addition to this shortcoming, some of its characteristics can be consid-

ered (small) disadvantages when compared with PBKDF2. First, bcrypt employs a dedicated structure (EksBlowfish) rather than a conventional hash function, leading to the need of implementing a whole new cryptographic primitive and, thus, raising the algorithm's code size. Second, EksBlowfishSetup's internal loop grows exponentially with the T parameter, making it harder to fine-tune bcrypt's total execution time without a linearly growing external loop. Finally, bcrypt displays the unusual (albeit minor) restriction of being unable to handle passwords having more than 56 bytes.

Appendix C. Lyra

Lyra's steps as described in [15] are detailed in Algorithm 5.

Algorithm 5 The Lyra Algorithm.

PARAM: $Hash$ ▷ Sponge with block size b (in bits) and underlying permutation f
PARAM: ρ ▷ Number of rounds of f in the Setup and Wandering phases
INPUT: pwd ▷ The password
INPUT: $salt$ ▷ A random salt
INPUT: T ▷ Time cost, in number of iterations
INPUT: R ▷ Number of rows in the memory matrix
INPUT: C ▷ Number of columns in the memory matrix
INPUT: k ▷ The desired key length, in bits
OUTPUT: K ▷ The password-derived k-long key

1: ▷ **Setup**: Initializes a $(R \times C)$ memory matrix whose cells have b bits each
2: $Hash.absorb(\texttt{pad}(salt \,\|\, pwd))$ ▷ Padding rule: 10*1
3: $M[0] \leftarrow Hash.squeeze_\rho(C \cdot b)$
4: **for** $row \leftarrow 1$ **to** $R - 1$ **do**
5:      **for** $col \leftarrow 0$ **to** $C - 1$ **do**
6:          $M[row][col] \leftarrow Hash.duplexing_\rho(M[row - 1][col], b)$
7:      **end for**
8: **end for**

9: ▷ **Wandering**: Iteratively overwrites blocks of the memory matrix
10: $row \leftarrow 0$
11: **for** $i \leftarrow 0$ **to** $T - 1$ **do** ▷ **Time Loop**
12:      **for** $j \leftarrow 0$ **to** $R - 1$ **do** ▷ **Rows Loop**: randomly visits R rows
13:          **for** $col \leftarrow 0$ **to** $C - 1$ **do** ▷ **Columns Loop**: visits blocks in row
14:              $M[row][col] \leftarrow M[row][col] \oplus Hash.duplexing_\rho(M[row][col], b)$
15:          **end for**
16:          $col \leftarrow M[row][C - 1] \bmod C$
17:          $row \leftarrow Hash.duplexing(M[row][col], |R|) \bmod R$
18:      **end for**
19: **end for**

20: ▷ **Wrap-up**: key computation
21: $Hash.absorb(\texttt{pad}(salt))$ ▷ Uses the sponge's current state
22: $K \leftarrow Hash.squeeze(k)$

23: **return** K ▷ Outputs the k-long key

Like in Lyra2, Lyra also employs (reduced-round) operations of a cryptographic sponge for building a memory matrix, visiting its rows in a pseudo-random fashion, and providing the desired number of bits as output. One first difference between the two algorithms is that Lyra's Setup is quite simple, each iteration of its loop (lines 8 to 4) duplexing only the row that was computed in the previous iteration. As a result, the Setup can be executed with a cost of $R \cdot \sigma$ while keeping in memory a single row of the memory matrix instead of half of them as in Lyra2. The second and probably main difference is that Lyra's duplexing operations performed during the Wandering phase only involve one pseudorandomly-picked row, which is read and written upon, while two rows are modified per duplexing in Lyra2's basic algorithm. This is the reason why the processing time of an approximately memory-free attack against Lyra grows with a R^{T+1} factor. In comparison, as discussed in Section 5.1, in Lyra2's basic algorithm the cost of such attacks involves a R^{2T+2} factor, or $R^{(\delta+1)T+2}$ if the δ parameter is also employed.

Appendix D. The Extended Lyra2 algorithm

Algorithm 6 describes Lyra2 when integrated with a basic version of the extensions described in Sections 6.1, 6.2 and 6.3.

Appendix E. On the algorithm's name

The name "Lyra" comes from *Chondrocladia lyra*, a recently discovered type of sponge [67]. While most sponges are harmless, this harp-like sponge is carnivorous, using its branches to ensnare its prey, which is then enveloped in a membrane and completely digested. The "two" suffix is a reference to its predecessor, Lyra [15], which displays many of Lyra2's properties hereby presented but has a lower resistance to attacks involving time-memory trade-offs.

Lyra2's memory matrix displays some similarity with this species' external aspect, and we expect it to be at least as much aggressive against adversaries trying to attack it. ☺

Algorithm 6 The Lyra2 Algorithm, with p, δ and χ parameters.

PARAM: H ▷ Sponge with block size b (in bits) and underlying permutation f
PARAM: ρ ▷ Number of rounds of f during the Setup and Wandering phases
PARAM: W ▷ The target machine's word size (usually, 32 or 64)
PARAM: δ ▷ Number of pseudorandom rows involved in the duplexing operations
PARAM: p ▷ Degree of parallelism ($p \geqslant 1$ and $p|(R/2)$)
PARAM: χ ▷ Number of iterations in which rows in cache are repeatedly read after modification
INPUT: pwd ▷ The password
INPUT: $salt$ ▷ A salt
INPUT: T ▷ Time cost, in number of iterations
INPUT: R ▷ Number of rows in the memory matrix
INPUT: C ▷ Number of columns in the memory matrix ($C \geqslant 2$)
INPUT: k ▷ The desired key length, in bits
OUTPUT: K ▷ The password-derived k-long key

1: **for each** i in $[0, p[$ **do**   ▷ Operation performed in parallel, by each thread
2:     ▷ **Setup phase**: Initializes one out of p slices of the $(R \times C)$ memory matrix
3:     $basil \leftarrow len(k)||len(pwd)||len(salt)||T||R||C$ ▷ Multi-byte representation of inputs
4:     $H.absorb(\mathtt{pad}(pwd \,\|\, salt \,\|\, basil))$ ▷ Padding rule: 10*1.
5:     **if** $p > 1$ **then** $H_i.absorb(Int(p, b/2) \,\|\, Int(i, b/2))$ **end if**   ▷ Absorbs extra block
6:     $M_i[0] \,\|\, M_i[1] \leftarrow H_i.squeeze_\rho(2C \cdot b)$
7:     $row^* \leftarrow 0$; $prev \leftarrow 1$; $row \leftarrow 2$ ▷ The first and second rows will feed the sponge
8:     **do**   ▷ **Filling Loop**: initializes remainder rows
9:         **for** $col \leftarrow 0$ **to** $C - 1$ **do**   ▷ **Columns Loop**: updates both $M_i[row]$ and $M_i[row^*]$
10:             $rand \leftarrow H_i.duplexing_\rho(M_i[prev][col] \oplus M_i[row^*][col], b)$
11:             $M_i[row][col] \leftarrow rand$
12:             $M_i[row^*][col] \leftarrow M_i[row^*][col] \oplus rotW(rand)$ ▷ $rotW()$: right rotation of 1 word
13:         **end for**
14:         **for** $z \leftarrow 1$ **to** χ **do**   ▷ **Cache Loop**: reads rows already in cache
15:             $col^* \leftarrow truncL(rand, W) \bmod C$ ▷ Picks a pseudorandom column
16:             $rand \leftarrow H.duplexing_\rho(M_i[prev][col^*] \oplus M_i[row^*][col^*] \oplus M_i[row][col^*], b)$
17:         **end for**
18:         **if** $row^* \neq 0$ **then** $row^* \leftarrow row^* - 1$ **else** $row^* \leftarrow prev$ **end if**
19:         $prev \leftarrow row$; $row \leftarrow row + 1$ ▷ The next row in sequence will be initialized
20:     **while** $(row \leqslant R/p - 1)$
21:     ▷ **Wandering phase**: Iteratively overwrites cells of the memory matrix
22:     $dir \leftarrow -1$; $prev \leftarrow 0$; $row \leftarrow R/p - 1$ ▷ Start visiting rows in reverse order
23:     **for** $\tau \leftarrow 1$ **to** T **do**   ▷ **Time Loop**
24:         **do**   ▷ **Visitation Loop**: reverses the row visitation order
25:             **if** $p > 1$ **then**   ▷ Prepares the rows to be visited, including $M_j[row_p^*]$
26:                 **if** $row < R/2p$ **then** $\mathtt{offset} \leftarrow 0$ **else** $\mathtt{offset} \leftarrow R/2p$ **end if**
27:                 **for** $d \leftarrow 0$ **to** $\delta - 1$ **do**   ▷ δ random rows will be visited
28:                     $row_d^* \leftarrow \mathtt{offset} + ((truncL(rotW^d(rand), W) \oplus prev) \bmod (R/2p))$
29:                 **end for**
30:                 $row_p^* \leftarrow (row_{d=0}^* + \mathtt{offset}) \bmod (R/p)$
31:                 $j \leftarrow (truncM(rand, W) \bmod p)$
32:                 **if** $j = i$ **then** $j \leftarrow i + dir$ **end if**
33:             **else**   ▷ No parallelism: no need to consider different slices
34:                 **for** $d \leftarrow 0$ **to** $\delta - 1$ **do**   ▷ δ random rows will be visited
35:                     $row_d^* \leftarrow (truncL(rotW^d(rand), W) \oplus prev) \bmod R$
36:                 **end for**
37:             **end if**   ▷ The pseudorandom indices are picked
38:             **for** $col \leftarrow 0$ **to** $C - 1$ **do**   ▷ **Columns Loop**: updates $M_i[row]$ and $M_i[row_i^*]$
39:                 $food \leftarrow M_i[prev][col] \oplus M_i[row_{d=0}^*][col] \oplus \ldots \oplus M_i[row_{d=\delta-1}^*][col]$
40:                 **if** $p > 1$ **then** $food \leftarrow food \oplus M_j[row_p^*][col]$ **end if**
41:                 $rand \leftarrow H_i.duplexing_\rho(food, b)$
42:                 $M_i[row][col] \leftarrow M_i[row][col] \oplus rand$
43:                 **for** $d \leftarrow 0$ **to** $\delta - 1$ **do**
44:                     $M_i[row_d^*][col] \leftarrow M_i[row_d^*][col] \oplus rotW^{d+1}(rand)$
45:                 **end for**
46:             **end for**
47:             **for** $z \leftarrow 1$ **to** χ **do**   ▷ **Cache Loop**: reads rows already in cache
48:                 $col^* \leftarrow truncL(rand, W) \bmod C$ ▷ Picks a pseudorandom column
49:                 $food \leftarrow M_i[prev][col^*] \oplus M_i[row_{d=0}^*][col^*] \oplus \ldots \oplus M_i[row_{d=\delta-1}^*][col^*]$
50:                 **if** $p > 1$ **then** $food \leftarrow food \oplus M_j[row_p^*][col^*]$ **end if**
51:                 $rand \leftarrow H_i.duplexing_\rho(food, b)$
52:             **end for**
53:             $prev \leftarrow row$; $row \leftarrow row + dir$ ▷ The next row in sequence will be visited
54:         **while** $(0 \leqslant row \leqslant R - 1)$
55:         $dir \leftarrow -dir$; $prev \leftarrow R - row$; $row \leftarrow row + dir$ ▷ Inverses the visitation order
56:     **end for**
57:     ▷ **Wrap-up phase**: key computation
58:     $H_i.absorb(M_i[row_0^*][0])$ ▷ Absorbs a final column with the full-round sponge
59:     $K_i \leftarrow H_i.squeeze(k)$ ▷ Squeezes k bits with a full-round sponge
60: **end for**   ▷ All threads finished
61: **return** $K_0 \oplus \ldots \oplus K_p$ ▷ Provides k-long bitstring as output

---