The Lyra2 reference guide

Marcos A. Simplicio Jr[1]
(mjunior@larc.usp.br)
Leonardo C. Almeida[1]
(lalmeida@larc.usp.br)
Ewerton R. Andrade[1]
(eandrade@larc.usp.br)
Paulo C. F. dos Santos[1]
(pcarlos@ime.usp.br)
Paulo S. L. M. Barreto[1]
(pbarreto@larc.usp.br)

http://www.lyra2.net/

Version 2.5
August 25, 2014
São Paulo, Brazil

[1]Escola Politécnica
Universidade de São Paulo (Poli-USP)

Revision Pane

- **Version 2.3.0 (30-Mar-2014)**: Original version. Submitted to the Password Hashing Competition (PHC) as "v0".
- **Version 2.3.1 (01-Apr-2014)**: Details added: (1) little endianness; (2) initialization of underlying sponge's state; (3) block length used in benchmarks.
- **Version 2.3.2 (04-Apr-2014)**: Details added: *params* (instead of discussing that extra parameters could appear as part of the salt). Update: Inversion on the order in which the salt and password are fed into the sponge (easier to accommodate *params* and follows the general rule "feed data into hash functions in order of decreasing entropy" [39].) Submitted to the Password Hashing Competition (PHC) as "v1".
- **Version 2.4 (10-Jun-2014)**: Details added: (1) table with basic notation for easier reference; (2) extended discussion on underlying sponge (Section 4.4). Also, the following updates were made:

  1. The rows fed to sponge are not XORed anymore, but combined using wordwise addition (i.e., ignoring carries between words), denoted $\boxplus$. The reason is that the XOR operation could cancel previous sponge outputs that composed the value of the rows input to the sponge, thus requiring a more careful management of which pairs of rows would feed the sponge. This does not happen with the $\boxplus$ operation without impacting the algorithm's performance.
  2. In the Setup phase, aiming for better uniformity, only $M[0]$ is obtained by squeezing the sponge, while $M[1]$ is obtained by duplexing $M[0]$. The idea is to enforce the rule that "every row is initialized during Setup by the duplexing of two rows, the immediately previous one and another deterministically picked". In the case of $M[0]$, the rows fed are $M[-2]$ and $M[-1]$, while for $M[1]$ they are $M[-1]$ and $M[0]$, where both $M[-1]$ and $M[-2]$ are filled with zeros. Those "extra rows" are not explicitly shown in the pseudocode, though, because they are unnecessary in practice **(NOTE: this tweak was extended to $M[2]$ in the revision from 25-Aug-2014 — see modification ?? on that version).**.
  3. In the Setup phase, the columns of each row written from the highest to the lowest index, although they are still read in the "usual" order (from the lowest to the highest index). This tweak thwarts attacks in which previous rows are discarded for saving memory and then recomputed right before they are used, as discussed in the new section 5.1.2.5.
  4. In the Setup phase, (1) the visitation of previous rows now interleaves indices from the first and second halves of a window containing all previously initialized rows (when all the concerned rows are visited once each, the window doubles in size and the process restarts); (2) the initialization of rows changed from ($M[row] \leftarrow rand$) to ($M[row] \leftarrow M[prev] \oplus rand$). Both modifications are intended to improve the Setup phase's security against low-memory attacks. Specifically, the first ensures that the revisited rows are

not all clustered together but scattered along the memory matrix; this makes it harder to postpone the recomputation of rows until the final phase of the Setup, when many rows are not required anymore and, thus, the corresponding memory can be diverted for accelerating the recomputation process. The second, on its turn, hides the value of the sponge's output, $rand$, which can only be retrieved from $M[row]$ if one also knows $M[prev]$; since $rand$ is used to modify previously initialized rows, this increases the burden of recomputations. Combined, these modifications have a small impact on the algorithm's performance, slowing the Setup phase by $\approx 4\%$, but this seems to be a worthy trade-off.

5. In the Wandering phase, the visitation of rows now interleaves indices from the first and second halves of a window containing all rows of the memory matrix, using a step of $((R/2) - 1)\tau \bmod R$. This was done for better uniformity with the Setup phase, since this makes the Wandering phase's deterministic visitations similar to those obtained during Setup. As such, it should also provide a similar improvement in terms of security, although this also leads to a performance penalty of $\approx 1\%$ to the Setup phase. **(NOTE: this tweak was superseded in the revision from 25-Aug-2014).**
6. In the Wandering phase, the computation of the random index row^* was simplified, going from $(row^* \leftarrow (\mathrm{LSW}(rand) \oplus prev) \bmod R)$ to $(row^* \leftarrow (\mathrm{LSW}(rand)) \bmod R)$. This simplification was possible due to the adoption of the $\boxplus$ operation when feeding rows to the sponge, which made the "$\oplus prev$" operation unnecessary.

- **Version 2.5 (25-Aug-2014)**:
- Some small modifications in notation: each row fed to the sponge is named row^i instead of row and row^*; the wordwise truncation $\mathrm{truncL}(x, W)$ is denoted simply $\mathrm{LSW}(x)$ — "least-significant word" —; the "basil" variable was renamed "params", which describes better its contents.
- Security analysis updated: instead of covering only attacks with a very small amount of memory, the analysis now considers attacks that iteratively reduce the memory usage by half.
- Instead of recommending right rotations by one word, we are now recommending rotations by a multiple of words. The reason is that vectorized implementations of the Lyra2 may take more advantage of SIMD registers that span multiple words (e.g., 128 bits for a 64-bit machine): by configuring the rotations to match these registers' limits, this operation can be performed simply by rearranging registers instead of performing actual rotations or shifts on those registers. To better mark this change, $rotW$ was renamed $rotRt$, where L is a user-defined parameter.
- During the Setup and the Wandering phases, the sponge is now fed with all rows that have been revisited and modified in the previous iteration rather than only one. Those rows, denoted $M[prev^d]$, are likely to be in cache anyway, so the performance impact is imperceptible according to our tests, but this puts an extra burden on attackers trying to recompute previously discarded rows during a low-memory attack. As a result, the core algorithm of Lyra2 now duplexes the wordwise addition of three (instead of two) rows during the Setup phase

and of four (instead of three) rows during the Wandering phase. These numbers are only smaller for $M[0]$, $M[1]$ and $M[2]$, however, because they are initialized before there are three rows available to be processed by the sponge.

- In the Setup phase, the columns of $M[prev^d]$ are read in a pseudorandom manner rather than deterministically. This ensures that the algorithm does small random reads on the cache, which has no perceptible impact on its performance in a legitimate machine but should incur penalties on machines with smaller cache lines. This was originally suggested as a possible extension, but it is now integrated into the algorithm's core design.
- During the Wandering phase, all rows are now pseudorandomly picked. This simplifies the algorithm and its security analysis, since it hinders attackers trying to prefetch rows in a slow=memory attack or recompute previously discarded rows in a low memory attack, while introducing no impact in the algorithm's performance. This modification supersedes modification from version 2.4.
- In the Wrap-up phase, the final absorb operation is performed over a pseudorandom column of rather than on the first column of row^0. This provides better uniformity while slightly improving security, since absolutely any cell of the memory matrix may now the target of this full-round operation.

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.

Contents

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 [15, 18]. For better or for worse, and despite the existence of many proposals for their replacement [14], 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) [47]. Typically, the output of the PHS is employed in one of two manners [51]: 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 [38, 51].

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 [48]. Indeed, a study from 2007 with 544,960 passwords from real users has shown an average entropy of approximately 40.5 bits [28], 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 [15, 36], 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 [25, 62] or custom hardware [25, 42].

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 [15]. For this reason, modern password hashing solutions usually employ mechanisms for increasing the *cost* of brute force attacks. Schemes such as PBKDF2 [38] and bcrypt [54], 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 [51], 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 [53].

Aiming to address this need for stronger alternatives, our studies led to the proposal of Lyra [1], a mode of operation of cryptographic sponges [11, 12] 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 analyzes 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 Background: Cryptographic Sponges

The concept of *cryptographic sponges* was formally introduced by Bertoni *et al.* in [11] and is described in detail in [12]. The elegant design of sponges has also motivated the creation of more general structures, such as the Parazoa family of functions [2]. Indeed, their flexibility is probably among the reasons that led Keccak [13], one of the members of the sponge family, to be elected as the new Secure Hash Algorithm (SHA-3).

2.1 Notation and Conventions

In what follows and throughout this document, we use the notation show in Table 1. All operations are made assuming a little-endian convention, and should be adapted accordingly for big-endian architectures (this applies basically to the *rotRt* operation).

Symbol	*Meaning*
$\oplus$	bitwise Excusive-OR (XOR) operation
&	bitwise AND operation
$\boxplus$	wordwise add operation (i.e., ignoring carries between words)
$\|\|$	concatenation
$\|x\|$	bit-length of x, i.e., the minimum number of bits required for representing x
$len(x)$	byte-length of x, i.e., the minimum number of bytes required for representing x
$Int(x, y)$	the y-bit representation of number x
LSW(x)	the least significant word of x
$rotRt(x)$	L-bit right rotation of x
$rotRt^y(x)$	L-bit right rotation of x repeated y times

Table 1: *Basic notation used throughout the document.*

2.2 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,

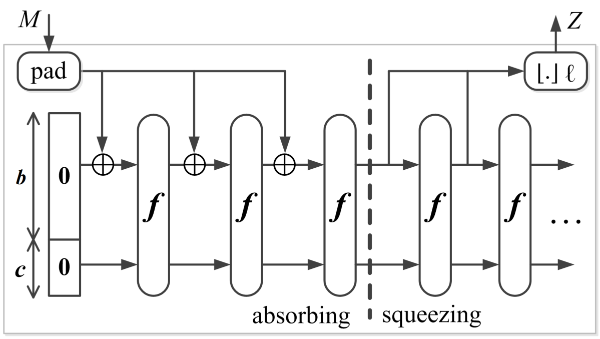

Figure 1: *Overview of the sponge construction $Z = [f, \texttt{pad}, b](M, \ell)$. Adapted from [12].*

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 $w = b + c$ 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 [13]).

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.

2.3 The duplex construction

A similar structure derived from the sponge concept is the *Duplex construction* [12], depicted in Figure 2.

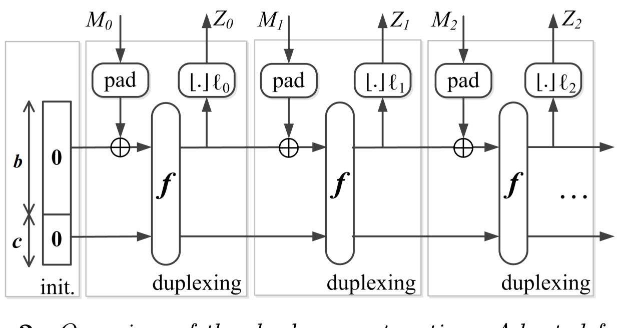

Figure 2: *Overview of the duplex construction. Adapted from [12].*

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 [40]. 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 [65, 48]).

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* [40, 38]. 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 [38].

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 [25].

3.1.1 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 [49] and OpenCL [41]). As a result, they became more intensively employed for more general purposes, including password cracking [62, 25].

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 [50]. Its computational power can also be further expanded by using the host machine's resources [49], 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.

3.1.2 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 [23, 37]). 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 [42]. When compared to GPUs, FPGAs may also be advantageous due to the latter's considerably lower energy consumption [17, 30], which can be further reduced if its circuit is synthesized in the form of custom logic hardware (ASIC) [17].

A recent example of password cracking using FPGAs is presented in [25]. Using a RIVYERA S3-5000 cluster [58] 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) [25, 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 [58] 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 [59]. 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 [59]. 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 [53]: PBKDF2 [38], bcrypt [54] and scrypt [51]. 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 [51] 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

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: $(B_0...B_{p-1}) \leftarrow$PBKDF2$_{HMAC-SHA-256}(pwd, salt, 1, p \cdot b)$
 2: for $i \leftarrow 0$ to $p-1$ do
 3:     $B_i \leftarrow$ROMIX$(B_i, R)$
 4: end for
 5: $K \leftarrow$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:         $V_i \leftarrow X$ ; $X \leftarrow$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:         $X \leftarrow$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
```

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 [51] 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 [46] 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 [9] 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.

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 [4] are not expected to put the security of scrypt in hazard [51], 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 [47]. Internally, the scheme's memory is organized as a matrix that is expected to remain in memory

Algorithm 2 The Lyra2 Algorithm.

```
PARAM: $H$   ▷ Sponge with block size $b$ (in bits) and underlying permutation $f$
PARAM: $H_\rho$   ▷ Reduced-round sponge for use in the Setup and Wandering phases (e.g., $f$ with $\rho$)
PARAM: $Rt$   ▷ Number of bits to be used in rotations (recommended: a multiple of the machine's word size, $W$)
INPUT: $pwd$   ▷ The password
INPUT: $salt$   ▷ A salt
INPUT: $T$   ▷ Time cost, in number of iterations ($T \geqslant 1$)
INPUT: $R$   ▷ Number of rows in the memory matrix (recommended: a power of two)
INPUT: $C$   ▷ Number of columns in the memory matrix (recommended: $C \cdot \rho \geqslant \rho_{max}$)
INPUT: $k$   ▷ The desired key length, in bits
OUTPUT: $K$   ▷ The password-derived $k$-long key
1: ▷ BOOTSTRAPPING PHASE: Initializes the sponge's state and local variables
2: $params \leftarrow len(k) \| len(pwd) \| len(salt) \| T \| R \| C$   ▷ Byte representation of input parameters (others can be added)
3: $H.absorb(\mathtt{pad}(pwd \| salt \| params))$   ▷ Padding rule: $10^*1$. Password can be overwritten after this point
4: $gap \leftarrow 1$ ; $stp \leftarrow 1$ ; $wnd \leftarrow 2$   ▷ Initializes visitation step and window
5: $prev^0 \leftarrow 2$ ; $row^1 \leftarrow 1$ ; $prev^1 \leftarrow 0$
6: ▷ SETUP PHASE: Initializes a $(R \times C)$ memory matrix, it's cells having $b$ bits each
7: for ($col \leftarrow 0$ to $C-1$) do {$M[0][C-1-col] \leftarrow H_\rho.squeeze(b)$} end for   ▷ Initializes $M[0]$
8: for ($col \leftarrow 0$ to $C-1$) do {$M[1][C-1-col] \leftarrow M[0][col] \oplus H_\rho.duplex(M[0][col], b)$} end for ▷ Initializes $M[1]$
9: for ($col \leftarrow 0$ to $C-1$) do ▷ Initializes $M[2]$ and updates $M[0]$
10:     $rand \leftarrow H_\rho.duplex(M[0][col] \boxplus M[1][col])$
11:     $M[2][C-1-col] \leftarrow M[1][col] \oplus rand$
12:     $M[0][col] \leftarrow M[0][col] \oplus rotRt(rand)$   ▷ $rotRt()$: right rotation by $L$ bits (e.g., 1 or more words)
13: end for
14: for ($row^0 \leftarrow 3$ to $R-1$) do   ▷ Filling Loop: initializes remainder rows
15:     for ($col \leftarrow 0$ to $C-1$) do   ▷ Columns Loop: $M[row^0]$ is initialized, while $M[row^1]$ is updated
16:         $rand \leftarrow H_\rho.duplex(M[row^1][col] \boxplus M[prev^0][col] \boxplus M[prev^1][col], b)$
17:         $M[row^0][C-1-col] \leftarrow M[prev^0][col] \oplus rand$
18:         $M[row^1][col] \leftarrow M[row^1][col] \oplus rotRt(rand)$
19:     end for
20:     $prev^0 \leftarrow row^0$ ; $prev^1 \leftarrow row^1$ ; $row^1 \leftarrow (row^1 + stp) \bmod wnd$   ▷ Picks rows to be revisited in next loop
21:     if ($row^1 = 0$) then   ▷ Window fully revisited
22:         $stp \leftarrow wnd + gap$ ; $wnd \leftarrow 2 \cdot wnd$ ; $gap \leftarrow -gap$   ▷ Doubles window size and roughly doubles step
23:     end if
24: end for
25: ▷ WANDERING PHASE: Iteratively overwrites pseudorandom cells of the memory matrix
26: for ($\tau \leftarrow 1$ to $T$) do   ▷ Time Loop
27:     for ($i \leftarrow 0$ to $R-1$) do   ▷ Visitation Loop: $2R$ rows revisited in pseudorandom fashion
28:         for ($d \leftarrow 0$ to 1) do {$row^d \leftarrow (\mathrm{LSW}(rotRt^d(rand))) \bmod R$} end for ▷ Picks pseudorandom rows
29:         for ($col \leftarrow 0$ to $C-1$) do   ▷ Columns Loop: updates each $M[row^d]$
30:             for ($d \leftarrow 2$ to 3) do {$col^d \leftarrow (\mathrm{LSW}(rotRt^d(rand))) \bmod C$} end for   ▷ Picks pseudorandom columns
31:             $rand \leftarrow H_\rho.duplex(M[row^0][col] \boxplus M[row^1][col] \boxplus M[prev^0][col^0] \boxplus M[prev^1][col^1], b)$
32:             for ($d \leftarrow 0$ to 1) do
33:                 $M[row^d][col] \leftarrow M[row^d][col] \oplus rotRt^d(rand)$   ▷ Updates the $d$ pseudorandom rows
34:             end for
35:         end for   ▷ End of Columns Loop
36:         for ($d \leftarrow 0$ to 1) do {$prev^d \leftarrow row^d$} end for   ▷ Next iteration revisits most recently updated rows
37:     end for   ▷ End of Visitation Loop
38: end for   ▷ End of the Time Loop
39: ▷ WRAP-UP PHASE: key computation
40: $H.absorb(M[row^0][col^0])$   ▷ Absorbs a final column with the full-round sponge
41: $K \leftarrow H.squeeze(k)$   ▷ Squeezes $k$ bits with the full-round sponge
42: return $K$   ▷ Provides $k$-long bitstring as output
```

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 core of the Lyra2 algorithm in detail and discuss its design rationale and resulting properties. Later, in Section 6, we discuss some possible variants of the algorithm that may be useful in different scenarios.

4.1 Structure and rationale

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

4.1.1 Bootstrapping

The very first part of Lyra2 comprises the *Bootstrapping* of the algorithm's sponge and internal variables (lines 1 to 5). The set of variables $\{gap, stp, wnd, prev^0, row^1, prev^1\}$ initialized in lines 4 and 5 are useful only for the next stage of the algorithm, the Setup phase, so their discussion is left to Section 4.1.2.

Lyra2's sponge is initialized by absorbing the (properly padded) password and salt, together with a *params* bitstring, initializing a *salt*- and *pwd*-dependent state (line 3). The padding rule adopted by Lyra2 is the multi-rate padding `pad10*1` described in [12], hereby denoted simply `pad`. This padding strategy 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. Since the password itself is not used in any other part of the algorithm, it can be discarded (e.g., overwritten with zeros) after this point.

In this first absorb operation, the goal of the *params* bitstring is basically to avoid collisions using trivial combinations of salts an passwords: for example, for any $(u, v \mid u + v = \alpha)$, we have a collision if $pwd = \mathtt{0}^u$, $salt = \mathtt{0}^v$ and *params* is an empty string; however, this should not occur if *params* explicitly includes u and v. Therefore, *params* can be seen as an "extension" of the salt, including any amount of additional information, such as: the list of parameters passed to the PHS (including the lengths of the salt, password, and output); a user identification string; a domain name toward which the user is authenticating him/herself (useful in remote authentication scenarios); among others.

4.1.2 The Setup phase

Once the internal state of the sponge is initialized, Lyra2 enters the *Setup Phase* (lines 6 to 24). 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).

For better performance when dealing with a potentially large memory matrix, the Setup relies on a "reduced-round sponge", i.e., the sponge's operation are 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}. The advantage of using a reduced-round f is that this approach accelerates the sponge's operations and, thus, it allows more memory positions to be covered than with the application of a full-round f in a same amount of time. The adoption of 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 [21, 22, 60, 61]. 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. For even higher speed, it may even be interesting to use a different function as f_ρ rather than a reduced-round version of f itself, as long the alternative satisfies the above-mentioned properties.

Except for rows $M[0]--M[2]$, the sponge's reduced duplexing operation $H_\rho.duplex$ is always called over the wordwise addition of three rows (line 16), all of which must be available in memory for the algorithm to proceed (see the Filling Loop, in lines 14–24).

- $M[prev^0]$: the last row ever initialized, i.e., $M[prev^0] = M[row^0 - 1]$ in any iteration of the Filling Loop;
- $M[row^1]$: a row that has been previously initialized and is now revisited; and
- $M[prev^1]$: the last row ever revisited (i.e., the most recently row indexed by row^1).

Given the short time between the computation and usage of $M[prev^0]$ and $M[prev^1]$, accessing them in a regular execution of the algorithm should not be a huge burden since both are likely to remain in cache. The same convenience does not apply to $M[row^1]$, though, since it is picked from a window comprising rows initialized prior to $M[prev^0]$. In addition, attacks in which a given $M[row^0]$ is computed from the corresponding inputs may not benefit from the caching of $M[prev^0]$ and $M[prev^1]$, so all three rows may end up coming from the main memory rather than from cache. This is expected to raise the costs of such recomputations. A similar effect could be achieved if the rows provided as the sponge's input were concatenated, but adding them together instead is advantageous because then the duplexing operation involves a single call to the underlying (reduced-round) f rather than three.

After the reduced duplexing operation is performed, its output ($rand$) then modifies two rows (lines 17 and 18): $M[row^0]$, which has not been initialized yet, receives the values of $rand$ XORed with $M[prev^0]$; meanwhile, the columns of the already initialized row $M[row^1]$ have their values updated after being XORed with $rotRt(rand)$, i.e., $rand$ rotated to the right by Rt bits. More formally, for $Rt = W$ and 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 that $M[row^0][C-1-i] \leftarrow rand[i] \oplus M[prev^0][i]$ and $M[row^1][i] \leftarrow M[row^1][i] \oplus rand[(i+1) \bmod (b/W)]$

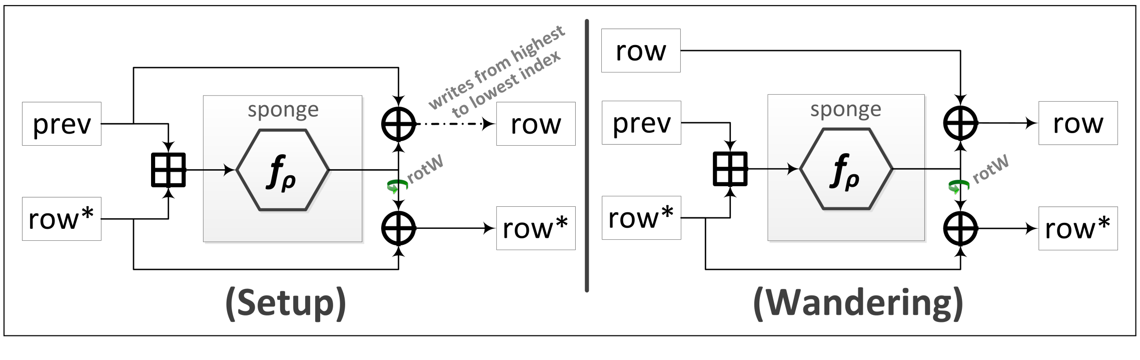

Figure 3: *Handling the sponge's inputs and outputs during the Setup (left) and Wandering (right) phases in Lyra2.*

row^0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	...
$prev^0$	–	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	...
row^1	–	–	0	1	0	3	2	1	0	3	6	1	4	7	2	5	0	9	2	B	4	D	6	F	8	1	A	3	...
$prev^1$	–	–	–	0	1	0	3	2	1	0	3	6	1	4	7	2	5	0	9	2	B	4	D	6	F	8	1	A	...
wnd	–	–	1	2	2	4	4	4	4	8	8	8	8	8	8	8	8	10	10	10	10	10	10	10	10	10	10	10	...

Table 2: *Indices of the rows that feed the sponge when computing $M[row]$ during Setup (hexadecimal notation).*

($0 \leqslant i \leqslant b/W - 1$). We notice that the rows are written from the highest to the lowest index, although read in the inverse order, which thwarts attacks in which previous rows are discarded for saving memory and then recomputed right before they are used, as further discussed in Section 5.1.2.5. In addition, thanks to the *rotRt* operation, each row receives slightly different outputs from the sponge, which reduces an attacker's ability to get useful results from XORing pairs of rows together. Notice that this rotation can be performed basically for free if Rt is set to a multiple of W as recommended: in this case, this operation corresponds to rearranging words rather than actually executing shifts or rotations.

The initialization of $M[0] - -M[2]$ in lines 7 to 13 follows the exact same principle as that of the Filling Loop, but they have to be treated explicitly because they do not have enough predecessors. The left side of Figure 3 illustrates how the sponge's inputs and output are handled by the Lyra2 algorithm during the Setup phase.

The Setup phase ends when all R rows of the memory matrix are initialized, which also means that any row ever indexed by row^1 has also been updated since its initialization. These row^1 indices are deterministically picked from a window that starts with a single row and doubles in size whenever all of its rows are visited (i.e., whenever row^1 reaches the value 0). The exact values assumed by row^1 in this window depend on the visitation step stp, which is approximately half the window size (wnd): it starts at index 1 (line 4) and is updated to $wnd/2 + (-1)^{\lg(wnd)}$ (line 22) whenever the window size changes. Table 2 shows some examples of the values of row^1 in each iteration of the Filling Loop (lines 14–24), as well as the corresponding window size. We note that, since the window size is always a power of 2, the modular operation in line 20 can be implemented with a simple bitwise AND with $wnd - 1$, potentially leading to better performance.

The particular approach adopted for computing the visitation step, in its turn, was motivated by the fact that it leads to several interesting features. First, it ensures that all rows in the window are visited, since stp is odd and wnd is a power of two and, thus, they are coprime. Second, since stp is approximately half the window size, row^1 alternates between indexes of rows from the first and second halves of the window, ensuring an uniform distribution between rows that have been modified since their original computation (the former) and those that have not (the latter). Third, for any given $M[row^1]$ in a window, its immediate successor $M[row^1 + 1]$ in that window is required approximately after $wnd/2$ iterations of the Filling Loop. Since the recomputation of $M[row^1 + 1]$ requires its predecessor $M[row^1]$ as part of the sponge's input, this places an additional burden on attackers trying to run Lyra2 with less memory than legitimate users (see discussion in section 5.1): discarding $M[row^1]$ after usage for saving memory makes the future recomputation of $M[row^1 + 1]$ more difficult; conversely, recomputing $M[row^1 + 1]$ right before discarding $M[row^1]$ is not very advantageous, since $M[row^1 + 1]$ will only be required as the sponge's input many iterations later.

Finally, even though computing stp as either "$wnd/2+1$" or "$wnd/2-1$" in line 22 would provide the properties above, the oscillation between equations was adopted because, then, rows revisited in one window are revisited in the reverse order in the next window. For example, as shown in Table 2, while $M[4]$ is revisited before $M[2]$ when $wnd = 8$, this order is reversed when $wnd = 16$. As a result, and according to our experiments, recomputing several values of $M[row^1]$ in the sequence they are required by the algorithm becomes harder with such alternating pattern.

4.1.3 The Wandering phase

The most time-consuming of all phases, the *Wandering Phase* (lines 26 to 38), 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 consists in the reduced duplexing of rows that are added together (line 31) for computing a random-like output $rand$ (line 31), which is then XORed with rows taken as input. One distinct aspect of the Wandering phase, however, refers to the way it handles the sponge's inputs and outputs, as illustrated in the right side of Figure 3. Namely, besides taking four rows rather than three as input for the sponge, these rows are not all deterministically picked anymore, but all involve some kind of pseudorandom, password-dependent variable in their picking and visitation:

- row^d $(d = 0, 1)$: indices computed in line 28 from the first and second words of the sponge's outer state, i.e., from $rand[0]$ and $rand[1]$ for $d = 0$ and $d = 1$, respectively. This particular computation ensures that each row^d index corresponds to a pseudorandom value $\in [0, R-1]$ that is only learned after all columns of the previously visited row are duplexed. Given the wide range of possibilities, those rows are unlike to be in cache; however, since they are visited sequentially, their columns can be prefetched by the processor to speed-up their processing.
- $prev^d$ $(d = 0, 1)$: set in line 36 to the indices of the most recently modified rows. Just like in the Setup phase, these rows are likely to still be in cache. Taking advantage of this fact, the visitation of its rows are not sequential but actually controlled by the pseudorandom, password-dependent variables $(col^0, col^1) \in [0, C-1]$. Each col^d $(d = 0, 1)$ is computed from the last and previous to last word of the sponge's outer state (i.e., from $rand[b/W-1]$ and $rand[b/W-2]$, respectively), either at the very beginning of the Wandering phase (line 30) or right after each duplexing operation (line **??**). As a result, the corresponding column indices cannot be determined prior to each duplexing, forcing all the columns to remain in memory for the whole duplexing operation and thwarting the construction of simple pipelines for their visitation.

The treatment given to the sponge's outputs is then quite similar to that in the Setup phase: the outputs provided by the sponge are sequentially XORed with $M[row^0]$ and, after being rotated, with $M[row^1]$ (line 33). However, in the Wandering phase the sponge's output is XORed with $M[row^0]$ from the lowest to the highest index, just like $M[row^1]$.

4.1.4 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 40) of a single, pseudorandly selected cell of the memory matrix, $M[row^0][col^0]$. The goal of this final call to absorb is mainly 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 several calls to the reduced-round f since the full-round absorb in line 3. This absorb operation is then followed by a full-round squeezing operation (line 41) for generating k bits, once again without resetting sponge's internal state to zeros. 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 the outputs are unpredictable. After all, violating such basic properties of Lyra 2 is equivalent to violate the same basic properties of the underlying sponge.

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 a given input c_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 15–19 (Setup phase) or in lines 29–35 (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 relation 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 inputs from consecutive iterations, c_i and c_{i+1}, the attacker needs to determine the sponge's internal state s_{i+1} that will result from duplexing c_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 the result of duplexing several inputs, 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 $b \cdot R \cdot C$ bits, where 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 execution.

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: $\lceil(|pwd| + |salt| + |params|)/b\rceil + R \cdot C \cdot \rho/\rho_{max}$ 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 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 [13], does not seem to be the best alternative for this purpose. This happens because Keccak excels in hardware rather than in software performance [31]. 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 as the sponge's f transformation. One example is Blake2b [5], a slightly tweaked version of Blake [7]. Blake itself displays a security level similar to that of Keccak [16], and its compression function has been shown to be a good permutation [6, 43] and to have a strong diffusion capability [7], while Blake2b is believed to retain most of these security properties [33].

The main (albeit minor) issue with Blake2b's permutation is that, to avoid fixed points, its internal state must be initialized with a 512-bit initialization vector (IV) rather than with a string of zeros as prescribed by the sponge construction. Therefore, the same IV should also be used for initializing the sponge's state in Lyra2. In addition, to prevent the IV from being overwritten by user-defined data, the sponge's capacity c employed when absorbing the user's input (line 3 of Algorithm 2) should have at least 512 bits, leaving up to 512 bits for the bitrate b. After this first absorb operation, though, the bitrate may be raised for increasing the overall throughput of Lyra2 if so desired.

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 3 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 [51], 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 $R \cdot C$. 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 $b \cdot C$ 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 Setup or Wandering phase's Columns Loops. 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 3 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 [34, 67].

5 Security analysis

Lyra2's design is such that (1) the derived key is both non-invertible and collision resistant, which is due to the initial and final full hashing operations, combined with reduced-round hashing operations in the middle of the algorithm; (2) attackers are unable to parallelize Algorithm 2 using multiple instances of the cryptographic sponge H, so they cannot significantly speed up the process of testing a password by means of multiple processing cores; (3) once initialized, the memory matrix is expected 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 algorithm. 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 potentially raising the overall throughput of the process. There are basically two methods for accomplishing this. The first is what we call a *Low-Memory attack*, which consists of trading memory for processing time, i.e., discarding (parts of) the matrix and recomputing the discarded information from scratch, when (and only when) it becomes necessary. 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, as well as 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. Consequently, the algorithm 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* [29], 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). The reason is that its sequential memory hard design is mainly intended to provide protection against this particular attack venue.

As a direct consequence of scrypt's memory hard design, we can formulate Theorem 1:

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 three previously computed rows for its computation.

For example, after using $M[0]$ and $M[1]$ for computing $M[2]$, both are once again employed in the computation of $M[3]$, 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 $M[3]$ 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 values of $M[1]$ and $M[2]$ if they are still in memory, taking advantage of the fact that the value of $rand[col]$ that modified $M[0][col]$ can be obtained as $M[1][col] \oplus M[2][C-1-col]$. Whichever the case, 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 expressive 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 for 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 [19, 35, 26] as done in [29, 27]) would be even better, but doing so, possibly building on the discussion hereby presented, remains as a matter for future work.

5.1.1 Preliminaries

For conciseness, along the discussion we denote by *CL* the Columns Loop of the Setup phase (lines 15—19 of Algorithm 2) and of the Wandering phase (lines 29—35). In this manner, ignoring the cost of XORing, reads/writes and other ancillary operations, *CL* corresponds approximately to $C \cdot \rho/\rho_{max}$ executions of f, a cost that is denoted simply as σ.

We denote by $s^0_{i,j}$ the state of the sponge right before $M[i][j]$ is initialized in the Setup phase. For $i \geqslant 2$, this corresponds to the state in line 15 of Algorithm 2. For conciseness, though, we often omit the "j" subscript, using s^0_i as a shorthand for $s^0_{i,0}$ whenever we the focus of the discussion are entire rows rather than their cells. We also employ a similar notation for the Wandering phase, denoting by s^τ_i 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 27 of Algorithm 2). Analogously, the i-th row ($0 \leqslant i < R$) output by the sponge during the Setup phase is denoted r^0_i, while r^τ_i denotes the output of the τ-th iteration of the Wandering phase's Time Loop.

Aiming to keep track of modifications made on rows of the memory matrix, we recursively use the subscript notation $M[X_{Y-Z-\ldots}]$ to denote a row X modified when it received the same values of $rand$ as row Y, then again when the row receiving the sponge's output was Z, and so on. For example, $M[0_2]$ corresponds to row $M[0]$ after its cells are XORed with $rotRt(rand)$ in the very first iteration of the Setup phase's Filling Loop. Finally, for conciseness, we write V^τ_1 and V^τ_2 to denote, respectively, the first and second half of: the Setup phase, for $\tau = 0$; or the Visitation Loop during the τ-th iteration of the Wandering phase's Time Loop, for $\tau \geqslant 1$.

5.1.2 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 4, which shows the corresponding visitation order of such rows and also their modifications due to these visitations.

5.1.2.1 Storing only what is needed: $1/2$ memory usage. Suppose that the attacker does not want to store all rows of the memory matrix during the algorithm's execution. One interesting approach for doing so is to keep in buffer only what will be required in future iterations of the Filling Loop, discarding rows that will not be used anymore. Since the Setup phase is purely deterministic, doing so is quite easy and, as long as the proper rows are kept, it incurs no processing penalty. This approach is illustrated in Figure 5 for our example scenario.

As shown in this figure, this simple strategy allows the execution of the Setup phase with a memory usage of $R/2+1$ rows, approximately half of the amount usually required. This observation

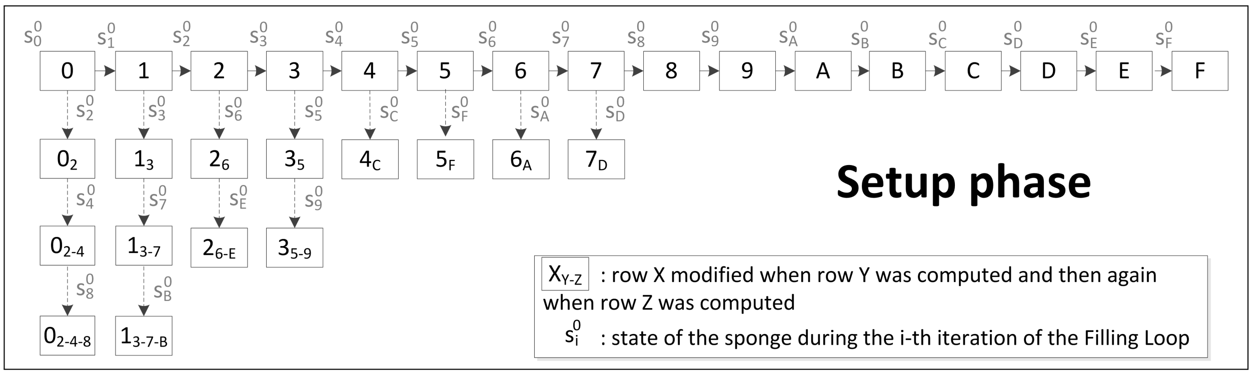
Setup phase
X_{Y-Z} : row X modified when row Y was computed and then again when row Z was computed
s^0_i : state of the sponge during the i-th iteration of the Filling Loop

Figure 4: *The Setup phase.*

comes from the fact that each half of the Setup phase requires all rows from the previous half and two extra rows (those more recently initialized/updated) to proceed. More precisely, $R/2 + 1$ corresponds to the peak memory utilization reached around the middle of the Setup phase, since (1) until then, part of the memory matrix has not been initialized yet and (2) rows initialized near the end of the Setup phase are only required for computing the next row and, thus, can be overwritten right after their cells are used. Even with this reduced 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 approximately $(R/2) \cdot C \cdot b$ 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 in this section. □

5.1.2.2 Storing less than what is needed: 1/4 memory usage.

If the attacker considers that storing half of the memory matrix is too much, he/she may decide to discard additional rows,

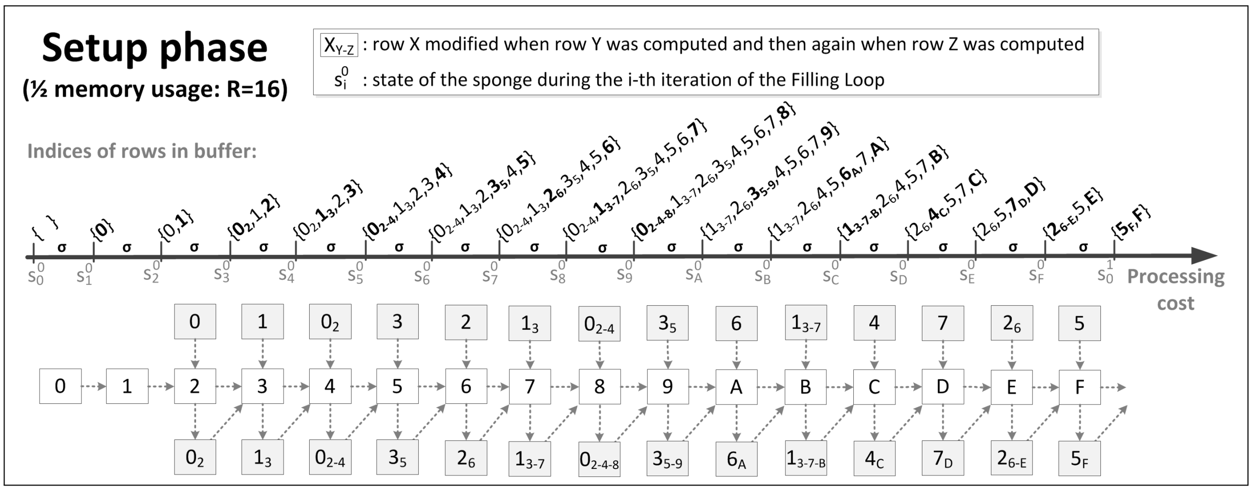
Setup phase
(½ memory usage: R=16)
X_{Y-Z} : row X modified when row Y was computed and then again when row Z was computed
s^0_i : state of the sponge during the i-th iteration of the Filling Loop
Indices of rows in buffer:
Processing cost

Figure 5: *Attacking the Setup phase: storing 1/2 of all rows. The most recently modified rows in each iteration are marked in bold.*

recomputing them from scratch only when they are needed. In that case, a reasonable approach is to discard rows that (1) will take longer to be used, either directly or for the recomputation of other rows, or (2) that can be easily computed from rows already available, so the impact of discarding them is low. The reasoning behind this strategy is that it allows the Setup phase to proceed smoothly for as long as possible. Therefore, as rows that are not too useful for the time being (or even not required at all anymore) are discarded from the buffer, the space saved in this manner can be diverted to the recomputation process, accelerating it.

The suggested approach is illustrated in Figure 6. As shown in this figure, at any moment we keep in memory only $R/4 = 4$ rows of the memory matrix besides the two most recently modified/updated, approximately half of what is used in the attack described in Section 5.1.2.1. This allows roughly 3/4 the Setup phase to run without any recomputation, since after that $M[4]$ is required to compute row $M[C]$. One simple way of doing so is to keep in memory the two most recently modified rows, $M[1_{3-7-B}]$ and $M[B]$, and then run the first half of the Setup phase once again with $R/4 + 2$ rows. This strategy should allow the recomputation not only of $M[4]$, but of all the $R/4$ rows previously discarded but still needed for the last 1/4 of the Setup phase (in our example, $\{M[4], M[7], M[2_6], M[5]\}$, as shown at the bottom of Figure 6). The resulting processing overhead would, thus, be approximately $(R/2)\sigma$, leading to a total cost of $(3R/2)\sigma$ for the whole Setup.

Obviously, there may be other ways of recomputing the required rows. For example, there is no need to discard $M[7]$ after $M[8]$ is computed, since keeping it in the buffer after that point would still respect the $R/4 + 2$ memory cost. Then, the recomputation procedure could stop after the recomputation of $M[2_6]$, reducing its cost in 1σ. Alternatively, $M[4]$ could have been kept in memory after the computation of $M[7]$, allowing the recomputations to be postponed by one iteration. However, then $M[7]$ could not be maintained as mentioned above and there would be not reduction in the attack's total cost. All in all, these and other tricks are not expected to reduce the total recomputation overhead significantly below $(R/2)\sigma$. This happens because the last 1/4 of the Setup phase is designed in such a manner that the row^1 index covers the entire first half of the memory matrix, including values near 0 and $R/2$. As a result, the recomputation of all values of $M[row^1]$ input to the sponge near the end of the Setup phase is likely to require most (if not all) of its first half to be executed.

These observations can be summarized in 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, $MemSetup(R) = R \cdot C \cdot b$ bits and $CostSetup(R) = R \cdot \sigma$, its execution with a memory cost of approximately $MemSetup(R)/4$ should raise its processing cost to approximately $3CostSetup(R)/2$.*

5.1.2.3 Storing less than what is needed: 1/8 memory usage. We can build on the previous analysis to estimate the performance penalty incurred when reducing the algorithm's memory usage by another half. Namely, imagine that Figure 6 represents the first half of the Setup phase (denoted V_1^0) for $R = 32$, in an attack involving a memory usage of $R/8 = 4$. In this case, recomputations are needed after approximately 3/8 of the Setup phase is executed. However, these are not the only recomputations that will occur, as the entire second half of the memory matrix (i.e., $R/2$ rows) still needs to be initialized during the second half of the Setup phase (denoted V_2^0). Therefore,

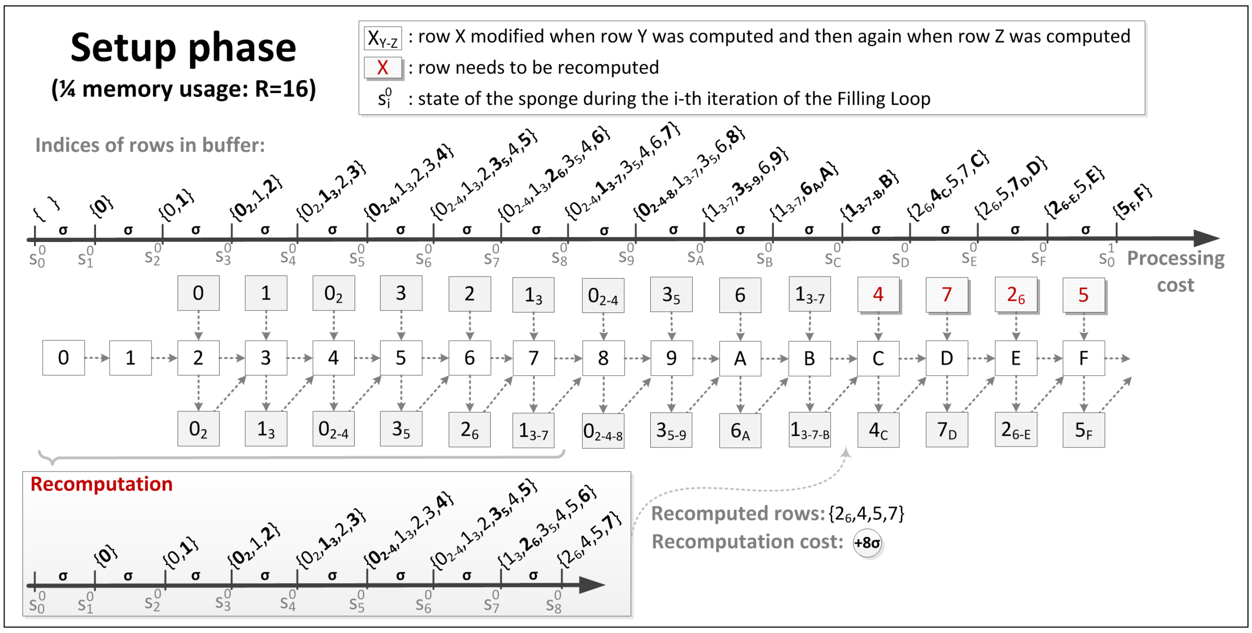
Setup phase
(¼ memory usage: R=16)
X_{Y-Z} : row X modified when row Y was computed and then again when row Z was computed
X : row needs to be recomputed
s_i^0 : state of the sponge during the i-th iteration of the Filling Loop
Indices of rows in buffer:
Processing cost
Recomputation
Recomputed rows: {2_6,4,5,7}
Recomputation cost: +8σ

Figure 6: *Attacking the Setup phase: storing 1/4 of all rows. The most recently modified rows in each iteration are marked in bold.*

the $R/2$ rows initialized/modified during V_1^0 will be once again required. Now suppose that the $R/8$ memory budget is employed in the recomputation of the required rows from scratch, running V_1^0 again whenever a group of previously discarded rows is needed. Since a total of $R/2$ rows need recomputation, the goal is to recover each of the $(R/2)/(R/8) = 4$ groups of $R/8$ rows in the sequence they are required during V_2^0, similarly to what was done a single time when the memory committed to the attack was $R/4$ rows (section 5.1.2.2). In our example, the four groups of rows required are (see Table 2): $g_1 = \{M[0_{2-4-8}], M[9], M[2_{6-E}], M[B]\}$, $g_2 = \{M[4_C], M[D], M[6_A], M[F]\}$, $g_3 = \{M[8], M[1_{3-7-B}], M[A], M[3_{5-9}]\}$, and $g_4 = \{M[C], M[5_F], M[E], M[7_D]\}$, in this sequence.

To analyze the cost of this strategy, assume initially that the memory budget of $R/8$ is enough to recover each of these groups by means of a single (partial or full) execution of V_1^0. First, notice that the computation of each group from scratch involves a cost of at least $(R/4)\sigma$, since the rows required by V_2^0 have all been initialized or modified after the execution of 50% of V_1^0. Therefore, the lowest cost for recovering any group is $(3R/8)\sigma$, which happens when that group involves only rows initialized/modified before $M[R/4 + R/8]$ (this is the case of g_3 in our example). A full execution of V_1^0, on the other hand, can be obtained from Conjecture 1: the buffer size is $MemSetup(R/2)/4 = R/8$ rows, which means that the processing cost is now $3CostSetup(R/2)/2 = (3R/4)\sigma$ (in our example, full executions are required for g_2 and g_4, due to rows $M[F]$ and $M[5_F]$). From these observations, we can estimate the four re-executions of V_1^0 to cost between $4(3R/8)\sigma$ and $4(3R/4)\sigma$, leading to an arithmetic mean of $(9R/4)\sigma$. Considering that a full execution of V_1^0 occurs once before V_2^0 is reached, and that V_2^0 itself involves a cost of $(R/2)\sigma$ even without taking the above overhead into account, the base cost of the Setup phase is $(3R/4 + R/2)\sigma$. With the overhead of $(9R/4)\sigma$ incurred by the re-executions of V_1^0, the cost of the whole Setup phase becomes then $(7R/2)\sigma$.

We emphasize, however, that this should be seen a coarse estimate, since it considers four (roughly complementary) factors described in what follows.

1. The one-to-one proportion between a full and a partial execution of V_1^0 when initializing rows

of V_2^0 is not tight. Hence, estimating costs with the arithmetic mean as done above may not be strictly correct. For example, going back to our scenario with $R = 32$ and a $R/8$ memory usage, the only group whose rows are all initialized/modified before $M[R/2 - R/8] = M[C]$ is g_3. Therefore, this is the only group that can be computed by running the part of V_1^0 that does not require internal recomputations. Consequently, the average processing cost of recomputing those groups during V_2^0 should be higher.

2. As discussed in section 5.1.2.2, the attacker does not necessarily need to always compute everything from scratch. After all, the committed memory budget can be used to bufferize a few rows from V_1^0, avoiding the need of recomputing them. Going back to our example with $R = 32$ and $R/8$ rows, if $M[2_{6-E}]$ remains available in memory when V_2^0 starts, g_1 can be recovered by running V_1^0 once, until $M[B]$ is computed, which involves no internal recomputations. This might reduce the average processing cost of recomputations, possibly compensating the extra cost incurred by factor 1.

3. The assumption that each of the four executions of V_1^0 can recover an entire group with the costs hereby estimated is not always realistic. The reason is that the costs of V_1^0 as described in section 5.1.2.2 are attained when what is kept in memory is only the set of rows strictly required during V_1^0. In comparison, in this attack scenario we need to run V_1^0 while keeping rows that were originally discarded, but now need to remain in the buffer because they are used in V_2^0. In our example, this happens with $M[6_A]$, the third row from g_2: to run V_1^0 with a cost of $(3R/4)\sigma$, $M[6_A]$ should be discarded soon after being modified (namely, after the computation of $M[B]$), thus making room for rows $\{M[4], M[7], M[2_6], M[5]\}$. Otherwise, $M[4_C]$ and $M[D]$ cannot be computed while respecting the $R/8 = 4$ memory limitation. Notice that discarding $M[6_A]$ would not be necessary if it could be consumed in V_2^0 before $M[4_C]$ and $M[D]$, but this is not the case in this attack scenario. Therefore, to respect the $R/8 = 4$ memory limitation while computing g_2, in principle the attacker would have to run V_1^0 twice: the first to obtain $M[4_C]$ and $M[D]$, which are promptly used in V_2^0, as well as $M[F]$, which remains in memory; and the second for computing $M[6_A]$ while maintaining $M[F]$ in memory so it can be consumed in V_2^0 right after $M[6_A]$. This strategy, illustrated in Figure 7, introduces an extra overhead of 11σ to the attack in our example scenario.

4. Finally, there is no need of computing an entire group of rows from V_1^0 before using those rows in V_2^0. For example, suppose that $M[0_{2-4-8}]$ and $M[9]$ are consumed by V_2^0 as soon as they are computed in the first re-execution of V_2^0. These rows can then be discarded and the attacker can use the extra space to build $g_1' = \{M[2_{6-E}], M[B], M[4_C], M[D]\}$ with a single run of V_1^0. This approach should reduce the number of re-executions of V_1^0 and possibly alleviate the overhead from factor 3.

5.1.2.4 Storing less than what is needed: generalization. We can generalize the discussion from section 5.1.2.4 to estimate the processing costs resulting from recursively reducing the Setup phase's memory usage by half. This can be done by imagining that any scenario with a $R/2^{n+2}$ ($n \geqslant 0$) memory usage corresponds to V_1^0 during an attack involving half that memory. Then, representing by $CostSetup_n(m)$ the number of times CL is executed in each window containing m

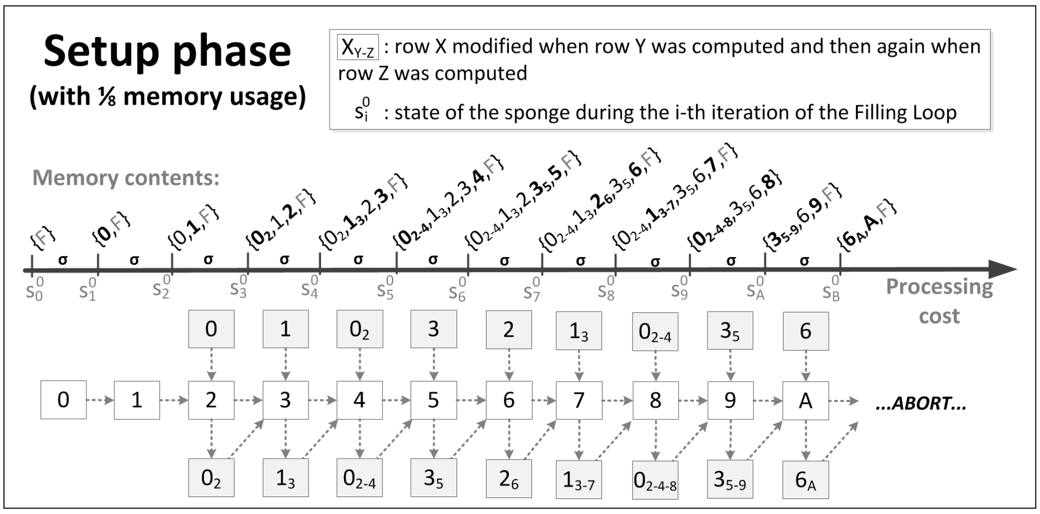
Setup phase
(with ⅛ memory usage)
X_{Y-Z} : row X modified when row Y was computed and then again when row Z was computed
s_i^0 : state of the sponge during the i-th iteration of the Filling Loop
Memory contents:
Processing cost
...ABORT...

Figure 7: *Attacking the Setup phase: recomputing $M[6_A]$ while storing 1/8 of all rows and keeping $M[F]$ in memory. The most recently modified rows in each iteration are marked in bold.*

rows (seen as V_1^0 by the subsequent window) and following the same assumptions and simplifications of section 5.1.2.3, we can write the following recursive equation:

$$
\begin{aligned}
CostSetup_0(m) &= 3m/2 \qquad \triangleright\ 1/4 \text{ memory usage scenario } (n=0)\\
CostSetup_n(m) &= \overbrace{CostSetup_{n-1}(m/2)}^{V_1^0} + \overbrace{m/2}^{V_2^0} + \overbrace{\underbrace{(3 \cdot CostSetup_{n-1}(m/2)/4)}_{\text{approximate cost of each execution}} \cdot \underbrace{(2^{n+1})}_{\text{number of executions}}}^{\text{Re-executions of } V_1^0}
\end{aligned}
\tag{1}
$$

For example, for $n = 2$ (and, thus, a memory usage of $R/16$), we have:

$$
\begin{aligned}
CostSetup_2(R) &= CostSetup_1(R/2) + R/2 + (3 \cdot CostSetup_1(R/2)/4) \cdot (2^{2+1})\\
&= 7CostSetup_1(R/2) + R/2\\
&= 7(CostSetup_0(R/4) + R/4 + (3 \cdot CostSetup_0(R/4)/4) \cdot (2^{1+1})) + R/2\\
&= 7(3R/8 + R/4 + (3 \cdot (3R/8)/4) \cdot 4) + R/2\\
&= 51R/4
\end{aligned}
$$

In Equation 1, we assume that the cost of each re-execution of V_1^0 can be approximated to 3/4 of its total cost. We argue that this is a reasonable approximation because, as discussed in section 5.1.2.3, between 50% and 100% of V_1^0 needs to be executed when recovering each of the $(R/2)/(R/2^{n+2}) = 2^{n+1}$ groups of $R/2^{n+2}$ rows required by V_2^0.

The fact that Equation 1 assumes that only 2^{n+1} re-executions of V_1^0 are required, on the other hand, is likely to become an oversimplification as R and n grow. The reason is that factor 4 discussed in section 5.1.2.3 is unlikely to compensate factor 3 in these cases. After all, as the memory available drops, it should become harder for the attacker to spare some space for rows that are not immediately needed. The theoretical upper limit for the number of times V_1^0 would have to be executed during V_2^0 when the memory usage is m would then be $m/4$: this corresponds to a hypothetical scenario in which, unless promptly consumed, no row required by V_2^0 remains in the buffer during V_1^0; then, since V_2^0 revisits rows from V_1^0 in an alternating pattern, approximately a pair of rows can be recovered with each execution of V_1^0, as the next row required is likely to have already been computed and discarded in that same execution.

The recursive equation for estimating this upper limit would then be (in number of executions of CL):

$$\begin{aligned} CostSetup_0(m) &= 3m/2 \qquad \triangleright 1/4 \text{ memory usage scenario } (n=0) \\ CostSetup_n(m) &= \overbrace{CostSetup_{n-1}(m/2)}^{V_1^0} + \overbrace{m/2}^{V_2^0} + \overbrace{\underbrace{(3 \cdot CostSetup_{n-1}(m/2)/4)}_{\text{approximate cost of each execution}} \cdot \underbrace{(m/4)}_{\text{number of executions}}}^{\text{Re-executions of } V_1^0} \end{aligned} \tag{2}$$

The upper limit for a memory usage of $R/16$ could then be computed as:

$$\begin{aligned} CostSetup_2(R) &= CostSetup_1(R/2) + R/2 + (3 \cdot CostSetup_1(R/2)/4) \cdot (R/4) \\ &= (1 + 3R/16)CostSetup_1(R/2) + R/2 \\ &= (1 + 3R/16)(CostSetup_0(R/4) + R/4 + (3 \cdot CostSetup_0(R/4)/4) \cdot (R/8)) + R/2 \\ &= (1 + 3R/16)(3R/8 + R/4 + (3 \cdot (3R/8)/4) \cdot (R/8)) + R/2 \\ &= 18(R/16) + 39(R/16)^2 + (3R/16)^3 \end{aligned}$$

Even though this upper limit is mostly theoretical, we do expect the R^{n+1} component resulting from Equation 2 to become expressive and dominate the running time of Lyra2's Setup phase as n grows and the memory usage drops much below $R/2^8$ (i.e., for $n \gg 1$). In summary, these observations can be formalized in the following Conjecture:

Conjecture 2. *Consider that Lyra2 operates with parameters T, R and C. Whilst the regular memory and processing costs of its Setup phase's are, respectively, $MemSetup = R \cdot C \cdot b$ bits and $CostSetup = R \cdot \sigma$, running it with a memory cost of approximately $MemSetup/2^{n+2}$ leads to a processing cost $CostSetup_n(R)$ given by recursive Equations 1 (for a lower bound) and 2 (for an upper bound).*

5.1.2.5 Storing only intermediate sponge states.

Besides the strategies mentioned in the previous sections, and possibly complementing them, one can try to explore the fact that the sponge states are usually smaller than a row's cells for saving memory: while rows have $b \cdot C$ bits, a state is up to C times smaller, taking $w = b + c$ bits. More precisely, by storing all sponge states, one can recompute any *cell* of a given row whenever it is required, rather than computing the entire row at once. For example, the initialization of each cell of $M[2]$ requires only one cell from $M[0]$ and one from $M[1]$. Similarly, initializing a cell of $M[4]$ takes one cell from $M[0]$ and one from $M[2]$ (both required for the computation of $M[0_2]$), as well as one from $M[1]$ and up to two from $M[3]$ (one because $M[3]$ is itself fed to the sponge and another required to the computation of $M[1_3]$).

This attack would be easy to build if the cells sequentially output by the sponge during the

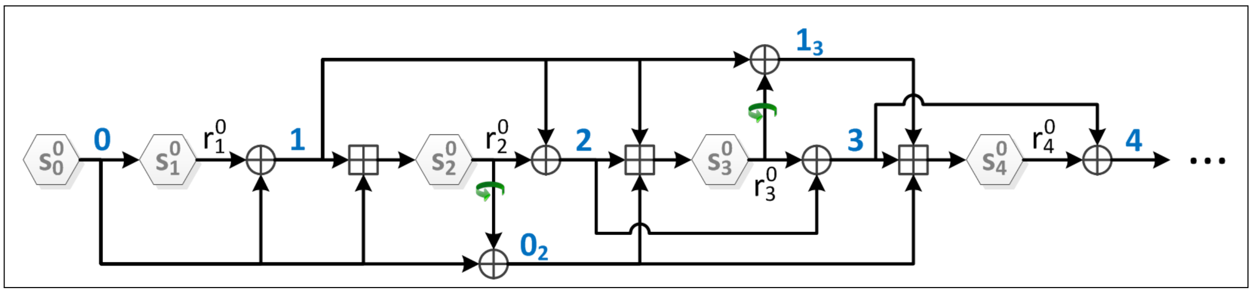

Figure 8: *Attacking the Setup phase: storing only sponge states.*

initialization of $M[i]$ could be sequentially employed as input in the initialization of $M[j > i]$ Indeed, in that hypothetical case, one could build a circuitry like the one illustrated in Figure 8 to compute cells as they are required. For example, computing $M[2][0]$ in this scenario would require (1) states $s^0_{0,0}$, $s^0_{1,0}$ and $s^0_{2,0}$, and (2) two b-long buffers, one for $M[0][0]$ and another for $M[1][0]$, which are used as inputs for the sponge in state $s^0_{2,0}$.

After that, the same buffers could be reused for storing $M[0][1]$ and $M[1][1]$ when computing $M[2][1]$, using the same sponge instances that are now in states $s^0_{0,1}$, $s^0_{1,1}$ and $s^0_{2,1}$. This process could then be iteratively repeated until the computation of $M[2][C-1]$. At that point, we would have the value of $s^0_{3,0}$ and could apply an analogous strategy for computing $M[3]$. The total processing cost of computing $M[2]$ would then be 3σ, since it would involve one complete execution of CL for each of the sponge instances initially in states $s^0_{0,0}$, $s^0_{1,0}$ and $s^0_{2,0}$. As another example, the computation of $M[4][col]$ could be performed in a similar manner, with states $s^0_{0,0}$ — $s^0_{4,0}$ and buffers for $M[0][col]$, $M[1][col]$ and $M[3][col]$ (used as inputs for the sponge in state $s^0_{4,0}$), as well as for $M[2][col]$ (required in the computation of $M[3][col]$ and $M[0_2]$); the total processing cost would then be 5σ.

Generalizing this strategy, any $M[row]$ could be processed using only row buffers and $row + 1$ sponge instances in different states, leading to a cost of $row \cdot \sigma$ for its computation. Therefore, for the whole Setup phase, the total processing cost would be around $(R^2/2)\sigma$ using approximately $2/C$ of the memory required in a regular execution of Lyra2.

Even though this attack venue may appear promising at first sight for a large C/R ratio, it cannot be performed as easily as described in the above theoretical scenario. This happens because Lyra2 reverses the order in which a row's cells are written and read, as illustrated in Figure 9. Therefore, the order in which the cells from any $M[i]$ are picked to be used as input during the initialization of $M[j > i]$ is the opposite of the order in which they are output by the sponge. Considering this constraint, suppose we want to sequentially recompute $M[1][0]$ through $M[1][C-1]$, as well as $M[0][0]$ through $M[0][C-1]$, as required (in that order) for the initialization of $M[2][C-1]$ through $M[2][0]$ during the first iteration of the Filling Loop.

From the start, we have a problem: since $M[1][0] = M[0][C-1] \oplus H_\rho.duplex(M[0][C-1], b)$, its recomputation requires $M[0][C-1]$ and $s^0_{1,C-1}$. Consequently, computing $M[2][C-1]$ as in our hypothetical scenario would involve roughly 1σ to compute $M[0][0]$ from $s^0_{0,0}$, as well as 1σ (and several inputs) for computing $s^0_{1,C-1}$ from $s^0_{1,0}$. A similar issue would occur right after that, when initializing $M[2][C-2]$ from $M[0][1]$ and $M[1][1]$: unless inverting the sponge's (reduced-round) internal permutation is itself easy, $M[0][1]$ cannot be easily obtained from $M[0][0]$, and neither the sponge state $s^0_{1,C-2}$ (required for recomputing $M[1][1]$) from $s^0_{1,C-1}$.

On the other hand, recomputing $M[0][1]$ and $s^0_{1,C-2}$ from the values of $s^0_{0,1}$ and $s^0_{1,1}$ resulting

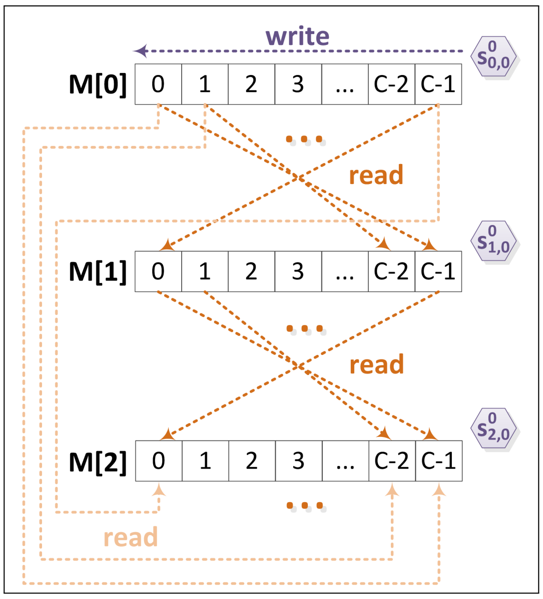

Figure 9: *Reading and writing cells in the Setup phase.*

from the previous step would involve a processing cost of approximately $(C-2)\sigma/C$. If we repeat this strategy for all cells of $M[2]$, the total processing cost of initializing this row should be on the order of C times higher the "1σ" obtained in our hypothetical scenario. Since the conditions for this C multiplication factor appear in the computation of any other row, the processing time of this attack venue against Lyra2 is expected to become $C(R^2/2)\sigma$ rather than simply $(R^2/2)\sigma$, counterbalancing the memory reduction lower than $1/C$ potentially obtained.

Obviously, one could store additional sponge states aiming for a lower processing time. For example, by storing the sponge state $s^0_{i,C/2}$ in addition to $s^0_{i,0}$, the attack's processing costs may be reducible by half. However, the memory cuts obtained with this approach diminish as the number of intermediate sponge states stored grow, eventually defeating the whole purpose of the attack. All things considered, even if feasible, this attack venue does not seem much more advantageous than the approaches discussed in the previous sections.

5.1.3 Adding the Wandering phase

During each iteration of the Wandering phase, the rows modified in the previous iteration are input to the sponge together with two other (pseudorandomly picked) rows. The latter two rows are then XORed with the sponge's output and the result is fed to the sponge in the subsequent iteration. To analyze the effects of this phase, it is useful to consider an "average", slightly simplified scenario like the one depicted in Figure 10, in which all rows are modified only once during each half of the Visitation Loop, i.e., during V^1_1 the sets formed by the values assumed by row^0 and by row^1 are disjoint. We then apply the same principle to V^1_2, modifying each row only once more in a different (arbitrary) pseudorandom order. We argue that this is a reasonable simplification, given the fact that the indices of the picked rows form an uniform distribution. In addition, we argue that this is actually beneficial for the attacker, since any row required during V^1_1 can be obtained simply by running the Setup phase once again, instead of involving recomputations of the Wandering phase itself. We also note that, in the particular case of Figure 10, we make the visitation order in V^1_1 be the exact opposite of the initialization/update of rows during V^0_2, while in V^1_2 the order is the

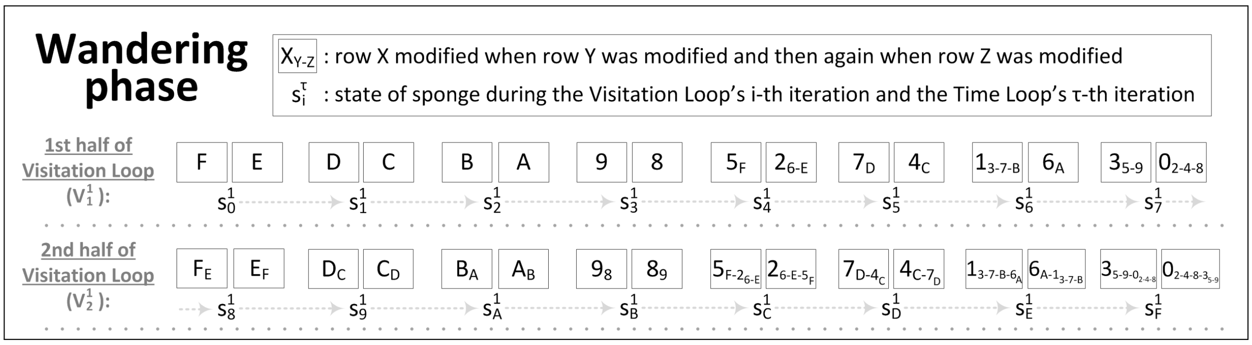
Wandering phase
$X_{Y\text{-}Z}$: row X modified when row Y was modified and then again when row Z was modified
s_i^τ : state of sponge during the Visitation Loop's i-th iteration and the Time Loop's τ-th iteration
1st half of Visitation Loop (V_1^1):
F E D C B A 9 8 5_F $2_{6\text{-}E}$ 7_D 4_C $1_{3\text{-}7\text{-}B}$ 6_A $3_{5\text{-}9}$ $0_{2\text{-}4\text{-}8}$
s_0^1 s_1^1 s_2^1 s_3^1 s_4^1 s_5^1 s_6^1 s_7^1
2nd half of Visitation Loop (V_2^1):
F_E E_F D_C C_D B_A A_B 9_8 8_9 $5_{F\text{-}2_{6\text{-}E}}$ $2_{6\text{-}E\text{-}5_F}$ $7_{D\text{-}4_C}$ $4_{C\text{-}7_D}$ $1_{3\text{-}7\text{-}B\text{-}6_A}$ $6_{A\text{-}1_{3\text{-}7\text{-}B}}$ $3_{5\text{-}9\text{-}0_{2\text{-}4\text{-}8}}$ $0_{2\text{-}4\text{-}8\text{-}3_{5\text{-}9}}$
s_8^1 s_9^1 s_A^1 s_B^1 s_C^1 s_D^1 s_E^1 s_F^1

Figure 10: *An example of the Wandering phase's execution.*

same as in V_1^1, for the sake of illustrating worst and best case scenarios (respectively).

In this scenario, the $R/2$ iterations of V_1^1 cover the entire memory matrix. The relationship between V_1^1 and V_2^0 is, thus, very similar to that between V_2^0 and V_1^0: if any row initialized/modified during V_2^0 is not available when it is required by V_1^1, then it is probable that the Setup phase will have to be (partially) run once again, until the point the attacker is able to recover that row. However, unlike the Setup phase, the probabilistic nature of the Wandering phase prevents the attacker from predicting which rows from V_1^1 can be safely discarded, which is deemed to raise the average number of re-executions of V_1^1. Consequently, we can adapt the arguments employed in section 5.1.2 to estimate the cost of low-memory attacks when the execution includes the Wandering phase, which is done in what follows for different values of T.

5.1.3.1 The first iteration of the Time Loop with 1/2 memory usage: Strategy 1. We start our analysis with an attack involving only $R/2$ rows and $T = 1$. Even though this memory usage would allow the attacker to run the whole Setup phase with no penalty (see section 5.1.2.1), the Wandering phase's Visitation Loop is not so lenient: in each iteration of V_1^1 there is only a 25% chance that row^0 and row^1 are both available in memory. Hence, 75% of the time the attacker will have to recompute at least one of the missing rows.

To minimize the cost of V_1^1 in this context, one possible strategy is to always keep in memory rows $M[i \geqslant 3R/4]$, using the remaining $R/4$ memory budget as a spare for recomputations. The reasoning behind this approach is that: (1) 3/4 of the Setup phase can be run with $R/4$ without internal recomputations (see section 5.1.2.2); (2) since rows $M[i \geqslant 3R/4]$ are already available, this execution gives the updated value of any row $\in [R/2, R[$ and of half of the rows $\in [0, R/2[$; and (3) by XORing pairs of rows $M[i \geqslant 3R/4]$ accordingly, the attacker can recover any $r^0_{i \geqslant 3R/4}$ output by the sponge and, then, use it to compute the updated value of any row $\in [0, R/2[$ from the values obtained from the first half of the Setup. In the scenario depicted by Figure 10, for example, $M[5_F]$ can be recovered by computing $M[5]$ and then making $M[5_F][col] = M[5][col] \oplus rotRt(r^0_F[col])$, where $r^0_F[col] = M[F][C - 1 - col] \oplus M[E][col]$.

With this approach, recomputing rows when necessary can take from $(R/4)\sigma$ to $(3R/4)\sigma$, or $(R/2)\sigma$ on average, if the Setup phase is executed just like shown in Section 5.1.2.1. It is not always necessary pay this cost for every iteration of V_1^1, however, if the needed row(s) can be recovered from those already in memory. For example, if during V_1^1 the rows are visited in the exact same order of their initialization/update in V_2^0, then each row can be used by V_1^1 before being discarded. A very lucky attacker could, thus, be able to run the entire V_1^1 by executing 3/4 of the Setup only once. Hence, the cost of each of the $R/2$ iterations of V_1^1 can be estimated as: 1 in 25% of these iterations, when row^0 and row^1 are both in memory; and roughly $(R/4)\sigma$ in 75% of its iterations, when one or a pair of rows need to be recovered. The total cost of V_1^1 becomes, thus, $((1/4) \cdot (R/2) + (3/4) \cdot (R/2) \cdot (R/4))\sigma \approx (3R^2/32)\sigma$.

After that, when V_2^1 is reached, 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 (partially) run the Setup phase once again to get their values. For example, in the scenario depicted in Figure 10, the rows required for iteration $i = 8$ of the Visitation Loop besides $M[prev^0] = M[A]$ and $M[prev^1] = 9$ are $M[8_{13-7-B}]$ and $M[B_{6_A}]$, both computed during V_1^1. Therefore, if these rows have not been kept in memory, V_1^1 will have to be (partially) run once again, which implies new runs of the Setup itself. The cost of these re-executions are likely to be lower than originally, though, because now the attacker can take advantage of the knowledge about which rows from V_2^0 are needed to compute each row from V_1^1. On the other hand, keeping $M[i \geqslant 3R/4]$ is unlikely to be much advantageous now, because that would reduce the attacker's ability to bufferize rows from V_1^1.

In this context, one possible approach is to keep in memory the sponge's state at the beginning of V_1^1 (i.e., s_0^1), as well as the corresponding value of $prev^0 \boxplus prev^1$ used as part of the sponge's input at this point (in our example, $M[F] \boxplus M[5_F]$). This allows the Setup and V_1^1 to run as different processes following a producer-consumer paradigm: the latter can proceed as long as the required inputs (rows) are provided by the former, the available memory budget being used to build their buffers. Using this strategy, the Setup needs to be run from 1 to 2 times during V_1^1. The first case refers to when each pair of rows provided by an iteration of V_2^0 can be consumed by V_1^1 right away, so they can be removed from the Setup's buffer similarly to what is done in Section 5.1.2.1. This happens if rows are revisited in V_1^1 in the same order lastly initialized/updated during V_2^0. The second extreme occurs when V_1^1 takes too long to start consuming rows from V_2^0, so some rows produced by the latter end up being discarded due to lack of space in the Setup's buffer. This happens, for example, if V_1^1 revisits rows indexed by row^0 during V_2^0 before those indexed by row^1, in the reverse order of their initialization/update, as is the case in Figure 10. Ignoring the fact that the Setup only starts providing useful rows for V_1^1 after half of its execution, on average we would have to run the Setup 1.5 times, these re-executions leading to an overhead of roughly $(3R/2)\sigma$.

As a result, we can estimate that recomputing any row from V_2^1 would require running 50% of V_1^1 on average. The cost of doing so would be $(R/4 + 3R/4)\sigma$, the first parcel of the sum corresponding to the cost of V_1^1 itself and the second to the overhead from the Setup's re-executions. As a side effect, this would also leave in V_1^1's buffer $R/2$ rows, which may reveal useful during the subsequent iteration of V_2^1. The average cost of the $R/2$ iterations of V_2^1 would then be: 1σ whenever both $M[row^0]$ and $M[row^1]$ are available, which happens in 25% of these iterations; roughly $R\sigma$ whenever $M[row^0]$ and/or $M[row^1]$ need to be recomputed, so for 75% of these iterations. This leads to a total cost of $(R/8 + 3R^2/8)\sigma$ for V_2^1. Consequently, adding up the cost of Setup, V_1^1 and V_2^1, the computation cost of Lyra2 when the memory usage is halved and $T = 1$ can be estimated as $R\sigma + (3R^2/32)\sigma + (R/8 + 3R^2/8)\sigma \approx (R^2/2)\sigma$ for this strategy.

5.1.3.2 The first iteration of the Time Loop with $1/2$ memory usage: Strategy 2. An alternative for the strategy described in Section 5.1.3.1 is to employ the following trick[1]: if we keep in memory all rows produced during V_1^0 and a few rows initialized during V_2^0 together with the corresponding sponge states, we can skip part of the latter's iterations when initializing/updating the rows required by V_1^1. In our example scenario, we would keep in memory rows $M[0_{2-4}]$—$M[7]$ as output by V_1^0. Then, by keeping rows $M[C]$ and $M[4_C]$ in memory together with state s_D^0, $M[D]$ and $M[7_D]$ can be recomputed directly from $M[7]$ with a cost of 1σ, while $M[F]$ and $M[5_F]$ can be recovered with a cost of 3σ. In both cases, $M[C]$ and $M[4_C]$ act as "sentinels" that allow us to skip the computation of $M[8]$—$M[C]$.

More generally, if we keep rows $M[0 \leqslant i < R/2]$, obtained by running V_1^0, as well as $\epsilon > 0$ sentinels equally distributed in the range $[R/2, R[$, the average cost of recovering any row output by V_2^0 becomes $R/4\epsilon$. The memory cost of such strategy is approximately $R/2 + 2\epsilon$. When compared with the approach described in Section 5.1.3.1, the main drawback is that only the 2ϵ sentinels rows can be promptly consumed by V_1^1, since rows provided by V_1^0 are overwritten during the execution of V_2^0. As a result, the average cost of V_1^1 ends up being approximately $(R/2) \cdot (R/4\epsilon)\sigma$ for a small ϵ.

We can employ a similar trick for the execution of V_2^1, by placing sentinels along the execution of V_1^1 to reduce the cost of its recomputations. However, with $\epsilon > 0$ sentinels, $R/4\epsilon$

along V_1^1 makes

the execution of V_2^1,

$M[D]$ can be computed with a cost of 1σ if we have $M[7]$ available.

any row initialized/updated during V_2^0 can be

which are those actually required by V_2^0, can be easily if those

with the rows produced by

in the range $M[R/2 \leqslant i < R]$ can be easily computed

running V_1^0 to compute $M[0 \leqslant i < R/2]$ and placing some "sentinels" along $M[R/2 \leqslant i < R]$, which would

then using those rows

, which are always kept in memory as basis for and are not updated, and placing

the Setup for the first

and using the resulting

the first $R/2$ iteration Setup

sponge states s_i^0 for $0 \leqslant i \leqslant 3R/4$ and the sponge's input .

xxx

(1) Sentinels at positions $8j$

, assuming that –3/4 of the Setup phase are executed for

Its total cost becomes, thus, $((R/8) \cdot 1 + (3R/8) \cdot (R/2))\sigma = (R/8 + 3R^2/16)\sigma$.

With these considerations, we can build a simplified scenario that ignores factors 1 and 3, thus allowing us to estimate a lower bound for the cost of such low-memory attack against Lyra2. First, observe that the operation of Lyra2 is such that the dependence between rows of the memory matrix is as illustrated in Figure **??**:

[1]This is analogous to the attack presented in [24] for the version of Lyra2 originally submitted to the Password Hashing Competition as "V1"

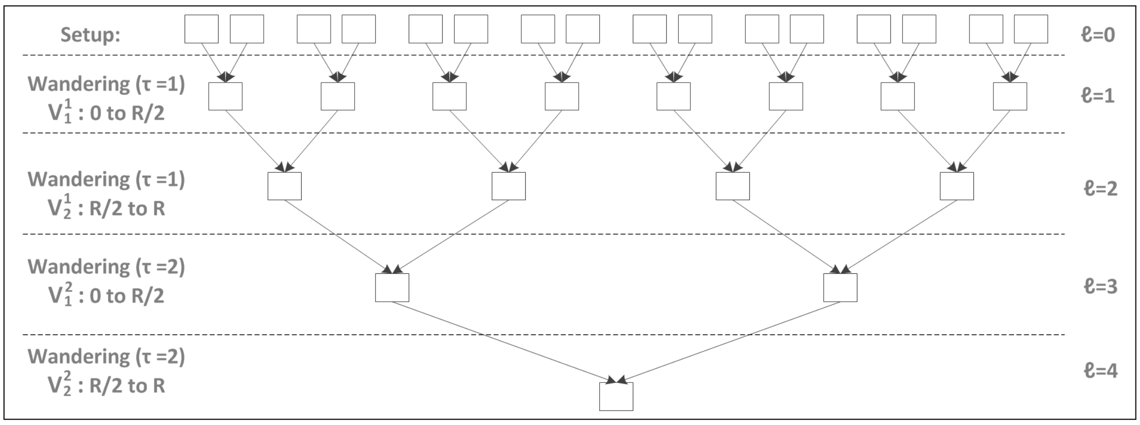

Figure 11: *Tree representing the dependences among rows in Lyra2.*

The attack is illustrated in Figure **??**, and described as follows[2].

More precisely, and as illustrated in Figure **??**, our simplification consists in assuming that each $M[i \geqslant R/2]$

xxx

in the 1/8 memory usage

is the one illustrated in

the scenario faced by the attacker

The strategy consists in running the Setup phase until we have $R/8$ rows available in memory. Then, and always keeping those rows after that (modificat

(2) keeping all states of the sponge $s^0_{i>R/8}$

More precisely, we assume that each row can be compu

xxx FINALIZAR COM ESTA FRASE (possivelmente elaborada melhor) Even though this attack is unrealistic due to the many underlying simplifications, it gives an lower bound estimate for low-memory attacks against Lyra2.

5.1.3.3 The whole Time Loop with 1/2 memory usage.

Generalizing the discussion for any iteration $\tau \geqslant 1$, the execution of V_1^τ (resp. V_2^τ) could use $V_2^{\tau-1}$ (resp. V_1^τ) similarly to what is done in section 5.1.3.1. Therefore, as Lyra2's execution progresses, it creates an inverted tree-like dependence graph like the one depicted in Figure 11, level $\ell = 0$ corresponding to the Setup phase and each half of the Visitation Loop raising the tree's depth by one. Since level $\ell > 0$ corresponds to $R/2$ iterations of the Visitation Loop, the full execution of that level requires roughly all rows modified in the previous level $(\ell - 1)$. With $R/2$ rows in memory, the original computation of any level ℓ can then be described by the following recursive equation (in number of executions of CL):

$$CostWander^*_\ell = \overbrace{\underbrace{(R/8)}_{\substack{25\%\text{ of}\\\text{iterations}}} \cdot 1}^{\substack{\text{no re-execution of}\\\text{previous levels}}} + \overbrace{\underbrace{(3R/8)}_{\substack{75\%\text{ of}\\\text{iterations}}} \cdot CostWander_{\ell-1}/2}^{\substack{\text{re-executions of}\\\text{previous levels}}} \tag{3}$$

The value of $CostWander_{\ell-1}$ in Equation 3 is lower than that of $CostWander^*_{\ell-1}$, however, since the former is purely deterministic. To estimate such cost, we can use the same strategy

[2]This is analogous to the attack presented in [24] for the version of Lyra2 originally submitted to the Password Hashing Competition as "V1", but displays higher costs even in this simplified scenario.

adopted in section 5.1.3.1: keeping the sponge's state at the beginning of each level ℓ and the corresponding value of $prev^0 \boxplus prev^1$, and then running level $\ell - 1$ 1.5 times on average to recover each row that needs to be consumed. For any level ℓ, the resulting cost can be described by the following recursive equation:

$$\begin{aligned} CostWander_0 &= R \qquad \triangleright \text{ The Setup phase itself} \\ CostWander_\ell &= \underbrace{R/2}_{\text{internal computations}} + \underbrace{(3/2) \cdot CostWander_{\ell-1}}_{\text{re-executions of previous level}} = R \cdot (2(3/2)^\ell - 1) \end{aligned} \tag{4}$$

Combining Equations 3 and 4 with Lemma 1, we get that the cost (in number of executions of CL) of running Lyra2 with half of the prescribed memory usage for a given T would be roughly:

$$\begin{aligned} CostLyra2_{1/2}(R,T) &= R + CostWander^*{}_1 + \cdots + CostWander^*{}_{2T} \\ &= (T+4) \cdot (R/4) + (3R^2/4) \cdot ((3/2)^{2T} - (T+2)/2) \\ &= \mathcal{O}((3/2)^{2T} R^2) \end{aligned} \tag{5}$$

5.1.3.4 The whole Time Loop with less than 1/2 memory usage. A memory usage of $1/2^{n+2}$ ($n \geqslant 0$) is expected to have three effects on the execution of the Wandering phase. First, the probability that row^0 and row^1 will both be available in memory at any iteration of the Visitation Loop drops to $1/2^{n+2}$, meaning that Equation 3 needs to be updated accordingly. Second, the cost of running the Setup phase is deemed to become higher, its lower and upper bounds being estimated by Equations 1 and 2, respectively. Third, level $\ell - 1$ may have to be re-executed 2^{n+2} times to allow the recovery of all rows required by level ℓ, which has repercussions on Equation 4: on average, $CostWander_\ell$ will involve $(1 + 2^{n+2})/2 \approx 2^{n+1}$ calls to $CostWander_{\ell-1}$.

Combining these observations, we arrive at

$$CostWander^*{}_{\ell,n} = \overbrace{\underbrace{(R/2) \cdot (1/2^{n+2})}_{1/2^{n+2} \text{ of iterations}} \cdot 1}^{\text{no re-execution of previous levels}} + \overbrace{\underbrace{(R/2) \cdot (1 - 1/2^{n+2})}_{\text{all other iterations}} \cdot (CostWander_{\ell-1,n})/2}^{\text{re-executions of previous levels}} \tag{6}$$

as an estimate for the original (probabilistic) executions of level ℓ, and at

$$\begin{aligned} CostWander_{0,n} &= CostSetup_n(R) \qquad \triangleright \text{ The Setup phase} \\ CostWander_{\ell,n} &= \overbrace{R/2}^{\text{internal computations}} + \overbrace{(2^{n+1}) CostWander_{\ell-1,n}}^{\text{re-executions of previous level}} \\ &= (R/2) \cdot (1 - (2^{n+1})^\ell)/(1 - 2^{n+1}) + (2^{n+1})^\ell \cdot CostSetup_n(R) \end{aligned} \tag{7}$$

for the deterministic re-executions of level ℓ.

Combined, Equations 6 and 7 lead to a total cost for Lyra2 of

$$\begin{aligned} CostLyra2_{1/2^{n+2}}(R,T) &= (CostSetup_n(R) + CostWander^*{}_{1,n} + \cdots + CostWander^*{}_{2T,n})\sigma \\ &\approx \mathcal{O}((R^2)(2^{2nT}) + R \cdot CostSetup_n(R) \cdot 2^{2nT}) \end{aligned} \tag{8}$$

Since Equation 2 suggests that $CostSetup_n = \mathcal{O}(R^{n+1})$ and this is likely to be a good estimate as n grows (see Section 5.1.2.4), we conjecture that the processing cost of Lyra2 using the discussed strategy will become $\mathcal{O}(2^{2nT} R^{n+2})$ for $n \gg 1$.

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 approach 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. This high bandwidth usage is also likely to hinder the construction of high-performance dedicated hardware for testing multiple password in parallel.

Another feature of Lyra2 is the fact that, during the Wandering phase, the columns of the most recently updated rows ($M[prev^0]$ and $M[prev^0]$) are read in a pseudorandom manner. Since these rows are expected to be in cache during a regular execution of Lyra2, a legitimate user that configures C adequately should be able to read these rows approximately as fast as if they were read sequentially. An attacker using a platform with a lower cache size, on the other hand, should experience a lower performance due to cache misses. In addition, this pseudorandom pattern hinders the creation of simple pipelines in hardware for visiting those rows: even if the attacker keeps all columns in fast memory to avoid latency issues, some selection function will be necessary to choose among these columns on the fly.

Finally, in Lyra2's design 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, 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 distinct 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) [10, 29]. This class of attacks has been shown to be effective, for example, against certain implementations of the Advanced Encryption Standard (AES) [45] and RSA [56], allowing the recovery of the secret key employed by the algorithms [10, 52].

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 [29]. 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 [55]. This happens because such environments potentially create the required conditions for making cache-timing measurements [55], 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 time-memory trade-offs discussed along Section 5.1.

The reasoning behind this design decision of 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 [29], according to which the server sends only the salt to the client (preferably using a secure channel), who responds with $x = \text{PHS}(pwd, salt)$, 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 execution of one 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 critical concern.

Third, as discussed in [44], 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 [55] 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 can be integrated into its core design for exploring different aspects, namely: giving users better control over the algorithm's bandwidth usage (parameter δ); taking advantage of parallelism capabilities potentially available on the legitimate user's platform (parameter p); and allowing finer-grained control over the algorithm's processing time (parameter γ).

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 each of the proposed extensions, although simple, result in a few lines of pseudo-code being modified. After that, we present the resulting pseudocode when all extensions are combined into the core algorithm, for recommended parameters (namely, $\delta = 1$ when $p = 1$ and $\delta = 0$ when $p \geqslant 2$).

6.1 Controlling the algorithm's bandwidth usage

One possible adaptation of the algorithm consists in allowing the user to control the number of rows involved in each iteration of the Visitation Loop. The reason is that, while Algorithm 2 suggests that a single row index besides row^0 should be employed during the Setup and Wandering phases, this number could actually be controlled by a $\delta \geqslant 0$ parameter. Algorithm 2 can, thus, be seen as the particular case in which $\delta = 1$, while the original Lyra is more similar (although not identical) to Lyra2 with $\delta = 0$. This allows a better control over the algorithm's total memory bandwidth usage, so it can better match the bandwidth available at the legitimate platform.

This parameterization brings positive consequences in terms of security. For example, the number of rows written during the Wandering phase defines the speed in which the memory matrix is modified and, thus, the number of levels in the dependence tree discussed in Section 5.1.3.3. As a result, the $2T$ observed in Equations 5 and 8 would actually become $(\delta + 1)T$. The number of rows read, on its turn, determines the tree's branching factor and, consequently, the probability that a previously discarded row will incur recomputations in Equations 3 an 6. With $\delta > 1$, it is also possible to raise the Setup phase minimum memory usage above the $R/2$ defined by Lemma 1. This can be accomplished by choosing visitation patterns for $row^{\delta \geqslant 2}$ that force the attacker to keep rows that, otherwise, could be discarded right after the middle of the Setup phase. Finding the best visitation order in the Setup phase for each row^{δ} when an arbitrarily large value of δ is adopted remains, however, as an open issue.

Even though the security implications of having $\delta \geqslant 2$ may be of interest, the main disadvantage of this approach is that the higher number of rows picked potentially leads to performance penalties due to memory-related operations. This may oblige legitimate users to reduce the value of T to keep Lyra2's running time below a certain threshold, which in turn would be beneficial to attack platforms having high memory bandwidth and able to mask memory latency (e.g., using idle cores that are waiting for input to run different password guesses). Indeed, according to our tests, we observed slow downs from more than 100% to approximately 50% with each increment of δ. Therefore, the interest of supporting a customizable δ depends on actual tests made on the target platform. For this reason, in this document we only explore further the ability of allowing $\delta = 0$, which is advantageous in combination with Lyra2's multicore variant described Section 6.2.

6.2 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 attacks involving 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 output 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 output is now computed from two instances of the PHS with the same 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) each of the p slices of the shared memory matrix depends on rows generated from multiple threads. The first property leads to a smaller need of synchronism between threads, facilitating the algorithm's processing by highly parallel platforms. The second property, on its turn, makes it harder to run each thread separately with a reduced memory usage, although it may be disabled if the algorithm should run in platforms that do not deal well with cross-reading between threads (e.g., current GPUs).

Along the discussion, we assume that $\delta = 0$, which is the recommended parameterization for attaining good performance with Lyra2$_p$.

6.2.1 Structure and rationale

First, during the Setup phase, p sponge copies are generated. This is done similarly to Lyra2, the difference being that the *params* fed to each sponge S_i $(0 \leqslant i \leqslant p-1)$ must contain the value of $p \,\|\, i$ in addition to any other information already included in line 2 of Algorithm 2.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 then 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 similarly to algorithm's non-parallelizable version, with one important difference: in each duplexing operation, one extra row index, row_p, is included in the computation of the sponge's input.

xxx

in the Setup and Wandering phases,

added to duplexed

besides the (1) the row^1 and $prev^1$ indices accessed by S_i during the Filling Loop

xxx

(i.e., those indexed by the variable) remain limited to its own slice instead of sweeping the whole memory matrix; (2)

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.

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 *rowx* picked in that iteration is computed in line 28 as "$\texttt{offset} + \mathrm{LSW}(rand) \oplus prev) \bmod R/2p$", with $\texttt{offset} = 0$ when $row < R/2p$ and $\texttt{offset} = R/2p$ otherwise.

Second, S_i's duplexing operation (line 31) is applied to $M_i[prev] \oplus M_i[rowx] \oplus M_j[rowx_p]$, where $M_j[rowx_p]$ corresponds to a pseudorandom row from a pseudorandom slice $j \neq i$. More precisely, we have $M_j[rowx_p] = M_j[(rowx + R/2p) \bmod R/p]$, meaning that $rowx_p$ refers to an index in M_j that is at the same position as $rowx$ 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 $j = (\mathrm{LSW}(rand) \bmod p)$); if the value of j computed in this manner is such that $j = i$, make it $j = i + gap$ 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.

6.2.2 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 $M_1[rowx_p]$ (resp. $M_0[rowx_p]$) for the used of S_0 (resp. S_1). Alternative memory distributions (e.g., one that explores the "storage-ahead" strategy described in Section **??**) 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.

6.3 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, on average, each row would be visited only once rather than twice.All things considered, 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 [5] 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 [5], 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

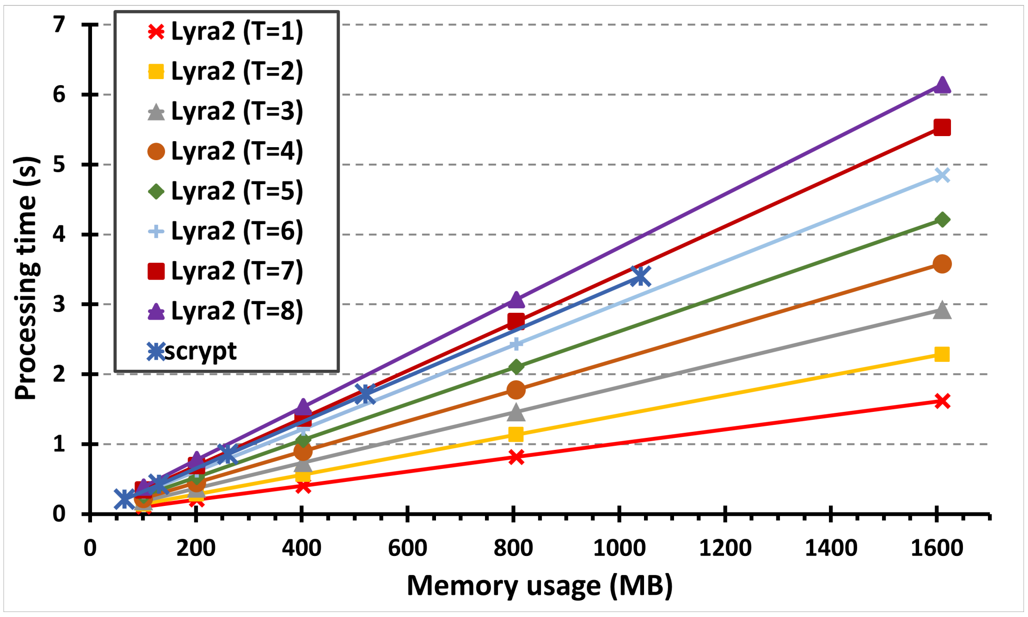

Figure 12: *Performance of our non-SSE Lyra2 implementation, for $C = 128$, $\rho = 1$, and different T and R settings, compared with non-SSE scrypt.*

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 12. 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.

Figure 12 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 [51] (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 $T \approx 6$.

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 [5]. Figure 13 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 [63], Table 3 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 [48, 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 3 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

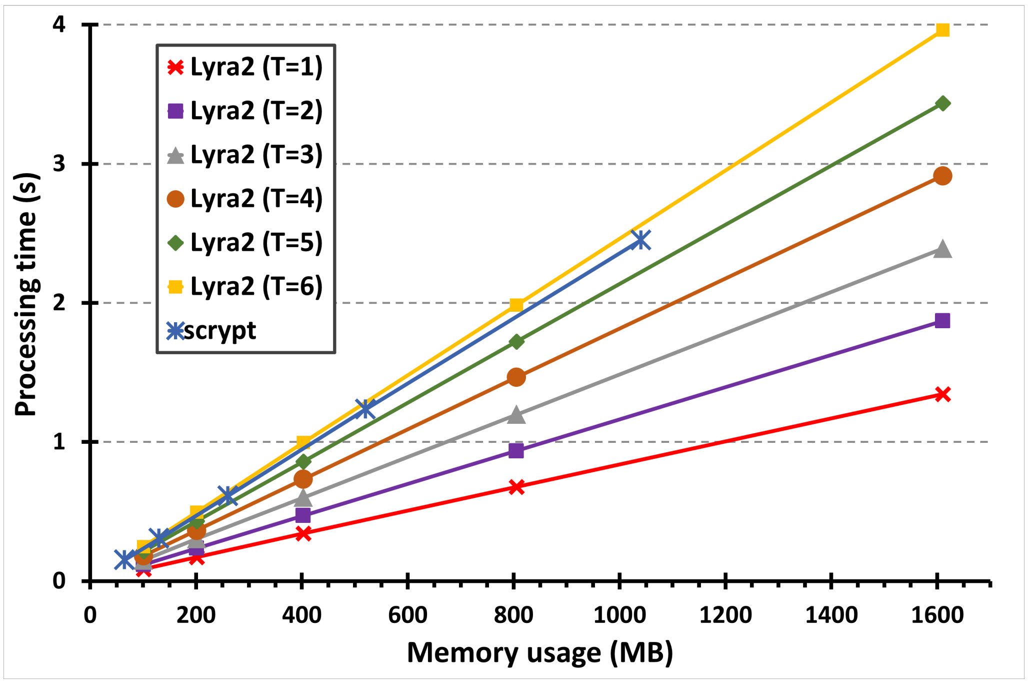

Figure 13: *Performance of our SSE-enabled Lyra2 implementation, for $C = 128$, $\rho = 1$, and different T and R settings, , compared with SSE-enabled scrypt.*

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 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 *BlockMix*, 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) [32]. 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).

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 14. 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

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 3: *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.*

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

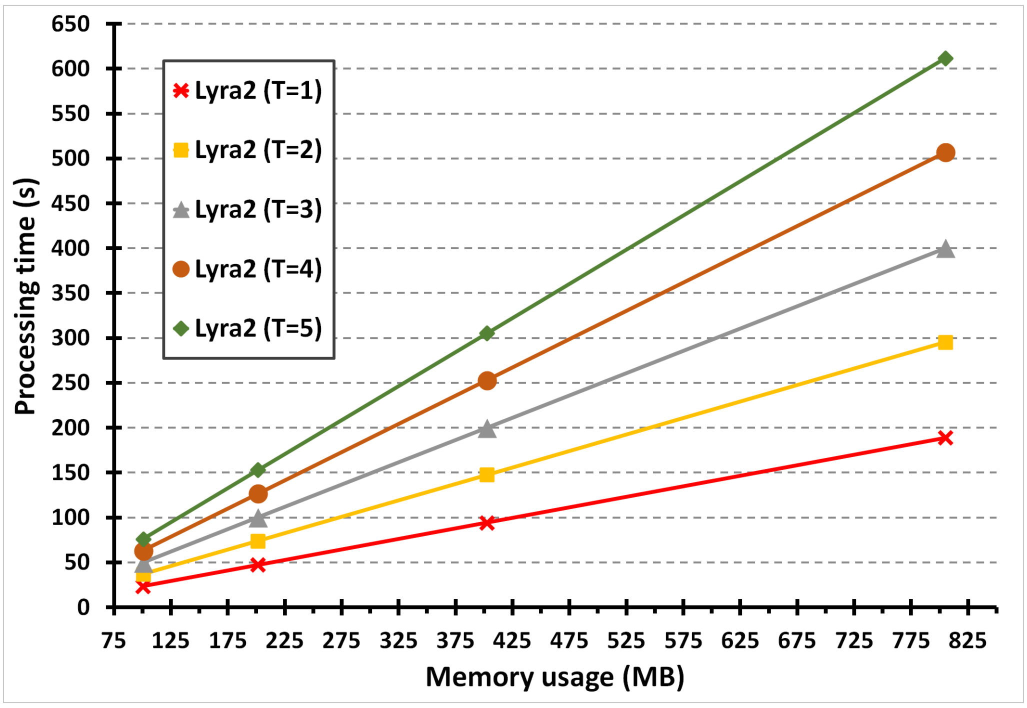

Figure 14: *Performance of a preliminary GPU-oriented Lyra2 implementation, for $C = 128$, $\rho = 1$, and different T and R settings, on NVIDIA GeForce GTX470.*

the interest of such tweaks and their potential integration into Lyra2's core remains, however, as a matter of future work.

Acknowledgements

This work was supported by the Brazilian National Counsel of Technological and Scientific Development (CNPq) under grants 482342/2011-0, 473916/2013-4, under productivity research grants 305350/2013-7 and 306935/2012-0, as well as by the São Paulo Research Foundation (FAPESP) under grant 2011/21592-8.

References

[1] L. Almeida, E. Andrade, P. Barreto, and M. Simplicio. Lyra: Password-based key derivation with tunable memory and processing costs. *J Cryptogr Eng (to appear)*, 2014. See also `eprint.iacr.org/2014/030`. 7, 55, 56

[2] E. Andreeva, B. Mennink, and B. Preneel. The Parazoa family: Generalizing the Sponge hash functions. *IACR Cryptology ePrint Archive*, 2011:28, 2011. 8

[3] Apple. iOS security. Technical report, Apple Inc., 2012. `http://images.apple.com/ipad/business/docs/iOS_Security_May12.pdf`. 53

[4] J-P. Aumasson, S. Fischer, S. Khazaei, W. Meier, and C. Rechberger. New features of latin dances: Analysis of Salsa, ChaCha, and Rumba. In *Fast Software Encryption*, volume 5084, pages 470–488, Berlin, Heidelberg, 2008. Springer-Verlag. 13

[5] J-P. Aumasson, S. Neves, Z. Wilcox-O'Hearn, and C. Winnerlein. BLAKE2: simpler, smaller, fast as MD5. `https://blake2.net/`, 2013. 21, 44, 45

[6] Jean-Philippe Aumasson, Jian Guo, Simon Knellwolf, Krystian Matusiewicz, and Willi Meier. Differential and invertibility properties of BLAKE. In *Fast Software Encryption*, pages 318–332. Springer, 2010. 21

[7] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Raphael Phan. SHA-3 proposal BLAKE (version 1.3). `https://131002.net/blake/blake.pdf`, 2010. 21

[8] Mihir Bellare, Thomas Ristenpart, and Stefano Tessaro. Multi-instance security and its application to password-based cryptography. In *Advances in Cryptology (CRYPTO 2012)*, volume 7417 of *LNCS*, pages 312–329. Springer Berlin Heidelberg, 2012. 53

[9] D. Bernstein. The Salsa20 family of stream ciphers. In Matthew Robshaw and Olivier Billet, editors, *New Stream Cipher Designs*, pages 84–97. Springer-Verlag, Berlin, Heidelberg, 2008. 13

[10] Daniel J. Bernstein. Cache-timing attacks on aes. Technical report, University of Illinois, 2005. `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`. 39

[11] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Sponge functions. (ECRYPT Hash Function Workshop 2007), 2007. Also available at `http://csrc.nist.gov/pki/HashWorkshop/Public_Comments/2007_May.html`. 7, 8

[12] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Cryptographic sponge functions - version 0.1. http://keccak.noekeon.org/, 2011. 7, 8, 9, 15

[13] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. The Keccak SHA-3 submission. Submission to NIST (Round 3), 2011. 8, 9, 21

[14] Joseph Bonneau, Cormac Herley, Paul C. van Oorschot, and Frank Stajano. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *IEEE Symposium on Security and Privacy*, pages 553–567, 2012. 7

[15] S. Chakrabarti and M. Singbal. Password-based authentication: Preventing dictionary attacks. *Computer*, 40(6):68–74, june 2007. 7

[16] Shu-jen Chang, Ray Perlner, William E Burr, Meltem Sönmez Turan, John M Kelsey, Souradyuti Paul, and Lawrence E Bassham. *Third-Round Report of the SHA-3 Cryptographic Hash Algorithm Competition*. US Department of Commerce, National Institute of Standards and Technology, 2012. 21

[17] Eric S. Chung, Peter A. Milder, James C. Hoe, and Ken Mai. 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, pages 225–236, Washington, DC, USA, 2010. IEEE Computer Society. 11

[18] A. Conklin, G. Dietrich, and D. Walz. 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*, pages 170–179, Washington, DC, USA, 2004. IEEE Computer Society. 7

[19] Stephen A. Cook. An observation on time-storage trade off. In *Proc. of the 5th Annual ACM Symposium on Theory of Computing (STOC'73)*, pages 29–33, New York, NY, USA, 1973. ACM. 24

[20] Becky Crew. 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`. 56

[21] J. Daemen and V. Rijmen. A new MAC construction ALRED and a specific instance ALPHA-mac. In *Fast Software Encryption – FSE'05*, pages 1–17, 2005. 16

[22] J. Daemen and V. Rijmen. Refinements of the ALRED construction and MAC security claims. *Information Security, IET*, 4(3):149–157, 2010. 16

[23] Yoginder S. Dandass. Using FPGAs to parallelize dictionary attacks for password cracking. In *Proc. of the 41st Annual Hawaii International Conference on System Sciences (HICSS 2008)*, pages 485–485. IEEE, 2008. 11

[24] Johann Großschädl Dmitry Khovratovich, Alex Biryukov. Tradeoff cryptanalysis of password hashing schemes. PasswordsCon'14, 2014. See also `https://www.cryptolux.org/images/4/4f/PHC-overview.pdf`. 35, 36

[25] Markus Dürmuth, Tim Güneysu, and Markus Kasper. Evaluation of standardized password-based key derivation against parallel processing platforms. In *Computer Security–ESORICS 2012*, volume 7459 of *LNCS*, pages 716–733. Springer Berlin Heidelberg, 2012. 7, 11

[26] Cynthia Dwork, Moni Naor, and Hoeteck Wee. Pebbling and proofs of work. In *Advances in Cryptology – CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 37–54. Springer Berlin Heidelberg, 2005. 24

[27] Stefan Dziembowski, Tomasz Kazana, and Daniel Wichs. Key-evolution schemes resilient to space-bounded leakage. In *Advances in Cryptology – CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 335–353. Springer Berlin Heidelberg, 2011. 24

[28] D. Florencio and C. Herley. A large scale study of web password habits. In *Proc. of the 16th International Conference on World Wide Web*, pages 657–666, Alberta, Canada, 2007. 7

[29] Christian Forler, Stefan Lucks, and Jakob Wenzel. Catena: A memory-consuming password scrambler. Cryptology ePrint Archive, Report 2013/525, 2013. `http://eprint.iacr.org/`. 23, 24, 39

[30] Jeremy Fowers, Greg Brown, Patrick Cooke, and Greg Stitt. 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)*, pages 47–56, New York, NY, USA, 2012. ACM. 11

[31] Kris Gaj, Ekawat Homsirikamol, Marcin Rogawski, Rabia Shahid, and Malik Umar Sharif. 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/`. 21

[32] GeForce. GeForce GTX 470: Specifications. `http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-470/specifications` (visited on Mar.29, 2014), 2014. 47

[33] Jian Guo, Pierre Karpman, Ivica Nikolić, Lei Wang, and Shuang Wu. Analysis of BLAKE2. In *Topics in Cryptology (CT-RSA 2014)*, volume 8366 of *LNCS*, pages 402–423. Springer International Publishing, 2014. see also `https://eprint.iacr.org/2013/467.pdf`. 21

[34] J. Halderman, S. Schoen, N. Heninger, W. Clarkson, W. Paul, J. Calandrino, A. Feldman, J. Appelbaum, and E. Felten. Lest we remember: cold-boot attacks on encryption keys. *Commun. ACM*, 52(5):91–98, May 2009. 22

[35] M.E. Hellman. A cryptanalytic time-memory trade-off. *IEEE Transactions on Information Theory*, 26(4):401–406, 1980. 24

[36] C. Herley, P. van Oorschot, and A. Patrick. Passwords: If we're so smart, why are we still using them? In *Financial Cryptography and Data Security*, volume 5628 of *LNCS*, pages 230–237. Springer Berlin / Heidelberg, 2009. 7

[37] Athanasios P. Kakarountas, Haralambos Michail, Athanasios Milidonis, Costas E. Goutis, and George Theodoridis. High-speed FPGA implementation of secure hash algorithm for IPSec and VPN applications. *The Journal of Supercomputing*, 37(2):179–195, 2006. 11

[38] B. Kaliski. *PKCS#5: Password-Based Cryptography Specification version 2.0 (RFC 2898)*, 2000. 7, 10, 12, 53

[39] Poul-Henning Kamp. Md5crypt. `https://www.usenix.org/legacyurl/md5-crypt`, 1999. See also `http://dir.gmane.org/gmane.comp.security.phc`. 2

[40] John Kelsey, Bruce Schneier, Chris Hall, and David Wagner. Secure applications of low-entropy keys. In *Proc. of the 1st International Workshop on Information Security*, ISW '97, pages 121–134, London, UK, UK, 1998. Springer-Verlag. 10

[41] Khronos Group. *The OpenCL Specification – Version 1.2*, 2012. 11

[42] M. Marechal. Advances in password cracking. *Journal in Computer Virology*, 4(1):73–81, 2008. 7, 11

[43] Mao Ming, He Qiang, and Shaokun Zeng. Security analysis of BLAKE-32 based on differential properties. In *2010 International Conference on Computational and Information Sciences (ICCIS)*, pages 783–786. IEEE, 2010. 21

[44] Keaton Mowery, Sriram Keelveedhi, and Hovav Shacham. Are AES x86 cache timing attacks still feasible? In *Proc.s of the 2012 ACM Workshop on Cloud Computing Security Workshop (CCSW'12)*, pages 19–24, New York, NY, USA, 2012. ACM. 40

[45] 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`. 39

[46] 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`. 13

[47] 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`. 7, 13

[48] 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`. 7, 10, 46

[49] Nvidia. CUDA C programming guide. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/`, 2012. 11

[50] Nvidia. Tesla Kepler family product overview. `http://www.nvidia.com/content/tesla/pdf/Tesla-KSeries-Overview-LR.pdf`, 2012. 11

[51] C. Percival. Stronger key derivation via sequential memory-hard functions. In *BSDCan 2009 – The Technical BSD Conference*, 2009. 7, 12, 13, 21, 45

[52] Colin Percival. Cache missing for fun and profit. In *Proc. of BSDCan 2005*, 2005. 39

[53] PHC. Password hashing competition. `https://password-hashing.net/`, 2013. 7, 12

[54] N. Provos and D. Mazières. A future-adaptable password scheme. In *Proc. of the FREENIX track: 1999 USENIX annual technical conference*, 1999. 7, 12, 53

[55] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 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, pages 199–212, New York, NY, USA, 2009. ACM. 39, 40

[56] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, Feb 1978. 39

[57] B. Schneier. Description of a new variable-length key, 64-bit block cipher (Blowfish). In *Fast Software Encryption, Cambridge Security Workshop*, pages 191–204, London, UK, 1994. Springer-Verlag. 53

[58] SciEngines. Rivyera s3-5000. `http://sciengines.com/products/computers-and-clusters/rivyera-s3-5000.html`. 11, 12

[59] SciEngines. Rivyera v7-2000t. `http://sciengines.com/products/computers-and-clusters/v72000t.html`. 12

[60] M. A. Simplicio Jr, P. Barbuda, P. Barreto, T. Carvalho, and C. Margi. The MARVIN message authentication code and the LETTERSOUP authenticated encryption scheme. *Security and Communication Networks*, 2:165–180, 2009. 16

[61] M. A. Simplicio Jr and P. S. L. M. Barreto. Revisiting the security of the ALRED design and two of its variants: Marvin and LetterSoup. *IEEE Transactions on Information Theory*, 58(9):6223–6238, 2012. 16

[62] Martijn Sprengers. GPU-based password cracking: On the security of password hashing schemes regarding advances in graphics processing units. Master's thesis, Radboud University Nijmegen, 2011. 7, 11

[63] TrendForce. DRAM contract price (jan.22 2014). http://www.trendforce.com/price (visited on Mar.29, 2014), 2014. 46

[64] TrueCrypt. TrueCrypt: Free open-source on-the-fly encryption – documentation. `http://www.truecrypt.org/docs/`, 2012. 53

[65] Matt Weir, Sudhir Aggarwal, Breno de Medeiros, and Bill Glodek. Password cracking using probabilistic context-free grammars. In *Proc. of the 30th IEEE Symposium on Security and Privacy*, SP'09, pages 391–405, Washington, DC, USA, 2009. IEEE Computer Society. 10

[66] F.F. Yao and Y.L. Yin. Design and analysis of password-based key derivation functions. *IEEE Transactions on Information Theory*, 51(9):3292–3297, 2005. 53

[67] J. Yuill, D. Denning, and F. Feer. Using deception to hide things from hackers: Processes, principles, and techniques. *Journal of Information Warfare*, 5(3):26–40, 2006. 22

Appendix A. PBKDF2

The Password-Based Key Derivation Function version 2 (PBKDF2) algorithm [38] was originally proposed in 2000 as part of RSA Laboratories' PKCS#5. It is nowadays present in several security tools, such as TrueCrypt [64] and Apple's iOS for encrypting user passwords [3], and has been formally analyzed in several circumstances [66, 8].

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.

Appendix B. Bcrypt

Another solution that allows users to configure the password hashing processing time is bcrypt [54]. The scheme is based on a customized version of the 64-bit cipher algorithm Blowfish [57], 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 EksBlowfishSetup function, and then using EksBlowfish for iteratively encrypting a constant string, 64 times.

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

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 $K$
```

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 $\pi$ 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 $T \,\|\, salt \,\|\, ctext$
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
```

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 considered (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 [1] are detailed in Algorithm 5.

Algorithm 5 The Lyra Algorithm.

```
PARAM: $Hash$   ▷ Sponge with block size $b$ and underlying perm. $f$
PARAM: $\rho$   ▷ 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
 2: $Hash.absorb(\mathtt{pad}(salt \parallel pwd))$   ▷ 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
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: $Hash.absorb(\mathtt{pad}(salt))$   ▷ 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 pseudorandom 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. On the algorithm's name

The name "Lyra" comes from *Chondrocladia lyra*, a recently discovered type of sponge [20]. 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 [1], 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. ☺