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

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

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

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

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

Note 1. *This paper corresponds to version 3 of Lyra2, originally submitted to the Password Hashing Competition (`https://password-hashing.net/`).*

1 Introduction

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

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

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

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

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

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

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

The rest of this paper is organized as follows. Section 2 outlines the concept of cryptographic sponges. Section 3 describes the main requirements of PHS solutions and discusses the related work. Section 4 presents the Lyra2 algorithm and its design rationale, while Section 5 analyzes its security. Section 6 discusses extensions of Lyra2, all of which can be integrated into the basic algorithm discussed in Section 4, presenting in especial the parallelizable version of the algorithm, called Lyra2$_p$. Section 7 shows our 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 [16] and is described in detail in [17]. The elegant design of sponges has also motivated the creation of more general structures, such as the Parazoa family of functions [18]. Indeed, their flexibility is probably among the reasons that led Keccak [19], one of the members of the sponge family, to be elected as the new Secure Hash Algorithm (SHA-3).

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 rot operation).

Symbol	*Meaning*
$\oplus$	bitwise Excusive-OR (XOR) 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
$\mathrm{lsw}(x)$	the least significant word of x
$x \ggg n$	n-bit right rotation of x
$\mathrm{rot}(x)$	ω-bit right rotation of x
$\mathrm{rot}^y(x)$	ω-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, 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 [19]).

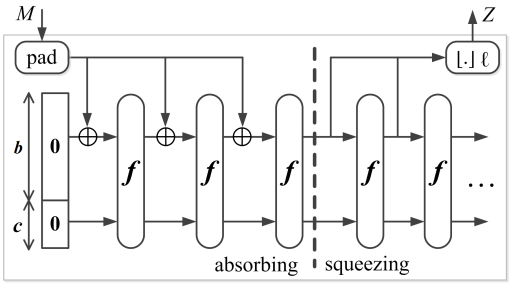


Fig. 1: Overview of the sponge construction $Z = [f, \texttt{pad}, b](M, \ell)$. Adapted from [17].

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* [17], depicted in Figure 2.

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 $\ell \leqslant b$.

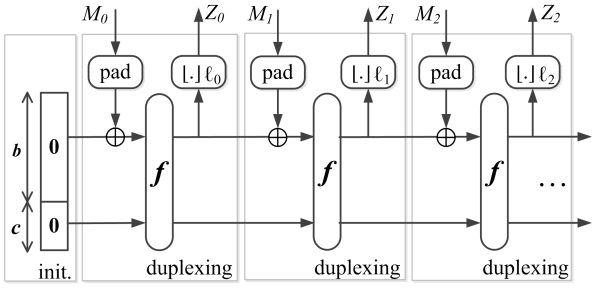


Fig. 2: Overview of the duplex construction. Adapted from [17].

3 Password Hashing Schemes (PHS)

As previously discussed, the basic requirement for a PHS is to be non-invertible, so that recovering the password from its output is computationally infeasible. Moreover, a good PHS's output is expected to be indistinguishable from random bit strings, preventing an attacker from discarding part of the password space based on perceived patterns [20]. In principle, those requirements can be easily accomplished simply by using a secure hash function, which by itself ensures that the best attack venue against the derived key is through brute force (possibly aided by a dictionary or "usual" password structures [7,21]).

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

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

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

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

3.1 Attack platforms

The most dangerous threats faced by any PHS comes from platforms that benefit from "economies of scale", especially when cheap, massively parallel hardware is available. The most prominent examples of such platforms are Graphics Processing Units (GPUs) and custom hardware synthesized from FPGAs [10].

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 [22] and OpenCL [23]). As a result, they became more intensively employed for more general purposes, including password cracking [10,11].

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

the GPU's memory limit of 6 GB. For example, if a sequential PHS requires 20 MB of DRAM, the maximum number of cores that could be used simultaneously becomes 300, only 11% of the total available.

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 [25,26]). Hence, as long as enough resources (i.e., logic gates and memory) are available in the underlying hardware, FPGAs potentially yield a more cost-effective solution than what would be achieved with a general-purpose CPU of similar cost [12]. When compared to GPUs, FPGAs may also be advantageous due to the latter's considerably lower energy consumption [27,28], which can be further reduced if its circuit is synthesized in the form of custom logic hardware (ASIC) [27].

A recent example of password cracking using FPGAs is presented in [10]. Using a RIVYERA S3-5000 cluster [29] with 128 FPGAs against PBKDF2-SHA-512, the authors reported a throughput of 356,352 passwords tested per second in an architecture having 5,376 password processed in parallel. It is interesting to notice that one of the reasons that made these results possible is the small memory usage of the PBKDF2 algorithm, as most of the underlying SHA-2 processing is performed using the device's memory cache (much faster than DRAM) [10, Sec. 4.2]. Against a PHS requiring 20 MB to run, for example, the resulting throughput would presumably be much lower, especially considering that the FPGAs employed can have up to 64 MB of DRAM [29] and, thus, up to three passwords can be processed in parallel rather than 5,376.

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

3.2 Scrypt

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

The design of scrypt [5] focus on coupling memory and time costs. For this, scrypt employs the concept of "sequential memory-hard" functions: an algorithm that asymptotically uses almost as much memory as it requires operations and

for which a parallel implementation cannot asymptotically obtain a significantly lower cost. As a consequence, if the number of operations and the amount of memory used in the regular operation of the algorithm are both $\mathcal{O}(R)$, the complexity of a memory-free attack (i.e., an attack for which the memory usage is reduced to $\mathcal{O}(1)$) becomes $\Omega(R^2)$, where R is a system parameter. We refer the reader to [5] for a more formal definition.

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

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

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

function ROMIX(B, R) ▷ Sequential memory-hard function
  $X \leftarrow B$
  for $i \leftarrow 0$ **to** $R - 1$ **do** ▷ Initializes memory array M
    $V_i \leftarrow X$; $X \leftarrow \text{BLOCKMIX}(X)$
  end for
  for $i \leftarrow 0$ **to** $R - 1$ **do** ▷ Reads random positions of M
    $j \leftarrow Integerify(X) \bmod R$
    $X \leftarrow \text{BLOCKMIX}(X \oplus M_j)$
  end for
  return X
end function

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

the Salsa20/8 [32] stream cipher, whose output length is $h = 512$. After the p *ROMix* processes are over, the B_i blocks are used as salt in one final iteration of the PBKDF2 algorithm, outputting key K.

Scrypt displays a very interesting design, being one of the few existing solutions that allow the configuration of both processing and memory costs. One of its main shortcomings is probably the fact that it strongly couples memory and processing requirements for a legitimate user. Specifically, scrypt's design prevents users from raising the algorithm's processing time while maintaining a fixed amount of memory usage, unless they are willing to raise the p parameter and allow further parallelism to be exploited by attackers. Another inconvenience with scrypt is the fact that it employs two different underlying hash functions, HMAC-SHA-256 (for the PBKDF2 algorithm) and Salsa20/8 (as the core of the *BlockMix* function), leading to increased implementation complexity. Finally, even though Salsa20/8's known vulnerabilities [33] are not expected to put the security of scrypt in hazard [5], using a stronger alternative would be at least advisable, especially considering that the scheme's structure does not impose serious restrictions on the internal hash algorithm used by *BlockMix*. In this case, a sponge function could itself be an alternative, with the advantage that, since sponges support inputs and outputs of any length, the whole *BlockMix* structure could be replaced.

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 thwarting the exploitation of time-memory trade-offs (TMTO). This characteristic should discourage 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 regular attack venues (i.e., those not exploring TMTO) beyond that of scrypt's.

4 Lyra2

As any PHS, Lyra2 takes as input a salt and a password, creating a pseudorandom output that can then be used as key material for cryptographic algorithms or as an authentication string [4]. Internally, the scheme's memory is organized as a matrix that is expected to remain in memory during the whole password hashing process: since its cells are iteratively read and written, discarding a cell for saving memory leads to the need of recomputing it whenever it is accessed once again, until the point it was last modified. 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 zero), ensuring the sequential nature of the whole process. Also, the number of times the matrix's cells are revisited after initialization is defined

Algorithm 2 The Lyra2 Algorithm.

PARAM: H ▷ Sponge with block size b (in bits) and underlying permutation f
PARAM: H_ρ ▷ Reduced-round sponge for use in the Setup and Wandering phases
PARAM: ω ▷ Number of bits to be used in rotations (recommended: a multiple of 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
INPUT: C ▷ Number of columns in the memory matrix (recommended: $C \cdot \rho \geqslant \rho_{max}$)
INPUT: k ▷ The desired hashing output length, in bits
OUTPUT: K ▷ The password-derived k-long hash

▷ **Bootstrapping phase**: Initializes the sponge's state and local variables
▷ Byte representation of input parameters (others can be added)
$params \leftarrow len(k) \parallel len(pwd) \parallel len(salt) \parallel T \parallel R \parallel C$
$H.absorb(\texttt{pad}(pwd \parallel salt \parallel params))$ ▷ Padding rule: 10^*1.
$gap \leftarrow 1$; $stp \leftarrow 1$; $wnd \leftarrow 2$; $sqrt \leftarrow 2$ ▷ Initializes visitation step and window
$prev^0 \leftarrow 2$; $row^1 \leftarrow 1$; $prev^1 \leftarrow 0$

▷ **Setup phase**: Initializes a $(R \times C)$ memory matrix, it's cells having b bits each
for $(col \leftarrow 0$ **to** $C-1)$ **do** $\{M[0][C-1-col] \leftarrow H_\rho.squeeze(b)\}$ **end for**
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**
for $(col \leftarrow 0$ **to** $C-1)$ **do** $\{M[2][C-1-col] \leftarrow M[1][col] \oplus H_\rho.duplex(M[1][col], b)\}$ **end for**
for $(row^0 \leftarrow 3$ **to** $R-1)$ **do** ▷ **Filling Loop**: initializes remainder rows
  ▷ **Columns Loop**: $M[row^0]$ is initialized; $M[row^1]$ is updated
  for $(col \leftarrow 0$ **to** $C-1)$ **do**
    $rand \leftarrow H_\rho.duplex(M[row^1][col] \boxplus M[prev^0][col] \boxplus M[prev^1][col], b)$
    $M[row^0][C-1-col] \leftarrow M[prev^0][col] \oplus rand$
    $M[row^1][col] \leftarrow M[row^1][col] \oplus \text{rot}(rand)$ ▷ rot(): right rotation by ω bits
  end for
  $prev^0 \leftarrow row^0$; $prev^1 \leftarrow row^1$; $row^1 \leftarrow (row^1 + stp) \bmod wnd$
  if $(row^1 = 0)$ **then** ▷ Window fully revisited
    ▷ Doubles window and adjusts step
    $wnd \leftarrow 2 \cdot wnd$; $stp \leftarrow sqrt + gap$; $gap \leftarrow -gap$
    if $(gap = -1)$ **then** $\{sqrt \leftarrow 2 \cdot sqrt\}$ **end if** ▷ Doubles $sqrt$ every other iteration
  end if
end for

▷ **Wandering phase**: Iteratively overwrites pseudorandom cells of the memory matrix
▷ **Visitation Loop**: $2R \cdot T$ rows revisited in pseudorandom fashion
for $(wCount \leftarrow 0$ **to** $R \cdot T - 1)$ **do**
  $row^0 \leftarrow \text{lsw}(rand) \bmod R$; $row^1 \leftarrow \text{lsw}(\text{rot}(rand)) \bmod R$ ▷ Picks pseudorandom rows
  for $(col \leftarrow 0$ **to** $C-1)$ **do** ▷ **Columns Loop**: updates $M[row^{0,1}]$
    ▷ Picks pseudorandom columns
    $col^0 \leftarrow \text{lsw}(\text{rot}^2(rand)) \bmod C$; $col^1 \leftarrow \text{lsw}(\text{rot}^3(rand)) \bmod C$
    $rand \leftarrow H_\rho.duplex(M[row^0][col] \boxplus M[row^1][col] \boxplus M[prev^0][col^0] \boxplus M[prev^1][col^1], b)$
    $M[row^0][col] \leftarrow M[row^0][col] \oplus rand$ ▷ Updates first pseudorandom row
    $M[row^1][col] \leftarrow M[row^1][col] \oplus \text{rot}(rand)$ ▷ Updates second pseudorandom row
  end for ▷ End of Columns Loop
  $prev^0 \leftarrow row^0$; $prev^1 \leftarrow row^1$ ▷ Next iteration revisits most recently updated rows
end for ▷ End of Visitation Loop

▷ **Wrap-up phase**: output computation
$H.absorb(M[row^0][0])$ ▷ Absorbs a final column with full-round sponge
$K \leftarrow H.squeeze(k)$ ▷ Squeezes k bits with full-round sponge
return K ▷ Provides k-long bitstring as output

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 6). The set of variables $\{gap, stp, wnd, sqrt, prev^0, row^1, prev^1\}$ initialized in lines 5 and 6 are useful only for the next stage of the algorithm, the Setup phase, so the discussion on their properties 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 4). The padding rule adopted by Lyra2 is the multi-rate padding $\texttt{pad10*1}$ described in [17], 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 and passwords: for example, for any $(u, v \mid u + v = \alpha)$, we have a collision if $pwd = \texttt{0}^u$, $salt = \texttt{0}^v$ 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 7 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 [34,35,36,37]. As further discussed in Section 4.2, even though the requirements in the context of password hashing are different, this strategy does not decrease the security of the scheme as long as f_ρ is non-cyclic and highly non-linear, which should be the case for the vast majority of secure hash functions. In some scenarios, it may even be interesting to use a different function as f_ρ rather than a reduced-round version of f itself to attain higher speeds, which is possible as long the alternative satisfies the above-mentioned properties.

Except for rows $M[0]$ to $M[2]$, the sponge's reduced duplexing operation $H_\rho.duplex$ is always called over the wordwise addition of three rows (line 14), all of which must be available in memory for the algorithm to proceed (see the Filling Loop, in lines 11–24).

- $M[prev^0]$: the last row ever initialized in any iteration of the Filling Loop, which means simply that $prev^0 = row^0 - 1$;
- $M[row^1]$: 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 *Lyra*2 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 $M[prev^0]$. Therefore, this design takes advantage of caching while penalizing attacks in which a given $M[row^0]$ is directly recomputed from the corresponding inputs: in this case, $M[prev^0]$ and $M[prev^1]$ may not be in cache, so all three rows must come from the main memory, raising memory latency and bandwidth. 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, the resulting output ($rand$) affects two rows (lines 15 and 16): $M[row^0]$, 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 rot($rand$), i.e., $rand$ rotated to the right by ω bits. More formally, for $\omega = W$ and representing $rand$ as an array of words $rand[0] \dots rand[b/W - 1]$ (i.e., the first b bits of the outer state, from top to bottom as depicted in Figures 1 and 2), we have that $M[row^0][C-1-i] \leftarrow M[prev^0][i] \oplus rand[i]$ and $M[row^1][i] \leftarrow M[row^1][i] \oplus rand[(i-1) \bmod (b/W)]$ $(0 \leqslant i \leqslant b/W-1)$. 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

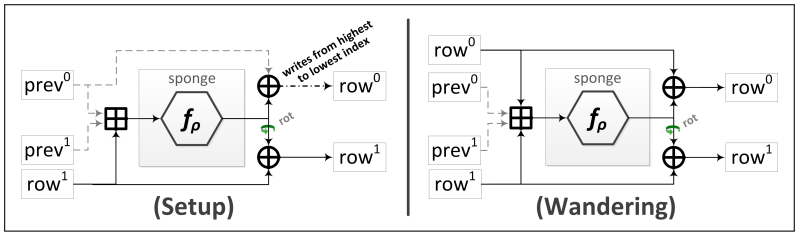


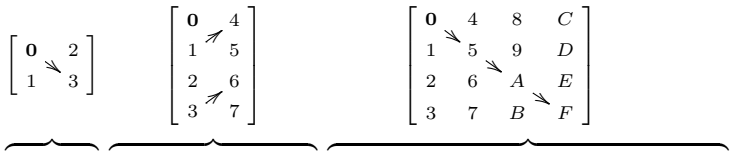
Fig. 3: Handling the sponge's inputs and outputs during the Setup (left) and Wandering (right) phases in Lyra2.

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 rot 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 in software if ω 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 left side of Figure 3 illustrates how the sponge's inputs and output are handled by Lyra2 during the Setup phase.

The initialization of $M[0] - M[2]$ in lines 8 to 10, in contrast, is slightly different because none of them has enough predecessors to be treated exactly like the rows initialized during the Filling Loop. Specifically, instead of taking three rows in the duplexing operation, $M[0]$ takes none while $M[1]$ and (for simplicity) $M[2]$ take only their immediate predecessor.

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 of size wnd, which 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 depend on wnd, following a logic whose aim is to ensure that, if two rows are visited sequentially in one window, during the subsequent window they are visited (1) in points far away from each other and (2) approximately in the reverse order of their previous visitation. This hinders the recomputation of several values of $M[row^1]$ from scratch in the sequence they are required, thwarting attacks that trade memory and processing costs, which are discussed in detail in Section 5.1. To accomplish this goal in a manner that is simple to implement, the following strategy was adopted (see Table 2):

– When wnd is a square number: the window can be seen as a $\sqrt{wnd} \times \sqrt{wnd}$ matrix. Then, row^1 is taken from the indices in that matrix's cyclic diagonals, starting with the main diagonal and moving right until the diagonal from the upper right corner is reached. This is accomplished by using a step variable $stp = \sqrt{wnd} + 1$, computed in line 21 of Algorithm 2, using the auxiliary $sqrt = \sqrt{wnd}$ variable to facilitate this computation.



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	–	–	–	1	**0**	3	2	1	**0**	3	6	1	4	7	2	5	**0**	5	A	F	4	9	E	3	8	D	2	7	...
$prev^1$	–	–	–	0	1	0	3	2	1	0	3	6	1	4	7	2	5	0	5	A	F	4	9	E	3	8	D	2	...
wnd	–	–	–	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).

– Otherwise: the window is represented as a $2\sqrt{wnd/2} \times \sqrt{wnd/2}$ matrix. The values of row^1 start with 0 and then corresponding to the matrix's cyclic anti-diagonals, starting with the main anti-diagonal and cyclically moving left one column at a time. In this case, the step variable is computed as $stp = 2\sqrt{wnd/2} - 1$ in the same line 21 of Algorithm 2, once again using the auxiliary $sqrt = 2\sqrt{wnd/2}$ variable.

Table 2 shows some examples of the values of row^1 in each iteration of the Filling Loop (lines 11–24), 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 18 can be implemented with a simple bitwise AND with $wnd - 1$, potentially leading to better performance.

4.1.3 The Wandering phase The most time-consuming of all phases, the *Wandering Phase* (lines 27 to 37), 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 32) for computing a random-like output *rand* (line 32), 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, which is 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 columns are not sequential but actually controlled by the pseudorandom, password-dependent variables $(col^0, col^1) \in [0, C-1]$. More precisely, each index col^d $(d = 0, 1)$ is computed from the sponge's outer state; for example, for $\omega = W$, it is taken from $rand[d+2]$) right before each duplexing operation (line 31). 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 for better performance 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 $M[row^0]$ (line 33) and, after being rotated, with $M[row^1]$ (line 34). However, in the Wandering phase the sponge's output is XORed with $M[row^0]$ from the lowest to the highest index, just like $M[row^1]$. This design decision was adopted because it allows faster processing, since the columns read are also those overwritten; at the same time, the subsequent reading of those columns in a pseudorandom order already thwarts the attack strategy discussed in Section 5.1.2.5, so there is no need to revert the the reading/writing order in this part of the algorithm.

4.1.4 The Wrap-up phase Finally, after $(R \cdot T)$ duplexing operations are performed during the Wandering phase, the algorithm enters the *Wrap-up Phase*. This phase consists of a full-round absorbing operation (line 39) of a single cell of the memory matrix, $M[row^0][0]$. 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, since the full-round absorb in line 4, the state was only updated by several calls to the reduced-round f. This absorb operation is then followed by a full-round squeezing operation (line 40) 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 Lyra2 is equivalent to violate the same basic properties of the underlying full-round 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 13–17 (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. The only difference is that, to parallelize the duplexing of inputs from its 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. 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: $\lceil (|pwd| + |salt| + |params|)/b \rceil$ calls in Bootstrapping phase, plus $R \cdot C \cdot \rho / \rho_{max}$ 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 [19], does not seem to be the best alternative for this purpose. This happens because Keccak excels in hardware rather than

in software performance [38]. Hence, for the specific application of password hashing, it gives more advantage to attackers using custom hardware than to legitimate users running a software implementation.

Our recommendation, thus, is toward using a secure software-oriented algorithm as the sponge's f transformation. One example is Blake2b [39], a slightly tweaked version of Blake [40]. Blake itself displays a security level similar to that of Keccak [41], and its compression function has been shown to be a good permutation [42,43] and to have a strong diffusion capability [40] even with a reduced number of rounds [44,45], while Blake2b is believed to retain most of these security properties [46].

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. The reason is that Blake2b does not use the constants originally employed in Blake2 inside its G function [39], relying on the IV for avoiding possible fixed points. Indeed, if the internal state is filled with zeros as usually done in cryptographic sponges, any block filled with zeros absorbed by the sponge will not change this state value. 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 4 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.

4.4.1 A dedicated, multiplication-hardened sponge: BlaMka. Besides plain Blake2b, another potentially interesting alternative is to employ a permutation that involves integer multiplications among its operations. The reason is that, as verified in several benchmarks available in the literature [47,48], the performance gain offered by hardware implementations of the multiplication operation is not much higher than what is obtained with software implementations running on x86 platforms, for which such operations are already heavily optimized. Those optimizations appear in different levels, including compilers, advanced instruction sets (e.g., MMX, SSE and AVX), and architectural details of modern CPUs that resemble those of dedicated FPGAs. Hence, if a legitimate user prefers to rely on a function that provides further protection against hardware platforms while maintaining a high efficiency on platforms such as CPUs, multiplications may be an interesting approach. Indeed, this is the main idea behind the "multiplication-hardening" strategy discussed in [49,50].

For this purpose the Blake2b structure may itself be adapted to integrate multiplications. Namely, multiplications can be integrated into Blake2b's G function (see the left side of Figure 4), which relies on sequential additions, rotations and XORs (ARX) for attaining bit diffusion and creating a mutual dependence between those bits [42,43]. If the additions employed are replaced by a permutation

$a \leftarrow a + b$
$d \leftarrow (d \oplus a) \ggg 32$
$c \leftarrow c + d$
$b \leftarrow (b \oplus c) \ggg 24$
$a \leftarrow a + b$
$d \leftarrow (d \oplus a) \ggg 16$
$c \leftarrow c + d$
$b \leftarrow (b \oplus c) \ggg 63$

(a) Blake2 G function.

$a \leftarrow a + b + 2 \cdot \mathrm{lsw}(a) \cdot \mathrm{lsw}(b)$
$d \leftarrow (d \oplus a) \ggg 32$
$c \leftarrow c + d + 2 \cdot \mathrm{lsw}(c) \cdot \mathrm{lsw}(d)$
$b \leftarrow (b \oplus c) \ggg 24$
$a \leftarrow a + b + 2 \cdot \mathrm{lsw}(a) \cdot \mathrm{lsw}(b)$
$d \leftarrow (d \oplus a) \ggg 16$
$c \leftarrow c + d + 2 \cdot \mathrm{lsw}(c) \cdot \mathrm{lsw}(d)$
$b \leftarrow (b \oplus c) \ggg 63$

(b) BlaMka G function.

Fig. 4: Multiplication-hardened (right) and original (left) $G(a, b, c, d)$ function from Blake2b.

that includes a multiplication and provides at least the same level of diffusion, its security should not be negatively affected.

One suggestion, originally made by Samuel Neves (one of the authors of Blake2) [51], is to replace the additions of integers x and y by something like the latin square function [52] $f(x, y) = x + y + 2 \cdot x \cdot y$. To make it more friendly for implementation using the instruction set of modern processors, however, one can use a slightly modified construction that employs the least significant bits of x and y, namely $f'(x, y) = x + y + 2 \cdot \mathrm{lsw}(x) \cdot \mathrm{lsw}(y)$, as shown in the right side of Figure 4. As a result, this function can be efficiently implemented using fast SIMD instructions (e.g., _MM_MUL_EPU, _MM_SLLI_EPI, _MM_ADD_EPI), and keeps an homogeneous distribution for the $\mathbb{F}_2^{2n} \mapsto \mathbb{F}_2^n$ mapping.

In terms of security, in a preliminary analysis the diffusion capability of f' seems to be at least as high as that provided by the simple word-wise addition employed by Blake2b. This impression comes from the assessment of XOR-differentials over f', defined in [53] as:

Definition 1. *Let $f : \mathbb{F}_2^{2n} \mapsto \mathbb{F}_2^n$ be a vector Boolean function and let α, β and γ be n-bit sized XOR-differences. We call $(\alpha, \beta) \mapsto \gamma$ a XOR-differential of f if there exist n-bit strings x and y that satisfy $f'(x \oplus \alpha, y \oplus \beta) = f'(x, y) \oplus \gamma$. Otherwise, if no such n-bit strings x and y exist, we call $(\alpha, \beta) \mapsto \gamma$ an impossible XOR-differential of f.*

Specifically, conducting an exhaustive search for $n = 8$, we found 4 differentials that hold for all 65536 pairs (x, y), both for f' and for the addition operation: $(0x00, 0x00) \mapsto 0x00$, $(0x80, 0x80) \mapsto 0x00$, $(0x00, 0x80) \mapsto 0x80$, and $(0x80, 0x00) \mapsto 0x80$ (in hexadecimal notation). However, while the addition operation displays 168 XOR-differentials that hold for 50% of all (x, y) pairs, the f' operation hereby described has only 48 of such XOR-differentials, which have the second highest probability for both functions. XOR-differentials with lower, but still high probabilities are also less frequent for f' that for the simple addition operation — e.g., 288 instead of 3024 differentials that hold for 25% of all (x, y) pairs, — although the former displays differentials with probabilities

that do not appear in the latter — e.g., 12 differentials that hold for 19200 out of the 65536 (x, y) pairs, the third highest differential probability for f'.

Even though this multiplication-hardened structure based on Blake2b (codenamed BlaMka) shows promise, we emphasize that it requires further security analysis to be indeed considered a recommended function for use with Lyra2. Indeed, actual instances of BlaMka would use $n = 32$ or $n = 64$ rather than the $n = 8$ considered in the simple example above, and differential cryptanalysis is not the only family of attacks that needs to be taken into account. As a remark, we note that, since the f' function is structurally similar to what is done in the NORX authenticated encryption scheme [54], but in the additive field, it is quite possible that analyses of this latter scheme can also apply to the construction hereby described. Providing such analysis remains, however, as a matter of future work.

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

Another feature of Lyra2 is that its memory matrix was designed to allow legitimate users to take advantage of memory hierarchy features, such as caching and prefetching. As observed in [5], such mechanisms usually make access to consecutive memory locations in real-world machines much faster than accesses to random positions, even for memory chips classified as "random access". As a result, a memory matrix having a small R is likely to be visited faster than a matrix having a small C, even for identical values of $R \cdot C$. 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 4 of Algorithm 2). This avoids situations in which a careless implementation ends up leaving *pwd* in the device's volatile memory or, worse, leading to its storage in non-volatile memory due to memory swaps performed during the algorithm's memory-expensive phases. Hence, it meets the general guideline of purging private information from memory as soon as it is not needed anymore, preventing that information's recovery in case of unauthorized access to the device [55,56].

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 pa-

rallel. 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* [57], 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 of Algorithm 1), 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 $M[0]$–$M[2]$, those three rows 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 $M[4]$, the value to be employed when computing rows that depend on it (e.g., $M[8]$) cannot be obtained directly from the password. Instead, recomputing the updated value of $M[0]$ requires (a) running the Setup phase until the point it was last modified (e.g., for the value required by $M[8]$, this corresponds to when $M[4]$ was initialized) or (b) using some rows still available in memory, XORing them together to obtain the values of $rand[col]$ that modified $M[0]$ since its initialization.

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 [58,59,60] as done in [57,61]) 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 13—17 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 $i \geqslant 3$, this corresponds to the state in line 13 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 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 iteration $R \cdot (\tau - 1) + i$ of the Visitation Loop (with $1 \leqslant \tau \leqslant T$), 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 given by the Visitation Loop's iteration $R \cdot (\tau - 1) + i$. In this manner, the τ symbol is employed to indicate how many times the Wandering phase performs a number of duplexing operations equivalent to that in the Setup phase.

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, $M[1_3]$ corresponds to row $M[1]$ after its cells are XORed with rot(*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 iteration $R \cdot (\tau - 1) + i$ of the Wandering phase's Visitation Loop, for $\tau \geqslant 1$.

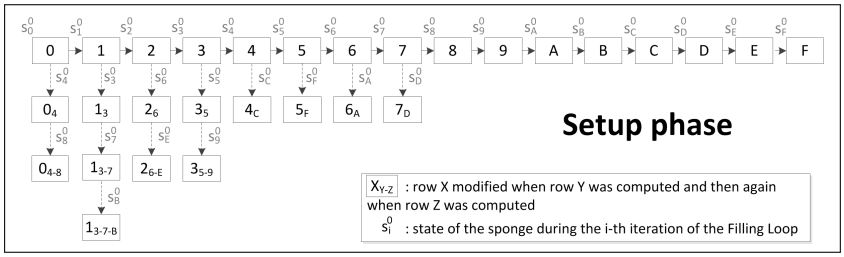


Fig. 5: The Setup phase.

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 5, 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 6 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 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. □

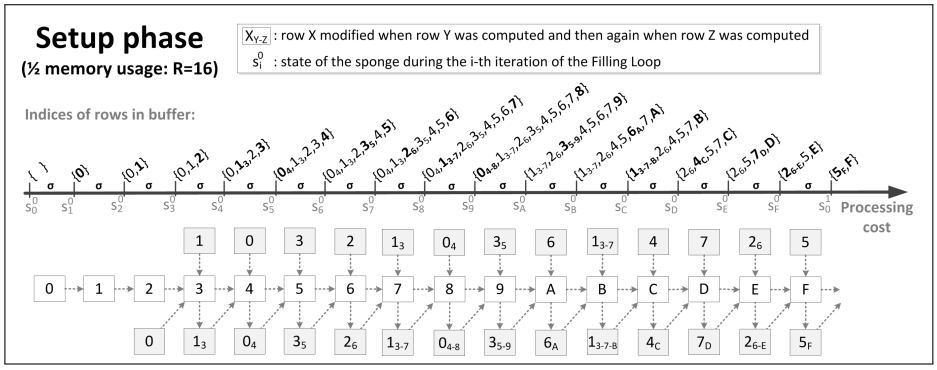


Fig. 6: Attacking the Setup phase: storing 1/2 of all rows. The most recently modified rows in each iteration are marked in bold.

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, 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 7. 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, but 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 7). 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 σ. Alternatively, $M[4]$ could have been kept in memory after

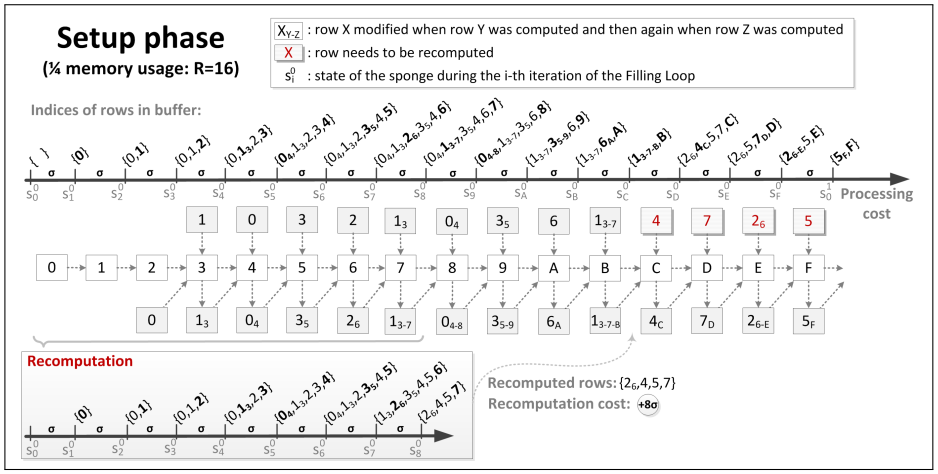


Fig. 7: Attacking the Setup phase: storing 1/4 of all rows. The most recently modified rows in each iteration are marked in bold.

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 7 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, 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): $g_1 = \{M[0_{4-8}], M[9], M[2_{6-E}], M[B]\}$, $g_2 = \{M[4_C], M[D], M[6_A], M[F]\}$, $g_3 = \{M[8], M[1_{3-7-B}], M[A], M[3_{5-9}]\}$, 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 8, 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 $M[0_{4-8}]$ 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.

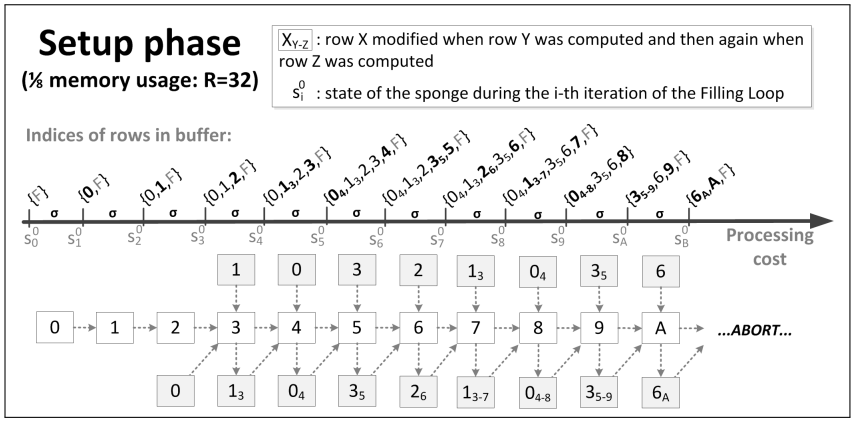


Fig. 8: Attacking the Setup phase: recomputing $M[6_A]$ while storing 1/8 of all rows and keeping $M[F]$ in memory. The most recently modified rows in each iteration are marked in bold.

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

$$CostSetup_0(m) = 3m/2 \quad \triangleright\ 1/4 \text{ memory usage scenario } (n = 0)$$

$$CostSetup_n(m) = \overbrace{CostSetup_{n-1}(m/2)}^{V_1^0} + \overbrace{m/2}^{V_2^0} + \overbrace{\underbrace{(3 \cdot CostSetup_{n-1}(m/2)/4)}_{\text{approximate cost of each execution}} \cdot \underbrace{(2^{n+1})}_{\text{number of executions}}}^{\text{Re-executions of } V_1^0} \tag{1}$$

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

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

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):

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

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

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

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 an average processing cost $CostSetup_n(R)$ that is 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 $M[1]$. Similarly, initializing a cell of $M[4]$ takes one cell from $M[0]$, as well as one from $M[1]$ and up to two cells from $M[3]$ (one because $M[3]$ is itself fed to the sponge and another required to the computation of $M[1_3]$).

An attack that computes only one cell at a time would be easy to build if the cells sequentially output by the sponge during the 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 9

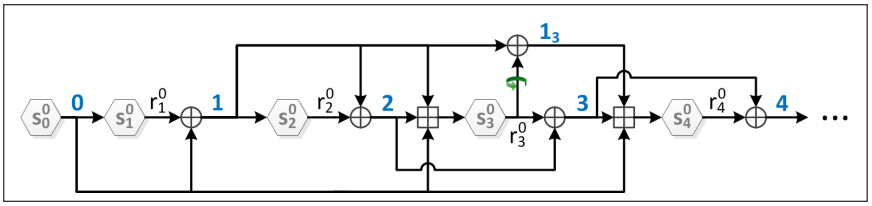


Fig. 9: Attacking the Setup phase: storing only sponge states.

to compute cells as they are required. For example, one could compute $M[2][0]$ in this scenario with (1) states $s^0_{0,0}$, $s^0_{1,0}$ and $s^0_{2,0}$, and (2) two b-long buffers, one for $M[0][0]$ so it can be used for computing $M[1][0]$, and the other for storing $M[1][0]$ itself, used as input 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 $M[3][col]$); 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 10. 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 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 σ to compute $M[0][0]$ from $s^0_{0,0}$. A similar issue would occur right after that, when initializing $M[2][C-2]$ from $M[1][1]$: 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

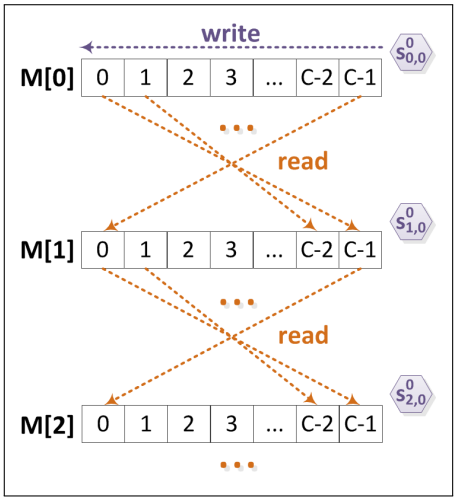


Fig. 10: Reading and writing cells in the Setup phase.

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 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 "σ" 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: consumer-producer strategy. 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 11, in which all rows are modified only once during every $R/2$ iterations 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

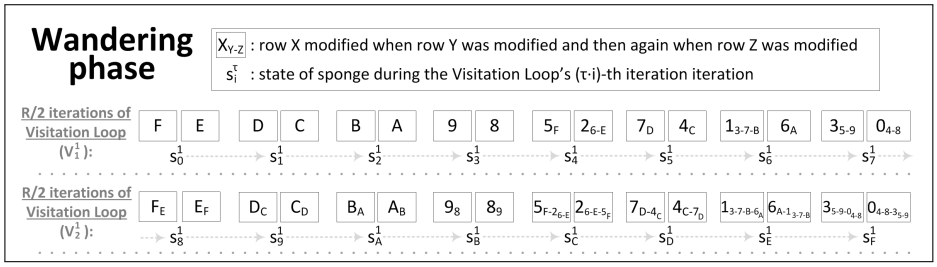


Fig. 11: An example of the Wandering phase's execution.

principle to V_2^1, 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 11, we make the visitation order in V_1^1 be the exact opposite of the initialization/update of rows during V_2^0, while in V_2^1 the order is the 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 $R/2$ iterations of the Wandering phase with $1/2$ memory usage. 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 11, for example, $M[5_F]$ can be recovered by computing $M[5]$ and then making $M[5_F][col] = M[5][col] \oplus \text{rot}(r^0_F[col])$, 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 $(3R/4)\sigma$ 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^0_2, then each row recovered can be used by V^1_1 before being discarded. In principle, a very lucky attacker could then be able to run the entire V^1_1 by executing 3/4 of the Setup only once. Assuming for simplicity that the $(R/2)\sigma$ average models a more usual scenario, the cost of each of the $R/2$ iterations of V^1_1 can be estimated as: 1 in 1/4 of these iterations, when row^0 and row^1 are both in memory; and roughly $(R/2)\sigma$ in 3/4 of its iterations, when one or a pair of rows need to be recovered. The total cost of V^1_1 becomes, thus, $((1/4) \cdot (R/2) + (3/4) \cdot (R/2) \cdot (R/2))\sigma \approx (3R^2/16)\sigma$.

After that, when V^1_2 is reached, the situation is different from what happens in V^1_1: since the rows required for any iteration of V^1_2 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 11, the rows required for iteration $i = 8$ of the Visitation Loop besides $M[prev^0] = M[A]$ and $M[prev^1] = 9$ are $M[8_{1_{3-7-B}}]$ 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^0_2 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^1_0), 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^0_2 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^0_2. The second extreme occurs when V^1_1 takes too long to start consuming

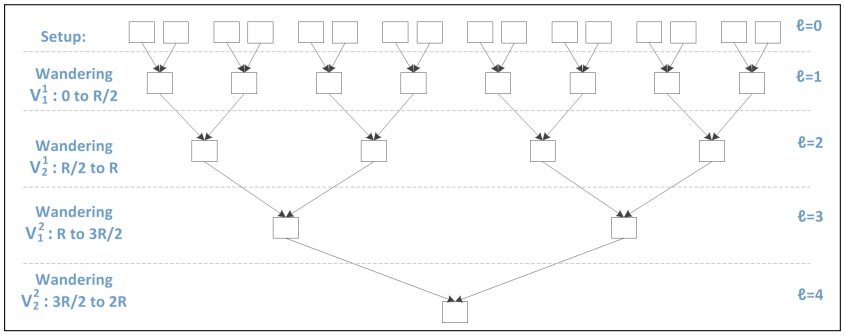


Fig. 12: Tree representing the dependence among rows in Lyra2.

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 11. Then, 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 $(3R/2)\sigma$.

From these observations, 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 cost of V_1^1's internal iterations and the second to the overhead incurred by the underlying Setup 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: σ whenever both $M[row^0]$ and $M[row^1]$ are available, which happens in $1/4$ of these iterations; roughly $R\sigma$ whenever $M[row^0]$ and/or $M[row^1]$ need to be recomputed, so for $1/4$ of these iterations. This leads to a total cost of $(R/8+3R^2/8)\sigma$ for V_2^1. 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 then be estimated as $R\sigma + (3R^2/16)\sigma + (R/8 + 3R^2/8)\sigma \approx (3R/4)^2\sigma$ for this strategy.

5.1.3.2 The whole Wandering phase with $1/2$ *memory usage.* Generalizing the discussion for all iterations of the Wandering phase, 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 a dependence graph in the form of an inverted tree as depicted in Figure 12, level $\ell = 0$ corresponding to the Setup phase and each $R/2$ iterations of the Visitation Loop raising the tree's depth by one. Hence, the full execution of any level $\ell > 0$ 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):

$$CostWander^*_\ell = \overbrace{\underbrace{(1/4)(R/2)}_{\substack{25\% \text{ of} \\ \text{iterations}}} \cdot 1}^{\substack{\text{no re-execution of} \\ \text{previous levels}}} + \overbrace{\underbrace{(3/4)(R/2)}_{\substack{75\% \text{ of} \\ \text{iterations}}} \cdot CostWander_{\ell-1}/2}^{\substack{\text{re-executions of} \\ \text{previous levels}}} \quad (3)$$

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 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:

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

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:

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

5.1.3.3 The whole Wandering phase 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

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

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

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

for the deterministic re-executions of level ℓ.

Equations 6 and 7 can then be combined to provide the following estimate to the total cost of an attack against Lyra2 involving $R/2^{n+2}$ rows instead of R:

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

Since, as suggested in Section 5.1.2.4, the upper bound $CostSetup_n = \mathcal{O}(R^{n+1})$ given by Equation 2 is likely to become a better estimate for $CostSetup_n$ as n grows, we conjecture that the processing cost of Lyra2 using the strategy hereby discussed be $\mathcal{O}(2^{2nT}R^{n+2})$ for $n \gg 1$.

5.1.4 Adding the Wandering phase: sentinel-based strategy. The analysis of the consumer-producer strategy described in Section 5.1.3 shows that updating many rows in the hope they will be useful in an iteration of the Wandering phase's Rows Loop does reduce the attack cost by too much, since these rows are only useful 25% of the time; in addition, it has the disadvantage of discarding the rows initialized/updated during $VLoop10$, which are certainly required 75% of the time. From these observations, we can consider an alternative strategy that employs 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 $M[0_4] - M[7]$ 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 σ, 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, suppose 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 $[R/2, R[$. Then, the cost of recovering any row output by V_2^0 would range from 0 (for the sentinels themselves) to $(R/2\epsilon)\sigma$ (for rows the farthest away from the sentinels), or $(R/4\epsilon)\sigma$ on average. The resulting memory cost of such strategy is approximately $R/2$ (for the rows from V_1^0), plus 2ϵ (for the fixed sentinels), plus 2 (for

[1] This is analogous to the attack presented in [62] for the version of Lyra2 originally submitted to the Password Hashing Competition as "V1"

storing the value of $prev^0$ and $prev^1$ while computing a given row inside the area covered by a fixed sentinel). When compared with the consumer-produces approach, one drawback is that only the 2ϵ rows acting as sentinels can be promptly consumed by V_1^1, since rows provided by V_1^0 are overwritten during the execution of V_2^0. Nonetheless, the average cost of V_1^1 ends up being approximately $(R/2) \cdot (R/4\epsilon)\sigma$ for a small ϵ, which is lower than in the previous approach for $\epsilon \geqslant 2$. With $\epsilon = R/32$ sentinels (i.e., $R/16$ rows), for example, the processing cost of V_1^1 would be $4R$ for a memory usage less than 10% above $R/2$.

We can then 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 the latter's recomputations. For instance, $M[9_8]$ and $M[8_9]$ could be used as sentinels to accelerate the recovery of rows visited in the second half of V_1^1 in our example scenario (see Figure 11). However, in this case the sentinels are likely to be less effective. The reason is that the steps taken from each sentinel placed in V_1^1 should cover different portions of V_2^0, obliging some iterations of V_2^0 to be executed. For example, using the same $\epsilon = R/32$ sentinels as before to keep the memory usage near $R/2$, we could distribute half of them along V_2^0 and the other half along V_1^1, so each would be covered by $\epsilon' = \epsilon/2$ sentinels. As a result, any row output by V_1^1 or V_2^0 could be recovered with $R/4(\epsilon') = 16$ executions of CL on average. Unfortunately for the attacker, though, any iteration of V_2^1 takes two rows from V_1^1, which means that $2 \cdot 16 = 32$ iterations of V_1^1 are likely to be executed and, hence, that roughly $2 \cdot 32 = 64$ rows from V_2^0 should be required. If all of those 64 rows fall into areas covered by different sentinels placed at V_2^0, the average cost when computing any row from V_2^1 would be approximately $64 \cdot 16 = 1024$ executions of CL. In this case, the cost of the $R/2$ iterations of V_2^1 would become roughly $(1024R/2)\sigma$ on average. This is lower than the $\approx (R^2/2)\sigma$ obtained with the consumer-producer strategy for $R > 1024$, but still orders of magnitude more expensive than a regular execution with a memory usage of R.

Obviously, two or more of the 64 rows required from V_2^0 may fall in the area covered by a same sentinel, which allows for a lower number of executions if the attacker computes those rows in a single sweep and keep them in memory until they are required. Even though this approach is likely to raise the attack's memory usage, it would lead to a lower processing cost, since any part of V_2^0 covered by a same sentinel would be run only once during any iteration of V_2^1. However, if the number of sentinels in V_2^0 is large in comparison with the number of rows required by each of V_2^1's iteration (i.e., for $\epsilon/2 \gg 64$, which implies $R \gg 8192$), we can ignore such "sentinel collisions" and the average cost described above should hold. This should also the cost obtained if the attacker prefers not to raise the attack's memory usage when collisions occur, but instead recomputes rows that can be obtained from a given sentinel by running the same part of V_2^0 more than once.

For the sake of completeness, it is interesting to analyze such memory-processing tradeoffs for dealing with collisions when the cost of this sentinel-based strategy starts to get higher than the one obtained with the consumer-producer strategy. Specifically, for $R = 1024$ this strategy is deemed to create

many sentinel collisions, with each of the $\epsilon' = 16$ sentinels placed along V_2^0 being employed for recomputing roughly $64/16 = 4$ out of the 64 rows from V_2^0 required by each iteration of V_2^1. In this scenario, the 4 rows under a same sentinel's responsibility can recovered in a single sweep and then stored until needed. Assuming that those 4 rows are equally distributed over the corresponding sentinel's coverage area, the average cost of the executions related to that sentinel would then be $(7/8)(R/2)/(\epsilon/2) = 28\sigma$. This leads to $16 \cdot 28\sigma = 448\sigma$ for all 16 partial runs of V_2^0, and consequently to $(448R/2)\sigma$ for the whole V_2^1. In terms of memory usage, the worst case scenario from the attacker's perspective refers to when the rows computed last from each sentinel are the first ones required during V_2^1, meaning that recovering 1 row that is immediately useful leaves in memory 3 that are not. This situation would lead to a storage of $3(\epsilon/2) = 3R/64$ rows, which corresponds to 75% of the $R/16$ rows already employed by the attack besides the $R/2$ base value.

As a last remark, notice that the 64 rows from V_2^0 can be all recovered in parallel, using 64 different processing cores, the same applying to the 2 rows from V_1^1, with 2 extra cores. The average cost of V_2^1 as perceived by the attacker would then be roughly $(16 + 16)(R/2)\sigma$, which corresponds to a parallel execution of V_2^0 followed by a parallel execution of V_1^1. In this case, however, the memory usage would also be slightly higher: since each of the 66 threads would have to be associated its own $prev^0$ and $prev^1$, the attack would require an additional memory usage of 132 rows.

5.1.4.1 On the (low) scalability of the sentinel-based strategy. Even though the sentinel strategy shows promise in some scenarios, it has low scalability for values of T higher than 1. The reason is that, as T grows, the computation of any given row depends on rows recomputed from an exponentially large number of sentinels. This is more easily observed if we analyze the dependence graph depicted in Figure 13 for $T = 2$, which shows the number of rows from level $\ell - 1$ that are needed in the sentinel-based computation of level ℓ. In this scenario, if we assume that the ϵ sentinels are distributed along V_2^0, V_1^1, V_2^1 and V_1^2 (levels $\ell = 0$ to 3, respectively), each level will get $\epsilon' = \epsilon/4$ sentinels, being divided in $R/2\epsilon'$ areas. As a result, even though computing a row from level $\ell = 4$ takes only 2 rows from level $\ell = 3$, computing a row from level $\ell < 4$ involves roughly $R/4\epsilon'$ iterations of that level, those iterations requiring $2(R/4\epsilon')$ rows from level $\ell - 1$. Therefore, any iteration of V_2^2 is expected to involve the computation of $2^4(R/4\epsilon')^3$ rows from V_2^0, which translates to 2^{19} rows for $\epsilon = R/32$. If each of these rows is computed individually, with the usual cost of $(R/4\epsilon')\sigma$ per row, the recomputations related to sentinels from V_2^0 alone would take $2^{19}(R/4\epsilon')\sigma = 2^{24} \cdot \sigma$, leading to a cost higher than $(2^{24} \cdot R/2)\sigma$ for the whole V_2^2.

More generally, for arbitrary values of T and $\epsilon = R/\alpha$ (and, hence, $\epsilon' = \epsilon/2T$), the recomputations in V_2^0 for each iteration of V_2^T would take $2^{2T} \cdot (R/4\epsilon')^{2T}\sigma$, so the cost of V_2^T itself would become $(\alpha \cdot T)^{2T}(R/2)\sigma$. Depending on the parameters employed, this cost may be higher than the $\mathcal{O}((3/2)^{2T}R^2)$ obtained with the consumer-producer strategy, making the latter a preferred attack venue. This

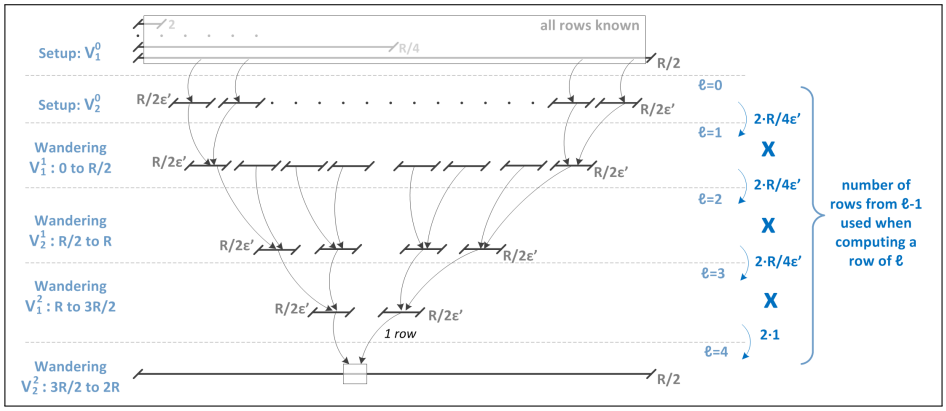


Fig. 13: Tree representing the dependence among rows in Lyra2 with $T = 2$: using ϵ' sentinels per level.

is the case, for example, when we have $\alpha = 32$, as in all previous examples, $R \leqslant 2^{20}$, as in all benchmarks presented in Section 7, and $T \geqslant 2$.

Once again, attackers may counterbalance this processing cost with the temporary storage of rows that can be recomputed from a same sentinel, or of a same row that is required multiple times during the attack. However, the attackers' ability of doing so while keeping the memory usage around $R/2$ is limited by the fact that this sentinel-based strategy commits a huge part of the attack's memory budget to the storage of all rows from V_1^0. Diverting part of this budget to the temporary storage of rows, on the other hand, is similar to what is done in the consumer-producer strategy itself, so the latter can be seen as an extreme case of this approach.

On the other extreme, the memory budget could be diverted to raise the number of sentinels and, thus, reduce α. As a drawback, the attack would have to deal with a dependence graph displaying extra layers, since then V_1^0 would not be fully covered. This would lead to a higher cost for the computation of each row from V_2^0, counterbalancing to some extent the gains obtained with the extra sentinels. For example, suppose the attacker (1) stores only $R/4$ out of the $R/2$ rows from V_1^0, using the remainder budget of $R/4$ rows to make $\epsilon = R/8$ sentinels, and then (2) places $\epsilon^* = R/32$ sentinels (i.e., $R/16$ rows) along the part of V_1^0 that is not covered anymore, thus keeping the total memory usage at $R/2 + R/16$ rows as in the previous examples. In this scenario, the number of rows from V_2^0 involved in each iteration of V_2^2 should drop to $2^4(R/4\epsilon')^3 = 2^{13}$ if we assume once again that the sentinels are equally distributed through all levels (i.e., for $\epsilon' = \epsilon/4$). However, recovering a row from V_2^0 should not take only $R/4\epsilon' = 2^3$ executions of CL anymore, but roughly $(R/4\epsilon') \cdot (R/4\epsilon^*) = 2^5$ due to the recomputations of rows from V_1^0. The processing cost for the whole V_2^2 would then be $(2^{18} \cdot R/2)\sigma$, which still is not lower than what is obtained with the consumer-producer strategy for $R \leqslant 2^{17}$.

The low scalability of the sentinel-based strategy also impairs attacks with a memory usage lower than $R/2$, since then the number of sentinels and coverage of rows from V_1^0 would both drop. The same scalability issues apply to attempts of recovering all rows from V_2^0 in parallel using different processing cores, as suggested at the end of Section 5.1.4, given that the number of cores grows exponentially with T.

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 during the Wandering phase. 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 prefetching, even when the attacker employs (low-cost) random-access memory chips. Even though this latency may be (partially) hidden in a parallel attack by prefetching the rows needed by one thread while another thread is running, at least the attacker would have to pay the cost of frequently changing the context of each thread. We notice that this approach is particularly harmful against older model GPUs, whose internal structure were usually optimized toward deterministic memory accesses to small portions of memory [22, Sec. 5.3.2].

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, however, 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 those 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) [57,63]. This class of attacks has been shown to be effective, for example, against certain implementations of the Advanced Encryption Standard (AES) [64] and RSA [65], allowing the recovery of the secret key employed by the algorithms [63,66].

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 [57]. 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, it is quite possible they will 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 an interesting approach 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 [67]. This happens because such environments potentially create the required conditions for making cache-timing measurements [67], 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 this threat. 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 preventing 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 [57], according to which the server sends only the salt to the client (preferably using a secure channel), who responds with $x = \mathrm{PHS}(pwd, salt)$; then, the server only computes locally $y = H(x)$ and compares it to the value stored in its own database. The result of this 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 [68], recent advances in software and hardware technology may (partially) 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 [67] 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 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); and taking advantage of parallelism capabilities potentially available on the legitimate user's platform (parameter p).

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 security consequences. 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 5.1.3.2. 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 $row^{d \geqslant 2}$ that force the attacker to keep rows that, otherwise, could be discarded right after the middle of the Setup phase. One possible approach is, for example, to divide the revisitation window in the Setup phase into δ contiguous sub-windows, so each row^d revisits its own sub-window δ times. We note that this principle does not even need to be restricted to reads/writes on a same memory matrix: for example, one could add a row^2 variable that indexes a Read-Only Memory chip attached to the device's platform and then only perform several reads (no writes) on this external memory, giving support to the "rom-port-hardness" concept discussed in [69].

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 in the platforms used as testbed for our

benchmarks (see Section 7). Therefore, the interest of supporting a customizable δ depends on actual tests made on the target platform, although we conjecture that this would only be beneficial with DRAM chips faster than those commercially available today. 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 in Section 6.2, while its application for obtaining rom-port-hardness is not discussed.

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 multicore CPU or even a GPU. 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. Specifically, 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 lower need of synchronism between threads, facilitating the algorithm's processing by parallel platforms. The second property, on its turn, makes it harder to run each thread separately with a reduced memory usage and simply combine their final results together.

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

Algorithm 3 The Lyra2 Algorithm, with p parallel instances.

PARAM: H ▷ Sponge with block size b (in bits) and underlying permutation f
PARAM: ρ ▷ Number of rounds of f during the Setup and Wandering phases
PARAM: ω ▷ Number of bits to be used in rotations (recommended: a multiple of W)
PARAM: p ▷ Degree of parallelism ($p \geqslant 1$ and $p|(R/2)$)
INPUT: pwd ▷ The password
INPUT: $salt$ ▷ A salt
INPUT: T ▷ Time cost, in number of iterations
INPUT: R ▷ Number of rows in the memory matrix
INPUT: C ▷ Number of columns in the memory matrix (recommended: $C \cdot \rho \geqslant \rho_{max}$)
INPUT: k ▷ The desired key length, in bits
OUTPUT: K ▷ The password-derived k-long key

1: **for each** i in $[0, p[$ **do** ▷ Operations performed in parallel, by each thread
2: ▷ **Bootstrapping phase**: Initializes the sponges' states and local variables
3: $params \leftarrow len(k) \parallel len(pwd) \parallel len(salt) \parallel T \parallel R \parallel C \parallel p \parallel i$
4: $H_i.absorb(\texttt{pad}(pwd \parallel salt \parallel params))$ ▷ Padding rule: 10^*1.
5: $gap \leftarrow 1$; $stp \leftarrow 1$; $wnd \leftarrow 2$; $sqrt \leftarrow 2$; $sync \leftarrow 4$; $j \leftarrow i$
6: $prev^0 \leftarrow 2$; $row_p \leftarrow 1$; $prev_p \leftarrow 0$
7: ▷ **Setup phase**: Group of threads initialize a $(R \times C)$ memory matrix
8: **for** $(col \leftarrow 0$ **to** $C-1)$ **do** $\{M_i[0][C-1-col] \leftarrow H_i.squeeze_\rho(b)\}$ **end for**
9: **for** $(col \leftarrow 0$ **to** $C-1)$ **do** $\{M_i[1][C-1-col] \leftarrow M_i[0][col] \oplus H_i.duplex_\rho(M_i[0][col], b)\}$ **end for**
10: **for** $(col \leftarrow 0$ **to** $C-1)$ **do** $\{M_i[2][C-1-col] \leftarrow M_i[1][col] \oplus H_i.duplex_\rho(M_i[1][col], b)\}$ **end for**
11: **for** $(row^0 \leftarrow 3$ **to** $R/p-1)$ **do** ▷ **Filling Loop**: initializes remainder rows
12: ▷ **Columns Loop**: $M_i[row^0]$ is initialized; $M_j[row_p]$ is updated
13: **for** $(col \leftarrow 0$ **to** $C-1)$ **do**
14: $rand \leftarrow H_i.duplex_\rho(M_j[row_p][col] \boxplus M_i[prev^0][col] \boxplus M_j[prev_p][col], b)$
15: $M_i[row^0][C-1-col] \leftarrow M_i[prev^0][col] \oplus rand$
16: $M_j[row_p][col] \leftarrow M_j[row_p][col] \oplus \text{rot}(rand)$ ▷ rot(): right rotation by ω bits
17: **end for**
18: $prev^0 \leftarrow row^0$; $prev_p \leftarrow row_p$; $row_p \leftarrow (row_p + stp) \bmod wnd$
19: **if** $(row_p = 0)$ **then** ▷ Window fully revisited
20: $wnd \leftarrow 2 \cdot wnd$; $stp \leftarrow sqrt + gap$; $gap \leftarrow -gap$ ▷ Updates window and step
21: **if** $(gap = -1)$ **then** $\{sqrt \leftarrow 2 \cdot sqrt\}$ **end if** ▷ Doubles $sqrt$ every other iteration
22: **end if**
23: **if** $(row^0 = sync)$ **then** $\{sync \leftarrow sync + sqrt/2$; $j \leftarrow (j+1) \bmod p$; $SyncThreads\}$ **end if**
24: **end for**
25: $SyncThreads$
26: ▷ **Wandering phase**: Iteratively overwrites (random) cells of the memory matrix
27: $wnd \leftarrow R/2p$; $sync \leftarrow sqrt$; $\texttt{off}^0 \leftarrow 0$; $\texttt{off}_p \leftarrow wnd$
28: **for** $(wCount \leftarrow 0$ **to** $(R \cdot T)/p - 1)$ **do**
29: $row^0 \leftarrow \texttt{off}^0 + (\text{lsw}(rand) \bmod wnd)$; $row_p \leftarrow \texttt{off}_p + (\text{lsw}(\text{rot}(rand)) \bmod wnd)$
30: $j \leftarrow \text{lsw}(\text{rot}^2(rand)) \bmod p$
31: **for** $(col \leftarrow 0$ **to** $C-1)$ **do** ▷ **Columns Loop**: updates $M_i[row^0]$
32: $col^0 \leftarrow \text{lsw}(\text{rot}^3(rand)) \bmod C$ ▷ Picks pseudorandom column from $M_i[prev^0]$
33: $rand \leftarrow H_i.duplex_\rho(M_i[row^0][col] \boxplus M_i[prev^0][col^0] \boxplus M_j[row_p][col])$
34: $M_i[row^0][col] \leftarrow M_i[row^0][col] \oplus rand$ ▷ Updates row picked from slice M_i
35: **end for** ▷ End of Columns Loop
36: $prev^0 \leftarrow row^0$ ▷ Next iteration revisits most recently updated row from slice M_i
37: **if** $(wCount = sync)$ **then** $\{sync \leftarrow sync + sqrt$; $\text{swap}(\texttt{off}^0, \texttt{off}_p)$; $SyncThreads\}$ **end if**
38: **end for** ▷ End of Visitation Loop
39: $SyncThreads$
40: ▷ **Wrap-up phase**: output computation
41: $H_i.absorb(M_i[row^0][0])$ ▷ Absorbs a final column with full-round sponge
42: $K_i \leftarrow H_i.squeeze(k)$ ▷ Squeezes k bits with full-round sponge
43: **end for** ▷ All threads finished
44: **return** $K_0 \oplus \ldots \oplus K_{p-1}$ ▷ Provides k-long bitstring as output

6.2.1 Structure and rationale Lyra2$_p$'s steps are shown in Algorithm 3. First, during the Bootstrapping phase, p sponge copies are generated. This is done similarly to Lyra2, the main difference being that the *params* fed to each sponge S_i ($0 \leqslant i \leqslant p-1$) must contain the values of p and i in addition to any other information already included in line 3 of Algorithm 3. 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 input 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.

For the Setup phase, the p sponges are then evenly distributed over the memory matrix, becoming responsible for initializing p contiguous slices of R/p rows each, the said slices being 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 $M_i[x] = M[i \cdot R/p + x]$ for any given value of x.

The Setup phase of each sponge S_i then proceeds similarly to algorithm's non-parallelizable version, starting with the three first rows and then entering the Filling Loop to initialize the remainder rows while revisiting previously initialized rows; the latter are denoted row_p in Algorithm 3, which play the exact same role as row^1 in Algorithm 2 during the Setup phase. However, Lyra2$_p$ has one important difference: in each duplexing operation performed by S_i, the revisited rows are not necessarily picked from slice M_i, but from a slice M_j that changes often during the Visitation Loop. Namely, the value of j starts at i (line 5) and is cyclically incremented whenever S_i revisits approximately $\sqrt{wnd}$ rows from the corresponding window (line 23). This approach ensures that each slice depends on data from other slices, enforcing the need of keeping all of their corresponding data in memory for better performance. This specific choice of how often j is updated, on its turn, was motivated by the fact that it builds upon the Setup's window visitation pattern to distribute those visitations among the different slices: if we see the window as a matrix, as discussed in Section 4.1.2, each p consecutive visitations of its diagonals and anti-diagonals happen in p different slices.

To prevent race conditions that might be caused by the Setup's cross-slice read/write operations, the execution of all threads is synchronized in line 23, which is indicated by the "*SyncThreads*" call. A final synchronization is also performed right after the end of the Setup phase (line 25), ensuring that all rows are initialized before the algorithm enters the Wandering phase. These synchronization points are enough to ensure that each thread's $prev^0$, $prev_p$ and row_p variables cover separate memory areas, so the threads can run independently until those points without the risk of inconsistencies.

As a final remark regarding the Setup phase, we note that the $M_j[prev_p]$, fed to S_i in line 14, certainly does not come from that sponge's cache right after j is updated, but actually corresponds to the row most recently updated by another

sponge. This should impact the algorithm's performance, but since this situation does not occur too often (approximately $\mathcal{O}(\lg(R/p) \cdot \sqrt{(R/p)})$ times), in practice the total impact of such cache misses should be low, which was confirmed by our experimental results.

Concerning the Wandering phase, an important difference between the non-parallelizable and paralellizable versions of Lyra2 is that in the latter each slice M_i is seen by the sponge S_i as two halves: one half is visited by S_i itself, in the positions indicated by the pseudorandomly picked index row^0; the other half, however, is meant to be freely visited by any sponge $S_{0 \leqslant j < p}$, in the positions indicated by the pseudorandomly picked index row_p. This separation between halves is accomplished by (1) fixing the wnd variable to $R/2p$ in line 27, which limits the range of the row^0 and row_p indices computed in line 30 to a half slice, and (2) combining row^0 and row_p with complementary offsets ($\texttt{off}^0$ and $\texttt{off}_p$, respectively) in line 30, before feeding them to the sponge. The pseudorandom value of j is then computed similarly to row^0 and row_p, from a word of the sponge's outer state (also in line 30). Analogously to the Setup, this makes the each slice dependent on data from other slices, penalizing attackers that might prefer to discard part of the data. However, since the visitation pattern during the Wandering phase is unpredictable, each S_i refrains from writing on the row taken from slice M_j, which is only read, as a way to prevent race conditions that could emerge from such cross-slice interactions. As a result, each iteration of the Visitation Loop updates a single row from M_i with the sponge's output, namely row^0 (line 34), while row_p remains unmodified; for this reason, there is no "$prev_p$" in this part of the algorithm, so the duplexing operation in line 33 takes as input three rows rather than four.

To ensure that the updates made by S_i on its own half slice affect the other parallel threads reading from the other half, these two halves are switched after approximately $\sqrt{R/p}$ iterations of the Visitation Loop (line 37), at which moment all threads are synchronized. This switching frequency is consonant with the one adopted during Setup, besides leading to a curious property: following to the Birthday Paradox, there is a $\approx 50\%$ chance that at least one row updated by S_i while processing a half of its slice is read by one of the p sponges when they all access that same half, i.e., after the subsequent switch. Therefore, even though each thread may be run independently of any other thread between synchronizations, it would be error prone to run a single thread beyond the synchronization point if other threads have not yet finished their own processing.

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 XORed together, yielding then Lyra2$_p$'s output K.

6.2.2 Security analysis The main advantage of this parallelizable version of Lyra2 is that, in theory, it allows legitimated users to process the memory matrix 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, as well as to the need of eventual synchronizations among threads. However, for the sake of the argument, consider that p is indeed the acceleration obtained. In this case, there are some ways by which legitimate users may take advantage of this faster operation for raising the algorithm's resistance against attacks. On one extreme, legitimate users may adopt as parameters $R_p = R \cdot p$ and $T_p = T$, which raises the algorithm's memory usage p times while maintaining a similar processing time. On the other extreme, legitimate users may use the multiple processing cores simply to raise the algorithm's total number of operations and bandwidth usage, without raising its processing time, which is accomplished by making $R_p = R$ and $T_p = T \cdot p$.

Whichever the parameterization adopted, performing a low-memory attack against the Setup phase of Lyra2$_p$ is expected to involve costs similar to those discussed in Section 5.1.2. The reason is that each thread of Lyra2$_p$ initializes and revisits rows during Setup just like Lyra2, the only significant difference being that among the rows fed to a given sponge S_i there are some initialized/updated by other sponges $S_{j \neq i}$ running in different threads. These cross-slice interactions oblige all threads to run approximately in sync, filling the memory with newly initialized rows, to allow other threads to proceed their computation. This need of synchronization comes especially from the fact that the rows revisited by S_i on every slice $M_{0 \leqslant j < p}$ are distributed all along that slice, including rows with low and high indices. Consequently, cross-slice reads by S_i on M_j following a given synchronization point can only be performed after S_j is near that same synchronization point, because otherwise (at least) the rows with higher indices will not be available. The group of p threads can, thus, be seen approximately as a single thread that sequentially initializes and updates p rows at a time, much like in the non-parallelizable version of the Lyra2 algorithm. Hence, running the Setup phase with a peak memory usage of $R_p/2$ rows and no processing penalty, for example, is still perfectly possible: since only the first half of each slice is revisited during the initialization of their second halves, the rows from the latter still can be discarded right after their computation, similarly to the attack discussed in Section 5.1.2.1. Attacks going below $R_p/2$ rows, however, should involve the need of discarding rows and recomputing them only when needed, from scratch or using intermediary results as sentinels, with processing penalties that are likely similar to those presented in Section 5.1.2.

The Wandering phase, on its turn, has a disadvantage when compared to Lyra2's non-parallelizable version: as a single row is updated per thread in each iteration of the Visitation Loop rather than two, the resulting dependence graph gains extra levels only after R_p iterations of that loop. Since this is twice slower than assumed in the original analysis of the Wandering phase (Sections 5.1.3 and 5.1.4), the main impact of this difference is that the equations thereby described should apply to Lyra2$_p$ with the "$2T$" parameter replaced by "T_p". A $T_p \geqslant 2T$ parameterization could compensate for this correction, leading to a similar resistance against both low-memory attack venues discussed in those sections. Nonetheless, if there is enough space available at the legitimate platform, the

$R_p = R \cdot p$ and $T_p = T$ parameterization would still be preferable: with $R_p = R$ and $T_p = T \cdot p$, the memory usage of R would allow attackers to run p regular instances of Lyra2$_p$ in parallel, using a total of $R \cdot p$ rows, obtaining a performance penalty of p due to the higher value of T; in comparison, if we have $R_p = R \cdot p$, bringing the memory cost down to R, so p instances can be run in parallel with the same $R \cdot p$ rows, would involve a penalty higher than simply p.

Other differences of Lyra2$_p$'s Wandering phase should have only small impacts on its security when compared with Lyra2, not influencing too much the asymptotic costs discussed in Sections 5.1.3 and 5.1.4. For example, in Lyra2$_p$ the group of p threads performs p times more read operations on the memory matrix per iteration of the Visitation Loop, so discarded rows should be recomputed more frequently. This should not raise the cost of the consumer-producer strategy by much, since the costs given in Section 5.1.3 already consider that recomputations occur at least 75% of the time; the cost of the sentinel-based strategy, on the other hand, should raise at most p times due to the number p times higher of sentinels from level $\ell - 1$ activated by level ℓ.

Concerning slow-memory attacks, the main advantage of the parallelizable version of Lyra2 is that it raises the memory bandwidth usage proportionally to p. Namely, the bandwidth of the Setup phase is around p times higher, while the Wandering phase's grows up to $3p/4$ times due to the lower number of write operations per thread, as discussed above. Therefore, even if Lyra2 and Lyra2$_p$ are configured to run with the same amount of memory and processing time, the latter can impose performance penalties up to p times higher to attacks in which multiple threads performing passwords tests share a same memory bus, besides requiring more processing cores. To avoid dealing with such inconvenience, attackers might prefer to serialize the algorithm's execution, running each thread in sequence instead of doing the whole computation of a given password guess in parallel. However, this approach would itself lead to a processing cost p times higher due to the serialization.

Finally, the low- and slow-memory approaches could be combined to take advantage of the fact that each sponge pseudorandomly visits a space of $R_p/2 + R_p/p$ rows instead of R_p. Specifically, this property allows the $\approx \sqrt{R_p}$ iterations of the Wandering phase between two synchronizations points to be run without recomputations even if only the $R_p/2 + R_p/2p$ rows that are known to be required by the thread being executed are kept in (fast) memory. If the remainder $(p - 1)R_p/2p$ rows are placed in a secondary storage devices instead of discarded, the only penalties to be paid in this case would be the cost of serializing the algorithm's execution and the eventual latency due to the data transfers between the secondary and main memory devices. The benefits of this approach are, however, quite limited, since the p times higher processing cost resulting from the serialization is not compensated by an equivalent memory reduction: after all, each individual thread will still require $(R_p/2 + R_p/p) > R_p/p$ rows to remain in memory.

7 Performance for different settings

In our assessment of Lyra2's performance, we used an SSE-enabled implementation of Blake2b's compression function [39] as the underlying sponge's f function of Algorithm 2 (i.e., without any of the extensions described in Section 6) and Algorithm 3 (i.e., the parallel extension described in Section 6.2). According to our tests, using SSE (Streaming SIMD Extensions, where SIMD stands for Single Instruction, Multiple Data) instructions allow performance gains of 20% to 30% in comparison with non-SSE settings, so we only consider such optimized implementations in this document. One important note about this implementation is that, as discussed in Section 4.4, the least significant 512 bits of the sponge's state are set to zeros, while the remainder 512 bits are set to Blake2b's Initialization Vector. Also, to prevent the IV from being overwritten by user-defined data, the sponge's capacity c employed when absorbing the user's input (line 4 of Algorithm 2) is kept at 512 bits, but reduced to 256 bits in the remainder of the algorithm to allow a higher bitrate (namely, of 768 bits) during most of its execution. The implementations employed, as well as test vectors, are available at `www.lyra2.net`.

7.1 Benchmarks for Lyra2 without parallelism

The results obtained with a SSE-optimized single-core implementation of Lyra2 are illustrated in Figure 14. The results depicted correspond to the average

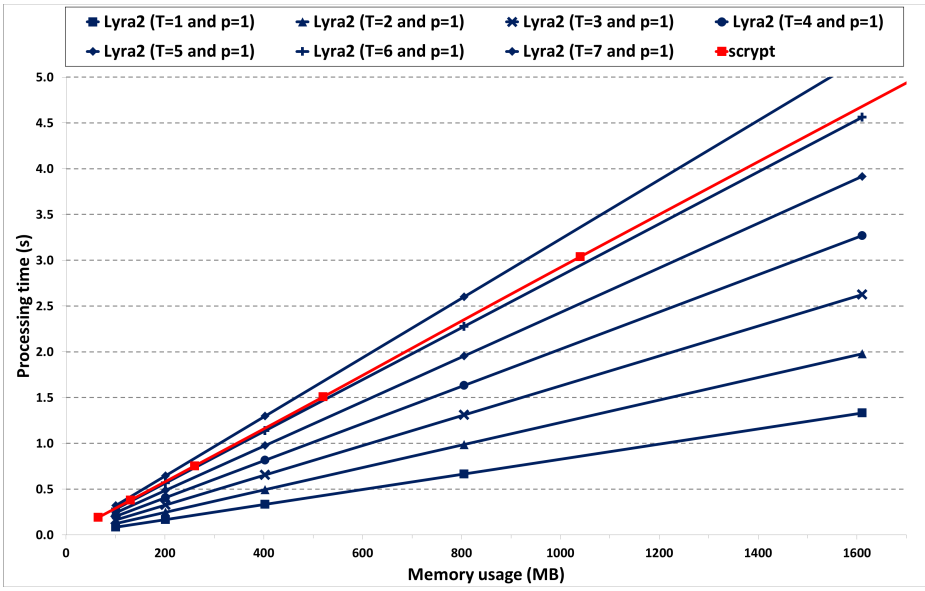


Fig. 14: Performance of SSE-enabled Lyra2, for $C = 256$, $\rho = 1$, $p = 1$, and different T and R settings, compared with SSE-enabled scrypt.

execution time of Lyra2 configured with $C = 256$, $\rho = 1$, $b = 768$ 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 able to execute in: less than 1 s while using up to 400 MB (with $R = 2^{14}$ and $T = 5$) or up to 1 GB of memory (with $R \approx 4.2 \cdot 10^4$ and $T = 1$); or in less than 5 s with 1.6 GB (with $R = 2^{16}$ and $T = 6$). All tests were performed on an Intel Xeon E5-2430 (2.20 GHz with 12 Cores, 64 bits) equipped with 48 GB of DRAM, running Ubuntu 14.04 LTS 64 bits. The source code was compiled using gcc 4.9.2.

The same Figure 14 also compares Lyra2 with the scrypt "SSE-enabled" implementation publicly available at `www.tarsnap.com/scrypt.html`, using the parameters suggested by scrypt's author in [5] (namely, $b = 8192$ and $p = 1$). The results obtained show that, to achieve a memory usage and processing time similar to that of scrypt, Lyra2 could be configured with $T \approx 6$.

We also performed tests aiming to compare the performance of Lyra2 and the other 5 memory-hard PHC finalists: Argon, battcrypt, Catena, POMELO, and yescrypt. Parameterizing each algorithm to ensure a fair comparison between them is not an obvious task, however, because the amount of resources taken by each PHS in a legitimate platform is a user-defined parameter chosen to influence the cost of brute-force guesses. Hence, ideally one would have to find the parameters for each algorithm that normalize the costs for *attackers*, for example in terms of energy and chip area in hardware, the cost of memory-processing trade-offs in software, or the throughput in highly parallel platforms such as GPUs. In the absence of a complete set of optimized implementations

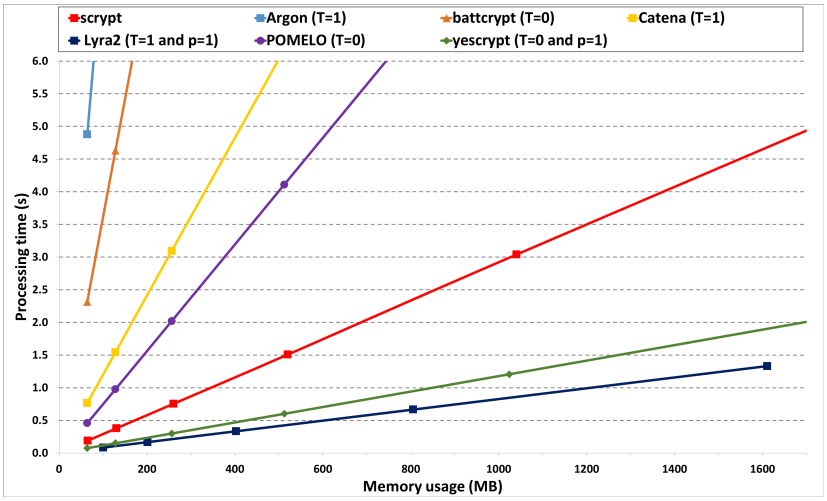


Fig. 15: Performance of SSE-enabled Lyra2, for $C = 256$, $\rho = 1$, $p = 1$, and different T and R settings, compared with SSE-enabled scrypt and memory-hard PHC finalists with minimum parameters.

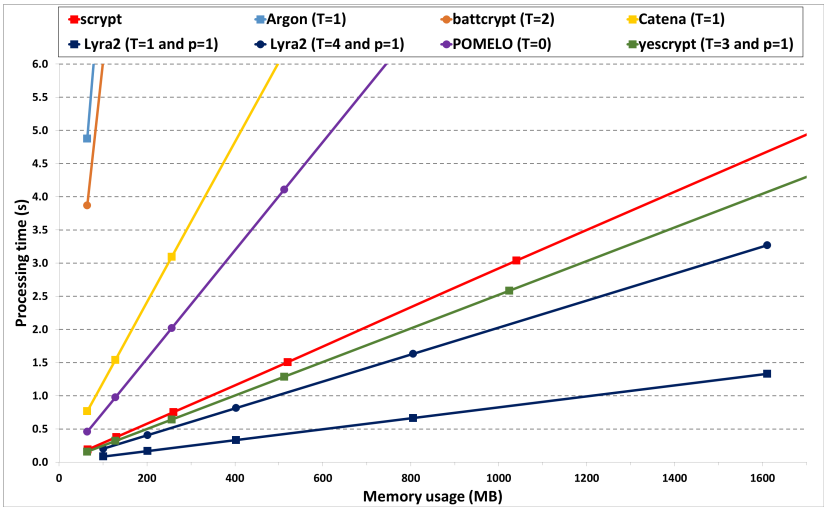


Fig. 16: Performance of SSE-enabled Lyra2, for $C = 256$, $\rho = 1$, $p = 1$ and different T and R settings, compared with SSE-enabled scrypt and memory-hard PHC finalists with a similar number of calls to the underlying function (comparable configurations are marked with the same symbol, ■ or •).

for gathering such data, a reasonable approach is to consider the minimum parameters suggested by the authors of each scheme: even though this analysis does not ensure that the attack costs are similar to all schemes, it at least shows what the designers recommend as the bare minimum cost for legitimate users. The results, which basically confirm existing analysis done in [70], are depicted in Figure 15, which shows that Lyra2 is a very competitive solution in terms of performance.

Another normalization can be made if we consider that, in a nutshell, a memory-hard PHS consists of an iterative program that initializes and revisits several memory positions. Therefore, one can assess each algorithm's performance when they are all parameterized to make the same number of calls to the underlying non-invertible (possible cryptographic) function. The goal with this normalization is to evaluate how efficiently the underlying primitive is employed by the scheme, giving an overall idea of its throughput. It also provides some insight on how much that primitive should be optimized to obtain similar processing times for a given memory usage, or even if it is worthy replacing that primitive by a faster algorithm (assuming that the scheme is flexible enough to allow users to do so).

The benchmark results are shown in Figure 16, in which lines marked with the same symbol (e.g., ■ or •) denote algorithms configured with a similar number of calls to the underlying function. The exact choice of parameters in this figure comes from Table 3, which shows how each memory-hard PHC finalist handles the time- and memory-cost parameters (respectively, T and M), based

Algorithm	Calls to underlying primitive	SIMD instructions
Argon	$(1 + 33/32 \cdot T) \cdot M$	Yes
battcrypt	$\{2^{\lfloor T/2 \rfloor} \cdot [(T \mod 2) + 2] + 1\} \cdot M$	No
Catena[2]	$(T + 1) \cdot M$	Yes
Lyra2	$(T + 1) \cdot M$	Yes
POMELO	$\left(3 + 2^{2T}\right) \cdot M$	No
yescrypt	$(T - 1) \cdot M$	Yes

Table 3: PHC finalists: calls to underlying primitive in terms of their time and memory parameters, T and M, and their implementations.

on the analysis of the documentation provided by their authors [14,50,71]. The source codes were all compiled with the `-O3` option whenever the authors did not specify the use of another compilation flag. Once again, Lyra2 displays a superior performance, which is a direct result of adopting an efficient and reduced-round cryptographic sponge as underlying primitive.

One remark concerning these results is that, as also shown in Table 3, the implementations of battcrypt and POMELO employed in the benchmarks do not employ SIMD instructions, which means that the comparison is not completely fair. Nevertheless, even if such advanced instructions are able to reduce their processing times by half, their relative positions on the figure would not change.

7.2 Benchmarks for Lyra2 with parallelism

To assess the performance of our scheme when executed with multiple processing cores in a legitimate platform, we conducted tests with the parallel version of Lyra2 described in Section 6.2, called Lyra2$_p$.

The results for $p = 2$ (i.e., two processing cores) are shown in Figure 17, which indicates a gain of roughly 46% when compared with the numbers discussed in Section 7.1. More precisely, Lyra2$_p$ is expected to execute in: approximately 1 s while using up to 800 MB (with $R = 2^{15}$, $T = 5$ and $p = 2$) or up to 1.1 GB of memory (with $R \approx 5.4 \cdot 10^4$, $T = 3$ and $p = 2$); or in less than 2.5 s with 1.6 GB (with $R = 2^{16}$, $T = 6$ and $p = 2$). With $p = 4$ (i.e., four processing cores), the gain becomes approximately 60% when compared with the implementation that does not take advantage of parallelism, as depicted in Figure 18.

Figures 17 and 18 also compare the performance of Lyra2$_p$ and yescrypt, the two fastest memory-hard PHC finalists, when both schemes are executed with the same number of processing cores. To allow the analysis of a broad

[2] The exact number of calls to the underlying cryptographic primitive in Catena is given by equation $(g - g_0 + 1) \cdot (T + 1) \cdot M$, where g and g_0 are, respectively, the current and minimum garlic. However, since normally $g = g_0$, here we use the simplified equation $(T + 1) \cdot M$.

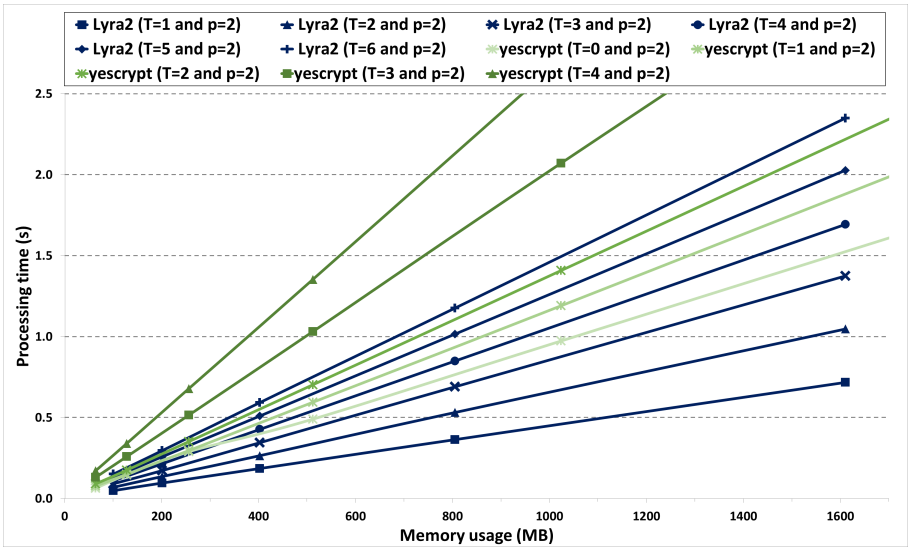


Fig. 17: Performance of SSE-enabled Lyra2, for $C = 256$, $\rho = 1$, $p = 2$, and different T and R settings, compared with SSE-enabled yescrypt. Configurations with a similar number of calls to the underlying function are marked with the same symbol, ■ or ▲.

spectrum of parameters, the notation on those figures is such that: (1) lines marked with a same symbol (■ or ▲) denote algorithms configured to execute the same number of calls to the underlying primitive; (2) lines marked with $*$ indicate that yescrypt has been parameterized to execute a lower number of calls to the underlying function than Lyra2 with $T = 1$; and (3) lines marked with other symbols denote the execution of Lyra2 with $T \geq 3$, for which the number of calls to the underlying function does not match any of the lines shown for yescrypt. As shown in these figures, Lyra2$_p$ remains quite competitive, and keeps surpassing the performance of yescrypt for both the "minimal" and the "similar number of calls to the underlying function" parameterizations.

It is also interesting to notice that the performance gain of Lyra2 when raising p from 2 to 4, although noticeable, is lower than the one obtained from raising p from 1 to 2. In fact, complementary tests with $p > 4$ were also performed, but neither Lyra2 or yescrypt have shown any substantial performance gain in our Intel Xeon E5-2430 employed as testbed. We believe that the main reason behind this barrier lies on the hardware's memory bandwidth limitations, of 32 GB/s [72], since a higher number of cores results in a higher occupation of the main memory bus for both algorithms.

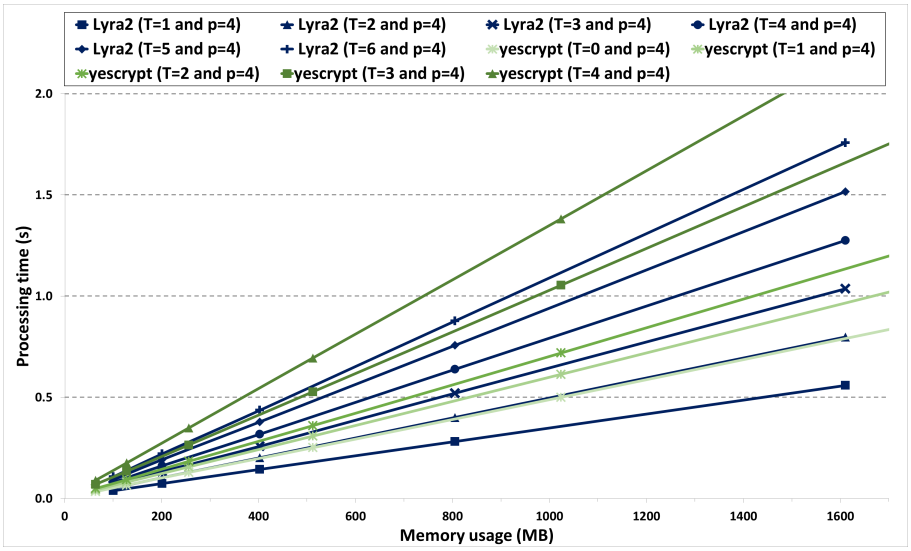


Fig. 18: Performance of SSE-enabled Lyra2, for $C = 256$, $\rho = 1$, $p = 4$, and different T and R settings, compared with SSE-enabled yescrypt. Configurations with a similar number of calls to the underlying function are marked with the same symbol, ■ or ▲.

7.3 Benchmark of GPU-based attacks

Aiming to evaluate the costs of attacks against Lyra2 using a GPU, we implemented the algorithm in CUDA for two different settings. In the first, we run a single instance of Lyra2 configured to use different amounts of memory (from 1.5 MB to 400 MB) , emulating a scenario in which the GPU has not enough memory to simultaneously accommodate multiple password guesses; in this case, used the device's shared memory to hold the sponge's state, and the number of threads run is that defined by the algorithm's parallelism parameter, p. In the second, we configure Lyra2 to run with a small amount of memory (namely, 2.25 MB), and then evaluate the throughput provided by the execution of several password guesses in parallel; in this scenario, aiming to maximize the GPU's occupancy, we kept the sponge's states in global memory without any use of the GPU's shared memory.

Regarding the implementations, the code obtained is basically a direct port of the CPU code, with some small adaptations for ensuring compatibility and good performance on the target platform, considering aspects such as the hardware characteristics and the virtual machine's instruction set. The GPU board used as testbed is an NVIDIA GeForce GTX TITAN (Kepler architecture, GK110) [73], which has 2688 CUDA cores (14 Multiprocessors with 192 CUDA Cores each) operating at 0.876 GHz, and a total amount of global memory of 6144 MB

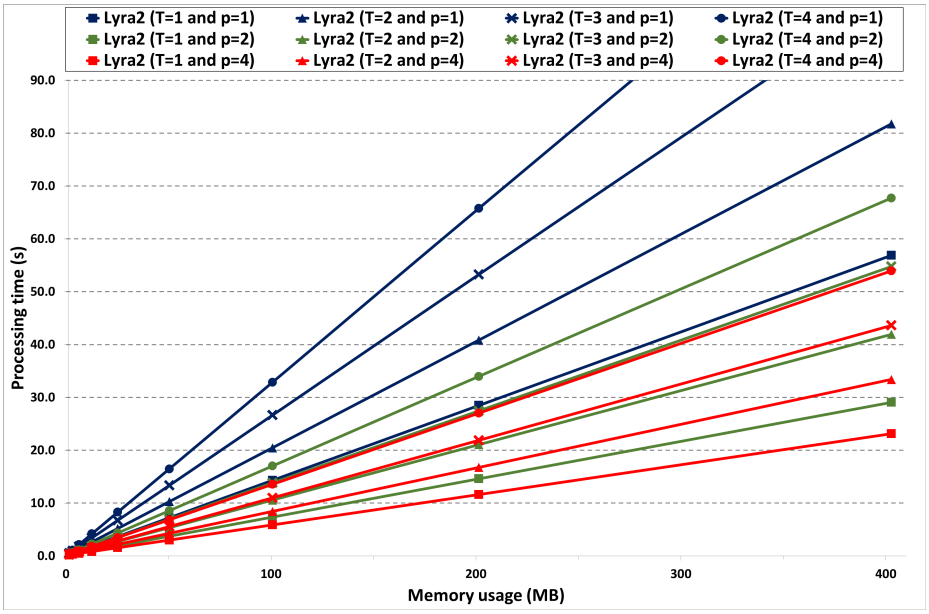


Fig. 19: Performance of GPU-oriented implementation of Lyra2, for a single instance configured with $C = 256$, $\rho = 1$, and different T, R and p settings, on NVIDIA GeForce GTX TITAN.

operating at 3 GHz. We used the CUDA 6.5 driver with 5.0 runtime version and configured the architecture to 3.5, the higher value allowed by the board.

The results obtained for the first scenario (i.e., the execution of a single instance), for an average of six executions of Lyra2 with $C = 256$ and different p, T and R settings are shown in Figure 19. As observed in this figure, the performance obtained in the GPU was very low: even for $T = 1$ and $p = 4$, which corresponds to the best performance on the GPU, the execution time is approximately 100 times higher that the one with the same settings on a CPU (see Figure 18). Such performance penalty is most likely due to the latency caused by the pseudorandom access pattern adopted in Lyra2, since GPUs are optimized for delivering high throughput rather than low latency.

The latency observed in the single-instance scenario can usually be masked by the GPU if it runs several threads in parallel. To measure this ability of GPUs of hiding latency and providing high throughput, an interesting metric is the GPU's occupancy. Namely, the occupancy is calculated as the total number of active warps (and, consequently, threads) per multiprocessor, which is a characteristic of the code being executed, divided by the maximum number of warps that could be active per multiprocessor, which depends on the GPU board's hardware. If the memory matrix is too large to allow many guesses to be performed in parallel, as emulated in the first scenario, the occupancy is very low. In contrast, the lower memory usage of the second test scenario, of only 2.25 MB, allows

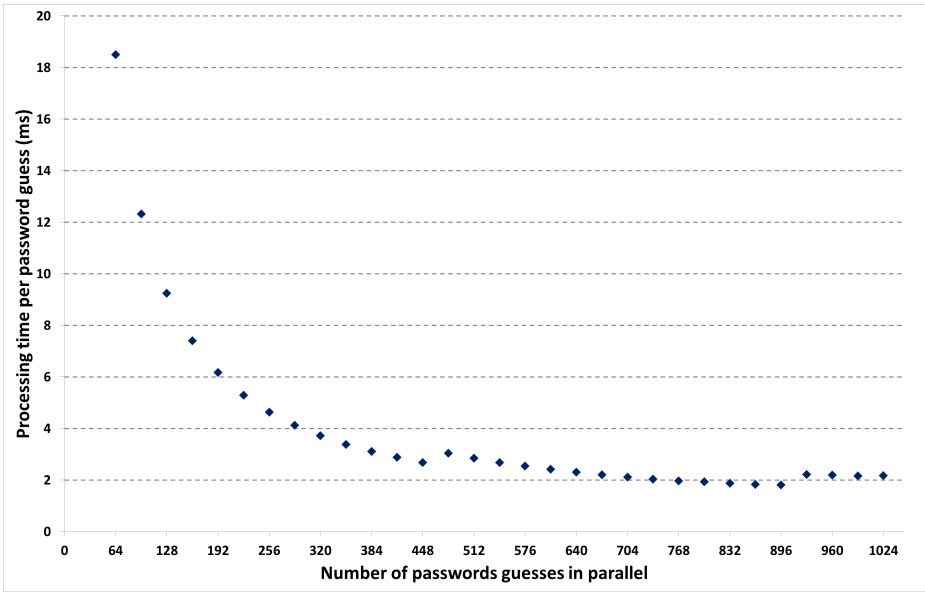


Fig. 20: Performance of a GPU-oriented attack against Lyra2, for $T = 1$, $C = 1024$, $R = 24$, $p = 4$, $\rho = 1$, and different number of passwords, on NVIDIA GeForce GTX TITAN.

a larger number of instances to be executed in parallel by the multiple GPU cores. Not surprisingly, as shown in Figure 20, the GPU's performance for Lyra2 configured with $T = 1$ and $p = 4$ as adopted in this second case is such that the average time taken per password test drops to 18 ms for 64 parallel instances (i.e., 256 threads), and to 1.8 ms when the GPU's memory is completely filled with 896 instances (for 3584 threads). One remark concerning these benchmarks is that, given the high number of instances running simultaneously, our tests have shown that it would not be advantageous to keep the sponges' states in the GPU's shared memory for this second scenario. The reason is that this approach would implicate in a lower number of threads being executed per block and, consequently, on a lower throughput due to the GPU's reduced capability of hiding latencies.

Nevertheless, even when the GPU's memory is completely committed to the 896 password hashing instances, the throughput provided in our tests is still 4.5 times lower than the 0.4 ms obtained with the same parameterization of Lyra2 on the CPU employed as testbed. Whilst this is much better than the 100 times slowdown obtained in the single-instance scenario of Figure 19, at least in principle this GPU-friendly scenario may still not advantageous enough to justify using a GPU as the preferred attack platform. After all, assuming similar purchasing prices for both platforms, the GPU would not only provide a lower throughput than the CPU employed, but is also likely to consume more

energy for this task. Nonetheless, we recommend that legitimate users adopt parameters resulting in a larger memory usage whenever the target application's requirements and constraints allow them to do so, thus hindering an attacker's ability to take full advantage of the parallelization and latency-hiding capabilities of commercial GPUs.

7.4 Benchmarks for Lyra2 with the BlaMka G function

Since BlaMka includes a larger number of operations than Blake2b, it is natural that the performance of Lyra2 when it employs BlaMka instead of Blake2b as underlying permutation will be lower than that reported in the previous subsections. Therefore, we conducted some benchmarks to assess the impacts of BlaMka over Lyra2's efficiency. Figure 21 shows the results for Lyra2 configured with $p = 1$, comparing it with the other memory-hard PHC finalists. As observed in this figure, Lyra2's performance remains quite competitive: for a given memory usage, Lyra2 is slower only than yescrypt configured with minimal settings, but remains faster than yescrypt when both are configured to make the same number of calls to the underlying function (i.e., for yescrypt with $T = 3$ and Lyra2 with $T = 1$).

When Lyra2 is configured to take advantage of parallelism, on the other hand, the impacts of BlaMka over the algorithm's performance are comparatively less noticeable. Indeed, as shown in Figure 22 for $p = 2$, as well as in Figure 23 for $p = 4$, with these configurations Lyra2 outperforms yescrypt both in

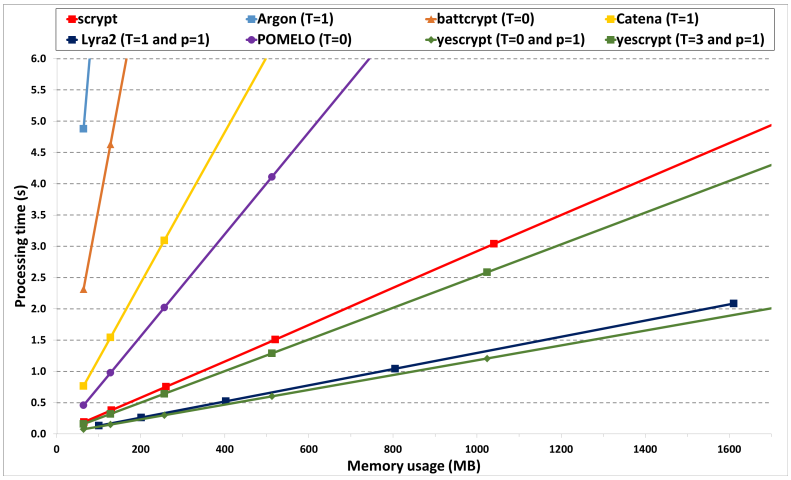


Fig. 21: Performance of SSE-enabled Lyra2 with BlaMka G function, for $C = 256$, $\rho = 1$, $p = 1$, and different T and R settings, compared with SSE-enabled scrypt and memory-hard PHC finalists (configurations with a similar number of calls to the underlying function are marked with the same symbol, ■).

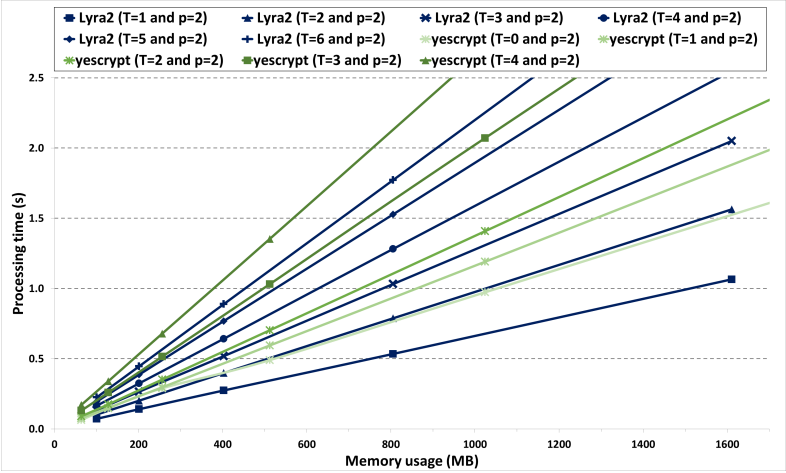


Fig. 22: Performance of SSE-enabled Lyra2 with BlaMka G function, for $C = 256$, $\rho = 1$, $p = 2$, and different T and R settings, compared with SSE-enabled yescrypt. Configurations with a similar number of calls to the underlying function are marked with the same symbol, ■ or ▲.

the "minimal" and in the "similar number of calls to the underlying function" parameterizations.

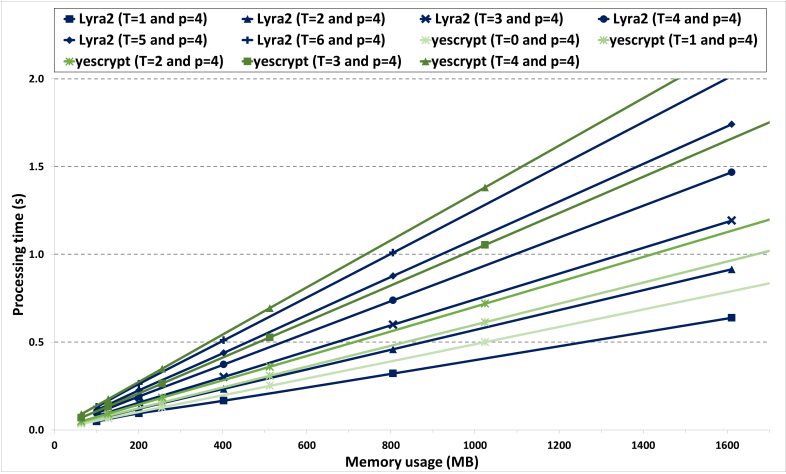


Fig. 23: Performance of SSE-enabled Lyra2 with BlaMka G function, for $C = 256$, $\rho = 1$, $p = 4$, and different T and R settings, compared with SSE-enabled yescrypt. Configurations with a similar number of calls to the underlying function are marked with the same symbol, ■ or ▲.

7.5 Expected attack costs

Considering that the cost of DDR3 SO-DIMM memory chips is currently around U$8.6/GB [74], Table 4 shows the cost added by Lyra2 with $T = 5$ when an attacker tries to crack a password in 1 year using the above reference hardware, for different password strengths — we refer the reader to [7, Appendix A] for a discussion on how to compute the approximate entropy of passwords. These costs are obtained considering the total number of instances that need to run in parallel to test the whole password space in 365 days and supposing that testing a password takes the same amount of time as in our testbed. Notice that, in a real scenario, attackers would also have to consider costs related to wiring and energy consumption of memory chips, besides the cost of the processing cores themselves.

We notice that if the attacker uses a faster platform (e.g., an FPGA or a more powerful computer), these costs should drop proportionally, since a smaller number of instances (and, thus, memory chips) would be required for this task. Similarly, if the attacker employs memory devices faster than regular DRAM (e.g., SRAM or registers), the processing time is also likely to drop, reducing the number of instances required to run in parallel. Nonetheless, in this case the resulting memory-related costs may actually be significantly bigger due to the higher cost per GB of such memory devices. Anyhow, the numbers provided in Table 4 are not intended as absolute values, but rather a reference on how much extra protection one could expect from using Lyra2, since this additional memory-related cost is the main advantage of any PHS that explores memory usage when compared with those that do not.

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

Indeed, from Equation 8 and for $T = 5$, the processing cost of an attack against Lyra2 using half of the memory defined by the legitimate user would be $\mathcal{O}((3/2)^{2T}R^2)$, which translates to $(3/2)^{2\cdot5} \cdot (2^{14})^2 \approx 2^{34}\sigma$ if the algorithm ope-

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	315.1	1.3k	5.0k	20.1k	917.8	3.7k	14.7k	59.1k
40	10.1k	40.2k	160.7k	642.9k	29.4k	117.7k	471.9k	1.9M
45	322.7k	1.3M	5.1M	20.6M	939.8k	3.8M	15.1M	60.5M
50	10.3M	41.2M	164.5M	658.3M	30.1M	120.6M	483.2M	1.9B
55	330.4M	1.3B	5.3B	21.1B	962.4M	3.9B	15.5B	62.0B

Table 4: 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 Xeon E5-2430 or equivalent processor.

rates regularly with 400 MB, or $(3/2)^{2.5} \cdot (2^{16})^2 \approx 2^{38}\sigma$ for a memory usage of 1.6 GB. For the same memory usage settings, the total cost of a *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 employed in our experiments. As expected, such elevated processing costs resulting from this small memory usage reduction are prone to discourage attack venues that try to avoid the memory costs of Lyra2 by means of extra processing.

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 extensions discussed in Section 6, Lyra2 can be further personalized for different scenarios, including parallel legitimate platforms (with the p parameter).

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, and in part by the Brazilian Coordination for the Improvement of Higher Education Personnel (CAPES) under grant 79414400249.

References

1. Chakrabarti, S., Singbal, M.: Password-based authentication: Preventing dictionary attacks. Computer **40**(6) (june 2007) 68–74
2. Conklin, A., Dietrich, G., Walz, D.: Password-based authentication: A system perspective. In: Proc. of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04). Volume 7 of HICSS'04., Washington, DC, USA, IEEE Computer Society (2004) 170–179

3. Bonneau, J., Herley, C., van Oorschot, P.C., Stajano, F.: The Quest to Replace Passwords: A Framework for Comparative Evaluation of Web Authentication Schemes. In: IEEE Symposium on Security and Privacy. (2012) 553–567
4. NIST: Special Publication 800-18 – Recommendation for Key Derivation Using Pseudorandom Functions. National Institute of Standards and Technology, U.S. Department of Commerce. (October 2009) `http://csrc.nist.gov/publications/nistpubs/800-108/sp800-108.pdf`.
5. Percival, C.: Stronger key derivation via sequential memory-hard functions. In: BSDCan 2009 – The Technical BSD Conference. (2009)
6. Kaliski, B.: PKCS#5: Password-Based Cryptography Specification version 2.0 (RFC 2898). (2000)
7. NIST: Special Publication 800-63-1 – Electronic Authentication Guideline. National Institute of Standards and Technology, U.S. Department of Commerce. (December 2011) `http://csrc.nist.gov/publications/nistpubs/800-63-1/SP-800-63-1.pdf`.
8. Florencio, D., Herley, C.: A Large Scale Study of Web Password Habits. In: Proc. of the 16th International Conference on World Wide Web, Alberta, Canada (2007) 657–666
9. Herley, C., van Oorschot, P., Patrick, A.: Passwords: If We're So Smart, Why Are We Still Using Them? In: Financial Cryptography and Data Security. Volume 5628 of LNCS., Springer Berlin / Heidelberg (2009) 230–237
10. Dürmuth, M., Güneysu, T., Kasper, M.: Evaluation of Standardized Password-Based Key Derivation against Parallel Processing Platforms. In: Computer Security – ESORICS 2012. Volume 7459 of LNCS. Springer Berlin Heidelberg (2012) 716–733
11. Sprengers, M.: GPU-based Password Cracking: On the Security of Password Hashing Schemes regarding Advances in Graphics Processing Units. Master's thesis, Radboud University Nijmegen (2011)
12. Marechal, M.: Advances in password cracking. Journal in Computer Virology **4**(1) (2008) 73–81
13. Provos, N., Mazières, D.: A future-adaptable password scheme. In: Proc. of the FREENIX track: 1999 USENIX annual technical conference. (1999)
14. PHC: Password Hashing Competition. `https://password-hashing.net/` (2013)
15. Almeida, L., Andrade, E., Barreto, P., Simplicio, M.: Lyra: Password-Based Key Derivation with Tunable Memory and Processing Costs. Journal of Cryptographic Engineering **4**(2) (2014) 75–89 See also `eprint.iacr.org/2014/030`.
16. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: Sponge functions. (ECRYPT Hash Function Workshop 2007) (2007) Also available at `http://csrc.nist.gov/pki/HashWorkshop/Public_Comments/2007_May.html`.
17. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: Cryptographic sponge functions - version 0.1. http://keccak.noekeon.org/ (2011)
18. Andreeva, E., Mennink, B., Preneel, B.: The Parazoa family: Generalizing the Sponge hash functions. IACR Cryptology ePrint Archive **2011** (2011) 28
19. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: The Keccak SHA-3 submission. Submission to NIST (Round 3) (2011)
20. Kelsey, J., Schneier, B., Hall, C., Wagner, D.: Secure Applications of Low-Entropy Keys. In: Proc. of the 1st International Workshop on Information Security. ISW '97, London, UK, UK, Springer-Verlag (1998) 121–134
21. Weir, M., Aggarwal, S., Medeiros, B.d., Glodek, B.: Password Cracking Using Probabilistic Context-Free Grammars. In: Proc. of the 30th IEEE Symposium

on Security and Privacy. SP'09, Washington, DC, USA, IEEE Computer Society (2009) 391–405
22. Nvidia: CUDA C programming guide (v6.5). http://docs.nvidia.com/cuda/cuda-c-programming-guide/ (August 2014)
23. Khronos Group: The OpenCL Specification – Version 1.2. (2012)
24. Nvidia: Tesla Kepler family product overview. http://www.nvidia.com/content/tesla/pdf/Tesla-KSeries-Overview-LR.pdf (2012)
25. Dandass, Y.S.: Using FPGAs to Parallelize Dictionary Attacks for Password Cracking. In: Proc. of the 41st Annual Hawaii International Conference on System Sciences (HICSS 2008), IEEE (2008) 485–485
26. Kakarountas, A.P., Michail, H., Milidonis, A., Goutis, C.E., Theodoridis, G.: High-Speed FPGA Implementation of Secure Hash Algorithm for IPSec and VPN Applications. The Journal of Supercomputing **37**(2) (2006) 179–195
27. Chung, E.S., Milder, P.A., Hoe, J.C., Mai, K.: Single-Chip Heterogeneous Computing: Does the Future Include Custom Logic, FPGAs, and GPGPUs? In: Proc. of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO'43, Washington, DC, USA, IEEE Computer Society (2010) 225–236
28. Fowers, J., Brown, G., Cooke, P., Stitt, G.: A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications. In: Proc. of the ACM/SIGDA Internbational Symposium on Field Programmable Gate Arrays (FPGA'12), New York, NY, USA, ACM (2012) 47–56
29. SciEngines: Rivyera s3-5000. http://sciengines.com/products/computers-and-clusters/rivyera-s3-5000.html
30. SciEngines: Rivyera v7-2000t. http://sciengines.com/products/computers-and-clusters/v72000t.html
31. NIST: Federal Information Processing Standard (FIPS PUB 198) – The Keyed-Hash Message Authentication Code. National Institute of Standards and Technology, U.S. Department of Commerce. (March 2002) http://csrc.nist.gov/publications/fips/fips198/fips-198a.pdf.
32. Bernstein, D.: The Salsa20 family of stream ciphers. In Robshaw, M., Billet, O., eds.: New Stream Cipher Designs. Springer-Verlag, Berlin, Heidelberg (2008) 84–97
33. Aumasson, J.P., Fischer, S., Khazaei, S., Meier, W., Rechberger, C.: New features of latin dances: Analysis of Salsa, ChaCha, and Rumba. In: Fast Software Encryption. Volume 5084., Berlin, Heidelberg, Springer-Verlag (2008) 470–488
34. Daemen, J., Rijmen, V.: A new MAC construction ALRED and a specific instance ALPHA-mac. In: Fast Software Encryption – FSE'05. (2005) 1–17
35. Daemen, J., Rijmen, V.: Refinements of the ALRED construction and MAC security claims. Information Security, IET **4**(3) (2010) 149–157
36. Simplicio, M.A., Barbuda, P., Barreto, P., Carvalho, T., Margi, C.: The MARVIN Message Authentication Code and the LETTERSOUP Authenticated Encryption Scheme. Security and Communication Networks **2** (2009) 165–180
37. Simplicio, M.A., Barreto, P.: Revisiting the Security of the ALRED Design and Two of Its Variants: Marvin and LetterSoup. IEEE Transactions on Information Theory **58**(9) (2012) 6223–6238
38. Gaj, K., Homsirikamol, E., Rogawski, M., Shahid, R., Sharif, M.U.: Comprehensive Evaluation of High-Speed and Medium-Speed Implementations of Five SHA-3 Finalists Using Xilinx and Altera FPGAs. Cryptology ePrint Archive, Report 2012/368 (2012) http://eprint.iacr.org/2012/368.
39. Aumasson, J.P., Neves, S., Wilcox-O'Hearn, Z., Winnerlein, C.: BLAKE2: simpler, smaller, fast as MD5. https://blake2.net/ (2013)

40. Aumasson, J.P., Henzen, L., Meier, W., Phan, R.: SHA-3 proposal BLAKE (version 1.3). https://131002.net/blake/blake.pdf (2010)
41. Chang, S., Perlner, R., Burr, W.E., Turan, M.S., Kelsey, J.M., Paul, S., Bassham, L.E.: Third-Round Report of the SHA-3 Cryptographic Hash Algorithm Competition. US Department of Commerce, National Institute of Standards and Technology (2012)
42. Aumasson, J.P., Guo, J., Knellwolf, S., Matusiewicz, K., Meier, W.: Differential and Invertibility Properties of BLAKE. In Hong, S., Iwata, T., eds.: Fast Software Encryption. Volume 6147 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2010) 318–332 See also http://eprint.iacr.org/2010/043.
43. Ming, M., Qiang, H., Zeng, S.: Security analysis of BLAKE-32 based on differential properties. In: 2010 International Conference on Computational and Information Sciences (ICCIS), IEEE (2010) 783–786
44. Ji, L., Liangyu, X.: Attacks on round-reduced BLAKE. Technical report, Cryptology ePrint Archive, Report 2009/238 (2009) http://eprint.iacr.org/2009/238.
45. Su, B., Wu, W., Wu, S., Dong, L.: Near-Collisions on the Reduced-Round Compression Functions of Skein and BLAKE. In: Cryptology and Network Security. Volume 6467 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2010) 124–139
46. Guo, J., Karpman, P., Nikoli, I., Wang, L., Wu, S.: Analysis of BLAKE2. In: Topics in Cryptology (CT-RSA 2014). Volume 8366 of LNCS., Springer International Publishing (2014) 402–423 see also https://eprint.iacr.org/2013/467.
47. Shand, M., Bertin, P., Vuillemin, J.: Hardware Speedups in Long Integer Multiplication. In: Proceedings of the Second Annual ACM Symposium on Parallel Algorithms and Architectures. SPAA'90, New York, NY, USA, ACM (1990) 138–145
48. Soderquist, P., Leeser, M.: An area/performance comparison of subtractive and multiplicative divide/square root implementations. In: Computer Arithmetic, 1995., Proceedings of the 12th Symposium on. (Jul 1995) 132–139
49. Cox, B.: TwoCats (and SkinnyCat): A Compute Time and Sequential Memory Hard Password Hashing Scheme. Password Hashing Competition. v0 edn. (March 2014) https://password-hashing.net/submissions/specs/TwoCats-v0.pdf.
50. Peslyak, A.: yescrypt - a Password Hashing Competition submission. Password Hashing Competition. v0 edn. (March 2014) https://password-hashing.net/submissions/specs/yescrypt-v0.pdf.
51. Neves, S.: Re: A review per day - Lyra2 – Public archives of PHC list. http://article.gmane.org/gmane.comp.security.phc/2045 (2014)
52. Wallis, W.D., George, J.: Introduction to Combinatorics. Discrete Mathematics and Its Applications. Taylor & Francis (2011)
53. Aumasson, J.P., Jovanovic, P., Neves, S.: Analysis of NORX. In: Proc. of the 3rd Int. Conf. on Cryptology and Information Security in Latin America (Latincrypt). (2014) 55–72 See also https://eprint.iacr.org/2014/317.
54. Aumasson, J.P., Jovanovic, P., Neves, S.: NORX: Parallel and scalable AEAD. In: Computer Security - ESORICS 2014. Volume 8713 of LNCS. (2014) 19–36 See also https://norx.io/.
55. Halderman, J., Schoen, S., Heninger, N., Clarkson, W., Paul, W., Calandrino, J., Feldman, A., Appelbaum, J., Felten, E.: Lest we remember: cold-boot attacks on encryption keys. Commun. ACM **52**(5) (May 2009) 91–98
56. Yuill, J., Denning, D., Feer, F.: Using deception to hide things from hackers: Processes, principles, and techniques. Journal of Information Warfare **5**(3) (2006) 26–40

57. Forler, C., Lucks, S., Wenzel, J.: Catena: A Memory-Consuming Password Scrambler. Cryptology ePrint Archive, Report 2013/525 (2013) http://eprint.iacr.org/2013/525.
58. Cook, S.A.: An Observation on Time-storage Trade off. In: Proc. of the 5th Annual ACM Symposium on Theory of Computing (STOC'73), New York, NY, USA, ACM (1973) 29–33
59. Hellman, M.E.: A cryptanalytic time-memory trade-off. IEEE Transactions on Information Theory **26**(4) (1980) 401–406
60. Dwork, C., Naor, M., Wee, H.: Pebbling and Proofs of Work. In: Advances in Cryptology – CRYPTO 2005. Volume 3621 of Lecture Notes in Computer Science., Springer Berlin Heidelberg (2005) 37–54
61. Dziembowski, S., Kazana, T., Wichs, D.: Key-Evolution Schemes Resilient to Space-Bounded Leakage. In: Advances in Cryptology – CRYPTO 2011. Volume 6841 of Lecture Notes in Computer Science., Springer Berlin Heidelberg (2011) 335–353
62. Khovratovich, D., Biryukov, A., Groschdl, J.: Tradeoff cryptanalysis of password hashing schemes. PasswordsCon'14 (2014) See also https://www.cryptolux.org/images/4/4f/PHC-overview.pdf.
63. Bernstein, D.J.: Cache-timing attacks on AES. Technical report, University of Illinois (2005) http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.
64. 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.
65. Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM **21**(2) (Feb 1978) 120–126
66. Percival, C.: Cache missing for fun and profit. In: Proc. of BSDCan 2005. (2005)
67. Ristenpart, T., Tromer, E., Shacham, H., Savage, S.: Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds. In: Proc.s of the 16th ACM Conference on Computer and Communications Security. CCS '09, New York, NY, USA, ACM (2009) 199–212
68. Mowery, K., Keelveedhi, S., Shacham, H.: Are AES x86 Cache Timing Attacks Still Feasible? In: Proc.s of the 2012 ACM Workshop on Cloud Computing Security Workshop (CCSW'12), New York, NY, USA, ACM (2012) 19–24
69. Solar Designer: New developments in password hashing: ROM-port-hard functions. [Online] http://www.openwall.com/presentations/ZeroNights2012-New-In-Password-Hashing/ZeroNights2012-New-In-Password-Hashing.pdf (2012)
70. Broz, M.: Another PHC candidates "mechanical" tests – Public archives of PHC list. http://article.gmane.org/gmane.comp.security.phc/2237 (2014)
71. PHC wiki: Password Hashing Competition – wiki. https://password-hashing.net/wiki/ (2014)
72. Intel: Intel Xeon Processor E5-2430 (15M Cache, 2.20 GHz, 7.20 GT/s Intel QPI). http://ark.intel.com/products/64616/Intel-Xeon-Processor-E5-2430-15M-Cache-2_20-GHz-7_20-GTs-Intel-QPI (2012)
73. GeForce: GeForce GTX 470: Specifications. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-470/specifications (visited on Mar.29, 2014) (2014)
74. TrendForce: DRAM contract price (jan.13 2015). http://www.trendforce.com/price (visited on Jan.13, 2015) (2015)
75. TrueCrypt: TrueCrypt: Free open-source on-the-fly encryption – documentation. http://www.truecrypt.org/docs/ (2012)

76. Apple: iOS security. Technical report, Apple Inc. (2012) http://images.apple.com/ipad/business/docs/iOS_Security_May12.pdf.
77. Yao, F.F., Yin, Y.L.: Design and Analysis of Password-Based Key Derivation Functions. IEEE Transactions on Information Theory **51**(9) (2005) 3292–3297
78. Bellare, M., Ristenpart, T., Tessaro, S.: Multi-instance security and its application to password-based cryptography. In: Advances in Cryptology (CRYPTO 2012). Volume 7417 of LNCS., Springer Berlin Heidelberg (2012) 312–329
79. Schneier, B.: Description of a new variable-length key, 64-bit block cipher (Blowfish). In: Fast Software Encryption, Cambridge Security Workshop, London, UK, Springer-Verlag (1994) 191–204
80. Crew, B.: New carnivorous harp sponge discovered in deep sea. Nature (2012) Available online: http://www.nature.com/news/new-carnivorous-harp-sponge-discovered-in-deep-sea-1.11789.
81. Capcom: Blanka – Capcom Database. http://capcom.wikia.com/wiki/Blanka (2015)

Appendix A. PBKDF2

The Password-Based Key Derivation Function version 2 (PBKDF2) algorithm [6] was originally proposed in 2000 as part of RSA Laboratories' PKCS#5. It is nowadays present in several security tools, such as TrueCrypt [75] and Apple's iOS for encrypting user passwords [76], and has been formally analyzed in several circumstances [77,78].

Basically, PBKDF2 (see Algorithm 4) 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.

Algorithm 4 PBKDF2.

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

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

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 [13]. The scheme is based on a customized version of the 64-bit cipher algorithm Blowfish [79], 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

Algorithm 5 Bcrypt.

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

$s \leftarrow InitState()$ ▷ Copies the digits of π into the sub-keys and S-boxes S_i
$s \leftarrow$ExpandKey$(s, salt, pwd)$
for $i \leftarrow 1$ **to** 2^T **do**
  $s \leftarrow$ExpandKey$(s, 0, salt)$
  $s \leftarrow$ExpandKey$(s, 0, pwd)$
end for
$ctext \leftarrow$ "$OrpheanBeholderScryDoubt$"
for $i \leftarrow 1$ **to** 64 **do**
  $ctext \leftarrow BlowfishEncrypt(s, ctext)$
end for
return $T \parallel salt \parallel ctext$
function EXPANDKEY$(s, salt, pwd)$
  for $i \leftarrow 1$ **to** 32 **do**
    $P_i \leftarrow P_i \oplus pwd[32(i-1) \ldots 32i-1]$
  end for
  for $i \leftarrow 1$ **to** 9 **do**
    $temp \leftarrow BlowfishEncrypt(s, salt[64(i-1) \ldots 64i-1])$
    $P_{0+2(i-1)} \leftarrow temp[0 \ldots 31]$
    $P_{1+2(i-1)} \leftarrow temp[32 \ldots 64]$
  end for
  for $i \leftarrow 1$ **to** 4 **do**
    for $j \leftarrow 1$ **to** 128 **do**
      $temp \leftarrow BlowfishEncrypt(s, salt[64(j-1) \ldots 64j-1])$
      $S_i[2(j-1)] \leftarrow temp[0 \ldots 31]$
      $S_i[1+2(j-1)] \leftarrow temp[32 \ldots 63]$
    end for
  end for
  return s
end function

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

Like PBKDF2, bcrypt allows users to parameterize only its total running time. In addition to this shortcoming, some of its characteristics can be 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 [15] are detailed in Algorithm 6.

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.

Algorithm 6 The Lyra Algorithm.

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

▷ **Setup**: Initializes a $(R \times C)$ memory matrix
$Hash.absorb(\texttt{pad}(salt \parallel pwd))$ ▷ Padding rule: 10^*1
$M[0] \leftarrow Hash.squeeze_\rho(C \cdot b)$
for $row \leftarrow 1$ **to** $R - 1$ **do**
  for $col \leftarrow 0$ **to** $C - 1$ **do**
    $M[row][col] \leftarrow Hash.duplexing_\rho(M[row - 1][col], b)$
  end for
end for

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

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

Appendix D. Naming conventions

The name "Lyra" comes from *Chondrocladia lyra*, a recently discovered type of sponge [80]. While most sponges are harmless, this harp-like sponge is carnivorous, using its branches to ensnare its prey, which is then enveloped in a membrane and completely digested. The "two" suffix is a reference to its predecessor, Lyra [15], which displays many of Lyra2's properties hereby presented but has a lower resistance to attacks involving time-memory trade-offs. Lyra2's memory matrix displays some similarity with this species' external aspect, and we expect it to be at least as much aggressive against adversaries trying to attack it. ☺

Regarding the multiplication-hard sponge, its name came from an attempt to combined the name "Blake", which is the basis for the algorithm, with the letter "M", for indicating multiplications. A natural (?) answer for this combination was BlaMka, a misspelling of Blanka, the only avatar from the Street Fighter

original game series [81] that comes from Brazil and, as such, is a compatriot of this document's authors. ☺