Lyra: Password-Based Key Derivation with Tunable Memory and Processing Costs[*]

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

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

Abstract. We present Lyra, a password-based key derivation scheme based on cryptographic sponges. Lyra was designed to be strictly sequential (i.e., not easily parallelizable), providing strong security even against attackers that use 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. We compare Lyra with similar-purpose state-of-the-art solutions, showing how our proposal provides a higher security level and overcomes limitations of existing schemes. Specifically, we show that if we fix Lyra's total processing time t in a legitimate platform, the cost of a memory-free attack against the algorithm is exponential, while the best-known result in the literature (namely, against the scrypt algorithm) is quadratic. In addition, for an identical same processing time, Lyra allows for a higher memory usage than its counterparts, further increasing the cost of brute force attacks.

Keywords: Password-based key derivation, memory usage, cryptographic sponges

Note 1. *In this version, the Setup phase was updated, since the originally published was outdated, an unfortunate mistake that was noticed only after publication. We also used the opportunity to revise our security analysis, adding a few considerations on the storage of a few rows rather than only sponge states.*

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

[*] This paper can be seen as an updated of the paper "Lyra: Password-Based Key Derivation with Tunable Memory and Processing Costs" published at JCEN – Journal of Cryptographic Engineering, ISSN 2190-8508, Springer, pages 1-15, 2014.

("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 [12, 15]. For better or for worse, and despite the existence of many proposals for their replacement [11], 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 kind of key derivation functions (KDFs), cryptographic algorithms that allow the generation of a pseudorandom string of bits from the password itself [32]. Typically, the output of a key derivation is employed in one of two manners [36]: 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 key derivation's output is computationally infeasible [26, 36].

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 [33]. Indeed, a study from 2007 with 544,960 passwords from real users has shown an average entropy of approximately 40.5 bits [21], 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 [24, 12], allowing attackers to completely bypass the non-invertibility property of the key derivation process. For example, an attacker could apply the key derivation function 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. Indeed, such attacks commonly benefit from platforms equipped with many processing cores, such as modern GPUs [44, 20] or custom hardware [20, 29].

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 [12]. For this reason, modern key derivation solutions usually employ mechanisms for increasing the *cost* of brute force attacks. Schemes such as PBKDF2 [26] and bcrypt [38], for example, include a configurable parameter that controls the number of iterations performed, allowing the user to adjust the time required by the key derivation process. A more recent proposal, scrypt [36], 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 KDF 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 [37].

Aiming to address this need for stronger alternatives, in this paper we propose Lyra, a new mode of operation of cryptographic sponges [8, 9] for password-based key derivation. Lyra combines security, flexibility and some of the most appealing features of the above-mentioned key derivation solutions, including the ability to configure the desired amount of memory, processing time and parallelism to be used by the algorithm. In addition, Lyra provides higher security than its counterparts. For example, the processing cost of memory-free attacks against the algorithm grows exponentially with its time-controlling parameter, surpassing scrypt's quadratic growth in the same conditions. Hence, with a suitable choice of parameters, the attack approach of using extra processing for circumventing (part of) the algorithm's memory needs becomes quickly impractical. In addition, for an identical processing time, Lyra allows for a higher memory usage than its counterparts, further raising the costs of any possible attack venue.

The rest of this paper is organized as follows. Section 2 outlines the concept of cryptographic sponges. Section 3 describes the main requirements of KDFs and discusses the related work. Section 4 presents the Lyra algorithm and its design rationale, while Section 5 analyses its security. Section 6 shows our preliminary benchmark results. Finally, Section 7 presents our final remarks and ideas for future work.

2 Cryptographic Sponges

The concept of *cryptographic sponges* was formally introduced by Bertoni *et al.* in [8] and is described in detail in [9]. The elegant design of sponges has also motivated the creation of more general structures, such as the Parazoa family of functions [1]. Indeed, their flexibility is probably among the reasons that led Keccak [10], one of the members of the sponge family, to be elected as the new Secure Hash Algorithm (SHA-3).

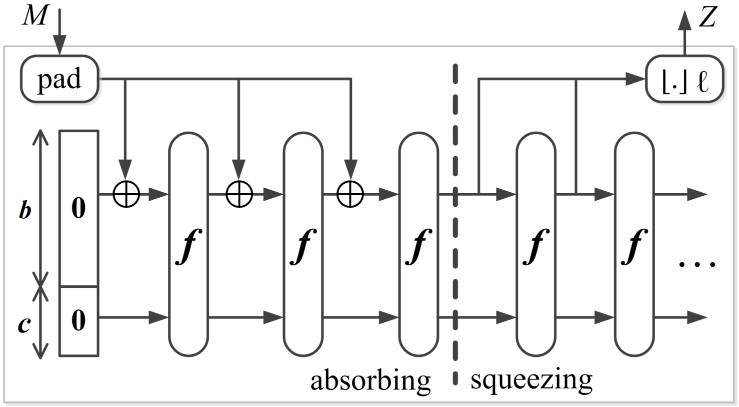

Fig. 1. Overview of the sponge construction $Z = [f, \text{pad}, b](M, \ell)$. Adapted from [9].

In what follows and throughout this document, we use the following notation: $\oplus$ denotes the XOR operation, $\|$ represents concatenation, and $|x|$ denotes the number of bits required for representing x.

2.1 Cryptographic Sponges: Basic Structure

In a nutshell, sponge functions provide an interesting way of building hash functions with arbitrary input and output lengths. Such functions are based on the so-called sponge construction, an iterated mode of operation that uses a fixed-length permutation (or transformation) f and a padding rule pad. More specifically, and as depicted in Figure 1, sponge functions rely on an internal state of $w = b + c$ 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 [10]). The parameters w, b and c are called, respectively, the *width*, *bitrate*, and the *capacity* of the sponge.

2.2 The duplex construction

A similar structure derived from the sponge concept is the *Duplex construction* [9], 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

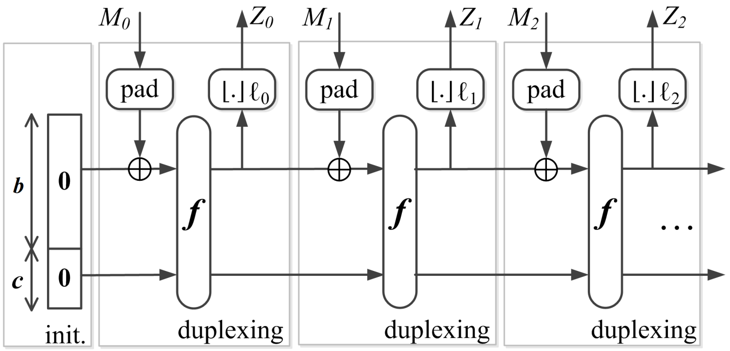

Fig. 2. Overview of the duplex construction. Adapted from [9].

case, the input string M must be short enough to fit in a single b-bit block after padding, and the output length ℓ must satisfy $\ell \leqslant b$.

3 Key Derivation Functions (KDFs)

As previously discussed, the basic requirement for a password-based KDF is to be non-invertible, so that recovering the password from its output is computationally infeasible. Moreover, a good KDF'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 [27]. 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 [47, 33]).

What modern KDFs 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* [27, 26]. 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 [26].

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 parameterized 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 KDF can be for themselves and for attackers. For example, a human user running a single KDF instance is unlikely to consider a nuisance that the derivation 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 key derivation 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 KDF 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 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 KDFs, in what follows we discuss the main characteristics of platforms used by attackers and then how existing solutions counter those threats.

3.1 Attack platforms

The most dangerous threats faced by KDFs 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 [20].

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 [34] and OpenCL [28]). As a result, they became more intensively employed for more general purposes, including password cracking [44, 20].

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 KDF'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 [35]. Its computational power can also be further expanded by using the host machine's resources [34], although this is also likely to limit the memory throughput. Supposing this GPU is used to attack a KDF whose parameterization 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 KDF requires 20 MB of DRAM, the maximum number of cores that could be used simultaneously becomes 300, only 11% of the total available.

Field Programmable Gate Arrays (FPGAs). An FPGA is a collection of configurable logic blocks wired together and with memory elements, forming a programmable and high-performance integrated circuit. In addition, as such devices are configured to perform a specific task, they can be highly optimized for its purpose (e.g., using pipelining [19, 25]). 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 [29]. When compared to GPUs, FPGAs may also be advantageous due to the latter's considerably lower energy consumption [14, 22], which can be further reduced if its circuit is synthesized in the form of custom logic hardware (ASIC) [14].

A recent example of password cracking using FPGAs is presented in [20]. Using a RIVYERA S3-5000 cluster [40] 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) [20, Sec. 4.2]. Against a KDF 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 [40] and, thus, up to three passwords can be processed in parallel rather than 5,376.

Interestingly, a KDF that requires a similar memory usage would be troublesome even for state-of-the-art clusters, such as the newer RIVYERA V7-2000T [41]. 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 [41]. Despite being much more powerful, in principle it would still be unable to test more than 2,600 passwords in parallel against a KDF that strictly requires 20 MB to run.

3.2 Scrypt

Arguably, the main password-based key derivation solutions available in the literature are [37]: PBKDF2 [26], bcrypt [38] and scrypt [36]. Since scrypt is only KDF among them that explores both memory and processing costs and, thus, is directly comparable to Lyra, its main characteristics are described in what follows. For the interested reader, a discussion on PBKDF2 and bcrypt is provided in Appendices A and B.

The design of scrypt [36] 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 [36] 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

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
$(B_0 ... B_{p-1}) \leftarrow$PBKDF2$_{HMAC-SHA-256}(pwd, salt, 1, p \cdot b)$
for $i \leftarrow 0$ to $p - 1$ do
    $B_i \leftarrow$ROMIX$(B_i, R)$
end for
$K \leftarrow$PBKDF2$_{HMAC-SHA-256}(pwd, B_0||B_1||...||B_{p-1}, 1, k)$
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 $V$
        $V_i \leftarrow X$ ; $X \leftarrow$BLOCKMIX$(X)$
    end for
    for $i \leftarrow 0$ to $R - 1$ do   ▷ Reads random positions of $V$
        $j \leftarrow Integerify(X) \bmod R$
        $X \leftarrow$BLOCKMIX$(X \oplus V_j)$
    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
```

memory-hard *ROMix* function. Basically, *ROMix* initializes an array V of R b-long elements by iteratively hashing B_i. It then visits R positions of V at random, updating the internal state variable X during this (strictly sequential) process in order to ascertain that those positions are indeed available in memory. The hash function employed by *ROMix* is called *BlockMix*, which emulates a function having arbitrary (b-long) input and output lengths; this is done using the Salsa20/8 [7] 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 [3] are not expected to put the security of scrypt in hazard [36], using a stronger alternative would be at least advisable, especially considering that the scheme's structure does not impose serious restrictions on the internal hash algorithm used by *BlockMix*. In this case, a sponge function could itself be an alternative. However, sponges' intrinsic properties make some of scrypt's operations unnecessary: since sponges support inputs and outputs of any length, the whole *BlockMix* structure could be replaced; in addition, sponges can operate in the stateful and sequential duplexing mode, meaning that the state variable X used by *ROMix* would become redundant.

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

4 Lyra

As any KDF, Lyra takes as input a salt and a password, creating a pseudorandom output that can be then be used as key material for cryptographic algorithms [32]. Internally, the scheme's memory is organized as a matrix that is required during the whole key derivation process. This matrix is iteratively accessed as many times as defined by the user, allowing Lyra's execution time to be fine-tuned according to the target platform's resources. The construction and visitation of the matrix is done using a stateful combination of the absorbing, squeezing and duplexing operations of the underlying sponge (i.e., its internal state is never reset to zeros), ensuring the sequential nature of the whole process.

In this section, we first describe the Lyra algorithm in detail, and then discuss its design rationale and possible variants.

4.1 Structure

Lyra's steps are detailed in Algorithm 2.

The first part of the algorithm is the *Setup Phase* (lines 1 – 8). 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

Algorithm 2 The Lyra Algorithm.

```
PARAM: $Hash$    ▷ Sponge with block size $b$ (in bits) and underlying permutation $f$
PARAM: $\rho$    ▷ Number of rounds of $f$ in the Setup and Wandering phases
INPUT: $pwd$    ▷ The password
INPUT: $salt$    ▷ A random salt
INPUT: $T$    ▷ Time cost, in number of iterations
INPUT: $R$    ▷ Number of rows in the memory matrix
INPUT: $C$    ▷ Number of columns in the memory matrix
INPUT: $k$    ▷ The desired key length, in bits
OUTPUT: $K$    ▷ The password-derived $k$-long key
 1: ▷ Setup: Initializes a $(R \times C)$ memory matrix whose cells have $b$ bits each
 2: $Hash.absorb(\mathrm{pad}(salt \parallel pwd))$    ▷ Padding rule: 10*1
 3: $M[0] \leftarrow Hash.squeeze_{\rho}(C \cdot b)$
 4: for $row \leftarrow 1$ to $R-1$ do
 5:     for $col \leftarrow 0$ to $C-1$ do
 6:         $M[row][col] \leftarrow Hash.duplexing_{\rho}(M[row-1][col], b)$
 7:     end for
 8: end for
 9: ▷ Wandering: Iteratively overwrites blocks of the memory matrix
10: $row \leftarrow 0$
11: for $i \leftarrow 0$ to $T-1$ do    ▷ Time Loop
12:     for $j \leftarrow 0$ to $R-1$ do    ▷ Rows Loop: randomly visits $R$ rows
13:         for $col \leftarrow 0$ to $C-1$ do    ▷ Columns Loop: visits blocks in row
14:             $M[row][col] \leftarrow M[row][col] \oplus Hash.duplexing_{\rho}(M[row][col], b)$
15:         end for
16:         $col \leftarrow M[row][C-1] \bmod C$
17:         $row \leftarrow Hash.duplexing(M[row][col], |R|) \bmod R$
18:     end for
19: end for
20: ▷ Wrap-up: key computation
21: $Hash.absorb(\mathrm{pad}(salt))$    ▷ Uses the sponge's current state
22: $K \leftarrow Hash.squeeze(k)$
23: return $K$    ▷ Outputs the $k$-long key
```

sponge's bitrate (in bits). This is accomplished first by allowing the sponge to absorb the (properly padded) salt and password, initializing a highly random, *salt*- and *pwd*-dependent internal state (line 2). The padding rule adopted by Lyra is the multi-rate padding pad10*1 described in [9], hereby denoted simply pad, which appends a single bit 1 followed by as many bits 0 as necessary followed by a single bit 1.

Without resetting the state of the sponge, its (reduced) duplexing operation $Hash.duplexing_{\rho}$ is then repeatedly called until all rows of the memory matrix are filled (line 6). Here, "reduced" means that the duplexing may actually be done with a reduced-round version of f, denoted f_{ρ} for indicating that ρ rounds are executed rather than the regular number of rounds ρ_{max}. This approach accelerates the duplexing operation and, thus, allows more memory positions to be covered in the same amount of time than what would be possible with the application of a full-round f. Using reduced-round primitives in the core of cryptographic constructions is not unheard in the literature, as it is the main idea behind the Alred family of message authentication algorithms [17, 18, 42, 43]. As further discussed in Section 4.2, even though the requirements in the context of KDFs are different, this strategy does not decrease the security of the scheme as long as f_ρ is non-cyclic and highly non-linear, which should be the case for the vast majority of secure hash functions. After the memory matrix is completely filled, the sponge's internal state is not reset to zeros, but once again kept so it can be used in the next phase.

The most resource consuming of all phases, the *Wandering Phase* (lines 10 – 19), takes place after the Setup phase is finished. A total of $(T \cdot R)$ rows of the memory matrix are iteratively visited, R rows per iteration of the loop starting at line 11, called the *Time Loop*. Every row visited has all of its cells read and combined with the output of the underlying sponge's (reduced) duplexing operation $Hash.duplexing_\rho$ (line 14).

The visitation order in the Wandering phase is determined by the *row* internal variable of the algorithm. Since *row* is initialized to zero in line 10, the first row $M[0]$ is always visited first. The remainder rows are then visited in a pseudo-random fashion, as the *row* variable is updated after each visit (line 17). This update is done by fully duplexing one of the cells in the most recently visited row, which results in a highly random value of *row* while further increasing the entropy of the underlying sponge's state. The index of the cell chosen in this manner is $M[row][C-1] \bmod C$, so it can only be determined after the last cell of the corresponding row is actually processed. This entire process is intended to ensure that all positions of the memory matrix need to remain available for the whole duration of the key derivation process.

Finally, in the *Wrap-up Phase* (lines 21 – 22), the final key is computed by first absorbing the salt one last time and then squeezing the (full-round) sponge, once again using its current internal state. The number of bits generated in this manner, k, can be as arbitrary as allowed by the underlying sponge. The stateful, full-round sponge employed in this last stage ensures that the whole process is both non-invertible and of sequential nature.

4.2 Strictly sequential design

Like with PBKDF2 and other existing KDF, Lyra's design is strictly sequential, as the sponge's internal state is iteratively updated whenever it processes a cell of the memory matrix. Specifically, and without loss of generality, assume that the sponge's state before processing the cell $c_i = M[row][col + i]$ is s_i; then, after c_i is processed and becomes c'_i, the updated state is s_{i+1}. Now, suppose the attacker wants to parallelize the Columns Loop (lines 13 – 15), processing $\{c_0, c_1, c_2\}$ faster than sequentially computing $c'_0 = s_0$, $c'_1 = s_0 \oplus c_0$, $c'_2 = c'_1 \oplus c_1$, $s_3 = c'_2 \oplus c_2$.

If the sponge's transformation f was affine, the above task would be quite easy. For example, if f was the identity function, the attacker could use four processing cores of a GPU to make $x = s_0 \oplus c_0$, $y = c_0 \oplus c_1$, $z = c_1 \oplus c_2$ in parallel and then, in a second step, make $c'_0 = s_0$, $c'_1 = x$, $c'_2 = x \oplus c_1$, $s_3 = x \oplus z$. With an FPGA 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 iterations of f_ρ. Interestingly, if f_ρ has some obvious cyclic behavior, always resetting the sponge to a known state s_0 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 the Wandering phase creates several "serial bottlenecks" that prevent cells from being visited 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 to compute s_1 faster than actually processing 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 apply them to 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 results obtained with the correct instantiation. At first sight, it might seem that a reduced-round f facilitates this task, since the consecutive states s_0 and s_1 may share some bits or relationships between bits, thus reducing the number of possibilities that need to be included among the guessed states. Even if that is the case, however, any transformation f is expected to have a complex relationship between the input and output of every single round and, to speed-up the visitation of a cell, 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. In addition, 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 simply running $v \cdot \rho$ rounds of f. 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 Rows Loop and the Time Loop. The difference for the former is that, to determine the next row to be visited and start processing it in parallel, the attacker needs to find the internal state that will result from the visitation of the current row without actually visiting it, which involves $C \cdot \rho$ rounds of f. For the latter, the state to be determined would be that resulting from the visitation of several rows chosen in a random fashion, which involves $C \cdot R \cdot \rho$ rounds of f.

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

4.3 Configuring memory usage and processing time

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

Ignoring ancillary operations, the processing cost of Lyra is basically determined by the number of calls to the sponge's underlying f function. Hence, considering all of the algorithm's phases and supposing that both $(|pwd| + |salt|)$ and k are smaller than b, Lyra's total cost is approximately: $1+(R-1) \cdot C \cdot \rho/\rho_{max}$ for the Setup phase, plus $T \cdot R \cdot (1 + C \cdot \rho/\rho_{max})$ for the Wandering phase, plus 2 for the Wrap-up phase, leading roughly to $(T+1) \cdot R \cdot C \cdot \rho/\rho_{max}$ calls to f. Therefore, while the amount of memory used by the algorithm imposes a lower bound on its total running time, the latter can be linearly increased without affecting the former by choosing a suitable T parameter. This allows users to explore the most abundant resource in a (legitimate) platform with unbalanced availability of memory and processing power. This design also allows Lyra to use more memory than scrypt for a similar processing time: while scrypt employs a full-round hash for processing each of its elements, Lyra employs a reduced-round, faster operation for the same task.

4.4 On the underlying sponge

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

Our recommendation, thus, is toward using a secure software-oriented algorithm with low parallelism as the sponge's f transformation. One example is Blake2b's compression function [5], which has been shown to be a good permutation [4, 30] and displays a security level similar to that of Keccak [13].

4.5 Practical considerations

Lyra 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. Therefore, we expect the algorithm to be very easily implementable in software, especially if a sponge function is already available.

Lyra's memory matrix was also designed to allow users to take advantage of memory hierarchy features, such as caching and prefetching. As observed in [36], 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 for which $R = 1$ is likely to be visited faster than a matrix having $C = 1$, even for identical values of $R \cdot C$.

Therefore, by choosing adequate R and C values, Lyra 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.

Another practical concern taken into account in Lyra 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 2 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 the device is stolen [23, 49].

4.6 Parallelism on legitimate platforms: the Lyra$_p$ variant

Even though a strictly sequential KDF 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 KDF's usage of memory, abundant in a desktop or GPU running a single KDF instance, while (2) keeping the KDF's total processing time within humanly acceptable limits.

Against an attacker making several guesses in parallel, this strategy instantly raises the memory costs proportionally to the number of cores used by the legitimate user. For example, if the key is computed from a sequential KDF that uses 10 MB of memory and takes 1 second to run in a single core, an attacker who has access to 1,000 processing cores and 10 GB of memory could make 1,000 password guesses per second (one per core). If the key is now computed from the output of instances of the KDF, testing a guess would take 20 MB and 1 second, meaning that the attacker would need 20 GB of memory in order 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 Lyra. We call this variant Lyra$_p$, where the $p \geqslant 1$ parameter is the desired degree of parallelism. Lyra$_p$'s operation is as follows. During the Setup phase, p sponge copies are generated, each of which is responsible for initializing and processing a single $R \times C$ matrix. This is done similarly to $Lyra$, the difference being that each sponge i ($0 \leqslant i \leqslant p-1$), right after being bootstrapped in line 2 of Algorithm 2, must perform $p-1$ full-round and stateful absorb operations on an extra block of value $\text{pad}(i)$, where i is represented as a $|p-1|$-bit value. For example, for $p = 2$, the first sponge absorbs once a properly padded bit 0, while the second sponge does the same for a single bit 1; for $p = 4$, the padded 2-bit representations of integers 0, 1, 2, and 3 are absorbed three times by each sponge. 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 number of absorb operations performed by each sponge depends on p ensures that computations made with $p' \neq p$ cannot be reused in an attack against Lyra$_p$, an interesting property for scenarios in which the attacker does not know the correct value of p. The rest of the Setup phase (lines 4 – 8 of Algorithm 2) proceeds as usual for each of the p sponges, and so do their own Wandering and Wrap-up phases. The operation of these sponges can, thus, be fully parallelized, with each processing thread taking approximately the same amount of memory for a target processing time. After all sponges finish processing, the p sub-keys generated are XORed together, yielding the KDF's output K.

It is important to notice that, for $p = 1$, Lyra$_p$ behaves exactly like Lyra, meaning that both algorithms are fully compatible. Therefore, Lyra itself can be seen as a shorthand for Lyra$_1$.

5 Security analysis

Lyra's design is such that (1) the derived key is non-invertible, due to the initial and final hashing of *pwd* and *salt*; (2) attackers are unable to parallelize Algorithm 2 using multiple instances of the cryptographic sponge $Hash$, so they cannot significantly speed up the process of testing a password by means of multiple processing cores; (3) once initialized, the memory matrix needs to remain available during most of the key derivation process, meaning that the optimal operation of Lyra requires enough (fast) memory to hold its contents.

For better performance, a legitimate user is likely to store the whole memory matrix in volatile memory, facilitating its access in each of the several iterations of the Wandering and Wrap-up phases. An attacker running multiple instances of Lyra, on the other hand, may decide not to do the same, but 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 Lyra, the goal with this strategy is to allow more guesses to be independently tested in parallel, thus raising the overall throughput of the process. There are basically two methods for accomplishing this. The first is to trade memory for processing time, i.e., storing only the sponge's internal state after each row is processed and recomputing the next row to be visited from scratch, when (and only when) it becomes necessary; we call this a *Low-Memory attack*. The second it to use low-cost (and, thus, slower) storage devices, such as magnetic hard disks, which we call a *Slow-Memory attack*.

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

5.1 Low-Memory attacks

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

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

Proof. The attacker runs the loop for initializing the memory array V (lines $9 - 11$), which we call $ROMix_{ini}$. Instead of storing the values of V_i, however, the attacker keeps only the value of the internal variable X. Then, whenever an element V_j of V should be read (line 14 of Algorithm 1), the attacker simply runs $ROMix_{ini}$ for j iterations, determining the value of V_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 Lyra would proceed as follows. First, the attacker runs the Setup phase, but stores only the sponge's resulting internal state. Since *row* is set to 0 at the start of the Wandering phase, the first row is always initially visited and, thus, only *salt* and *pwd* are required the very first time the Columns Loop is executed.

When line 17 is reached for the first time and the *row* variable is updated to a random r, the attacker can run the Setup phase for r iterations, keeping in memory only $M[r]$ rather than the entire memory matrix. After $M[r]$ is processed, its value can be removed from memory to give space for the next row to be visited. However, those subsequent visits have an extra complicating factor:

if the row to be visited, $M[r']$, has been previously visited and, thus, modified, simply running (part of) the Setup phase will give the attacker an outdated value of $M[r']$. To obtain the actual value to be fed to the sponge with a low memory cost, the attacker needs to run the whole Setup phase and also all steps of the Wandering phase prior to the last visitation of $M[r']$. Nonetheless, since such steps of the Wandering phase may once again depend on rows that have been modified after Setup, those rows need to be recomputed as well, leading to recursive calls that grow in number as the algorithm progresses and more rows are modified.

Providing a tight bound on the complexity of such attack against Lyra is, thus, an involved task. Indeed, the attack can be accelerated by keeping in memory some intermediate states, adding even more variables to the analysis. Nevertheless, aiming to give some insight on how the attacker could (but is unlikely to want to) utilize such memory-processing trade-offs, in what follows we consider some slightly simplified attack scenarios. In each scenario, we try to match the total memory or processing complexities appearing in Theorem 1, which allows a more direct comparison between Lyra and scrypt's security when the attacker tries to take advantage of such trade-offs.

Preliminaries. Following the notation shown in Algorithm 2, let s^i_j denote the state of the sponge when the Time Loop and Rows Loop control variables are i and j, respectively, and before the corresponding row is effectively visited. Let $M^j[r]$ denote the r-th row of the memory matrix during the j-th iteration of the Time Loop, once again considering that this row has not yet been visited during this iteration. Finally, let $M^j[r_{(i,j)}]$ denote the row that is visited when the sponge's internal state is s^i_j. This is illustrated in Figure 3.

To simplify the analysis, we consider that each row is visited only once during each Time Loop, meaning that every row is visited per iteration j and all rows are visited T times in total. We argue that this is a reasonable simplification for an average-case analysis because, as the row visitation should follow a uniform distribution, all rows are expected to be visited approximately the same number of times.

For conciseness, we also ignore the small difference in cost between creating a row in the Setup phase and visiting it in the Wandering phase, using simply σ to denote both processes. In this manner, the cost of σ is approximately $C \cdot \rho/\rho_{max}$ calls to f.

Scenario 1: storing all intermediate states. We first consider a simple attack, in which the adversary never stores a given row $M[r]$, but instead stores the sponge's state right before that row is processed. Then, whenever that row is required during iteration j, he/she computes $M^0[r]$ from the *salt* and *pwd* — making r calls to σ, — uses the j previously stored states for computing $M^j[r]$ — with j extra iterative calls to σ — and finally proceeds with the Wandering phase as usual — calling σ once again for visiting $M^j[r]$. The idea behind this approach is that, since a sponge's state is usually smaller than an entire row

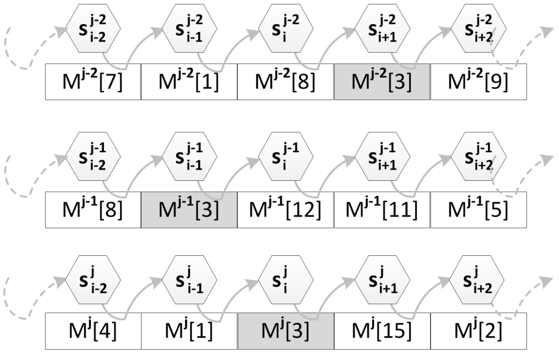

Fig. 3. Simplified example of Lyra's operation, showing part of the Wandering phase. Highlighted cells represent rows that are computed in sequence, considering that a single visit is made to them in each iteration of the Time Loop.

of the matrix (assuming that $w < b \cdot C$), the attacker might be able to reduce the total amount of memory required by the algorithm storing the former rather than the latter, paying the toll in terms of processing.

For example, in the scenario shown in Figure 3, the attacker could decide never to store $M[3]$, but only the sponge's states prior to its visitations. In that case, among the states shown, s_{i+1}^{j-2} and s_{i-1}^{j-1} would be in storage when the Wandering phase reaches its j-th iteration. At that point, the i-th row to be visited is $M^j[r_{(i,j)}] = M^j[3]$, which needs to be computed from scratch. This is done by running the Setup phase until $M^0[3]$ is obtained and then by iteratively running σ with the sponge states stored — e.g., $M^j[3]$ is obtained by calling σ with state s_{i-1}^{j-1} over $M^{j-1}[3]$, and the latter by calling σ with state s_{i+1}^{j-2} over $M^{j-2}[3]$.

The total cost of repeating this strategy for all $0 \leqslant j < T$ and any single row is, thus, $(R/2) \cdot T + T(T-1)/2$ calls to σ on average, and the storage of $(T-1)$ intermediate sponge states. Extending this strategy so that no row is stored, but only sponge states, the cost is multiplied by R, becoming approximately $(R+T) \cdot R \cdot T/2$ calls to σ on average and $R \cdot (T-1)$ intermediate sponge states.

Storing only a few memory rows in addition to the sponge states does accelerate this process: for example, storing the strategically positioned rows $M^j[R/2]$ for all j does allow the computation of any single row using only $(R/4)$ calls to σ on average. This happens because any desired row visited before $M^j[R/2]$ during iteration j can be computed from it approximately twice faster than doing so from scratch. In this case, the total cost of processing a row drops to $(R/4) \cdot T + T(T-1)/2$ for all iterations $0 \leqslant j < T$. Generalizing this approach, the storage of n rows should lead to a total cost of $R((R/2n) \cdot T + T(T-1)/2)$ calls to σ on average for the whole process. However, since any row is C times larger than a state, the total memory cost becomes equivalent to $R \cdot (T-1) + n \cdot T \cdot C$ intermediate sponge states.

These observations allow the formulation of Theorem 2. This theorem shows that, using the above strategy so that no row is stored, the attacker already raises its processing costs quadratically on parameter R, the best achievable with a memory-hard algorithm such as scrypt. However, and unlike what happens with scrypt, such trade-off is not enough to reduce the algorithm's memory usage to $\mathcal{O}(1)$; instead, the total memory cost remains quite high.

Theorem 2. *Consider that Lyra operates with parameters T, R and C. Whilst the regular algorithm's memory and processing costs are, respectively, $\mathcal{O}(R \cdot C)$ bits and $\mathcal{O}(T \cdot R)$ calls to σ, one can achieve a memory cost of $\mathcal{O}(R \cdot T)$ bits by raising the processing cost to $\mathcal{O}((R + T) \cdot R \cdot T)$.*

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

Scenario 2: storing a few intermediate states or none. The attacker can reduce the memory costs even further by storing a smaller number of intermediate states, computing them on demand as done with the rows of the memory matrix. Specifically, the attacker can determine state s^j_{i+1} if he/she knows the immediately previous state s^j_i and the row visited by the sponge in this latter state, $M^j r_{(i,j)}$. By recursively doing so, all iterations of the algorithm can be computed with minimal storage, reaching $\mathcal{O}(1)$. As shown in Theorem 3, however, the computational cost of this process is much higher than what would be attainable with a memory-hard algorithm.

Theorem 3. *Consider that Lyra operates with parameters T, R and C. Whilst the regular algorithm's memory and processing costs are, respectively, $\mathcal{O}(R \cdot C)$ bits and $\mathcal{O}(T \cdot R)$ calls to σ, one can achieve a memory cost of $\mathcal{O}(1)$ bits by raising the processing cost to $\mathcal{O}(R^{T+1})$.*

Proof. Suppose the attacker stores only the very last state computed, as well as the intermediate states obtained at the beginning of each Wandering phase, s^j_0, discarding all others. When the algorithm enters the Wandering phase (i.e., for $j = 0$), the attacker can obtain state s^0_{i+1} from s^0_i by computing the corresponding row $M^0[r_{(i,0)}]$ from scratch. This means that the Setup phase is run for $r_{(i,0)}$ iterations, leading to an average cost of $R/2$ calls to σ for each value of i. For $j = 1$, however, the average cost of such step grows to $R^2/4$: computing s^1_{i+1} from s^1_i requires the computation of $M^1[r_{(i,1)}]$ from s^1_0 (the nearest known state), meaning that the Setup phase is run $i+1$ times, once for each $M^1[r_{(\alpha,1)}]$, where $0 \leqslant \alpha \leqslant i$.

Following the same principle above, the average cost of computing a single state update s^j_{i+1} from s^j_i, is given by $(R/2)^{j+1}$ calls to σ, leading to $R \cdot (R/2)^{j+1}$ calls during the whole iteration j. For the last iteration of the Wandering phase alone, the total cost is $R \cdot (R/2)^T$, which should dominate Lyra's running time in this memory-free scenario. □

Summary. At this point, it becomes easier to see that Lyra provides a higher level of security than scrypt (and memory-hard algorithms in general), even for small values of T. For example, suppose that Lyra runs as fast as scrypt for some $T \geqslant 2$, and that both algorithms operate over the same number of memory elements R, yielding identical memory usages. In this case, a memory-free attack (Scenario 2) against Lyra has a complexity of $\mathcal{O}(R^{T+1}) \geqslant \mathcal{O}(R^3)$, while against scrypt this cost would be $\mathcal{O}(R^2)$. On the other hand, if the attacker can go only as far as raising the processing cost quadratically (Scenario 1), he/she would be able to attack scrypt with a $\mathcal{O}(1)$ memory cost. However, the memory costs in the same conditions would be far from dropping that much against Lyra.

5.2 Slow-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.

Lyra, similarly to scrypt, explores the properties of low-cost memory devices by visiting memory positions in a pseudo-random pattern. In particular, this strategy increases the latency of intrinsically sequential memory devices, such as hard disks, especially if the attack involves multiple instances simultaneously accessing different memory sections. Furthermore, as discussed in Section 4.5, such visitation 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.

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

Another appealing aspect about Lyra's design is the fact that its Wandering phase XORs the value of a row with the sponge's output, preventing the memory positions corresponding to that row from becoming quickly replaceable. This property is, thus, likely to hinder the attacker's capability of reusing that memory region in a parallel thread. Obviously, such properties may also impact a legitimate user, stressing the need of configuring the R, C and T parameters according to the target platform.

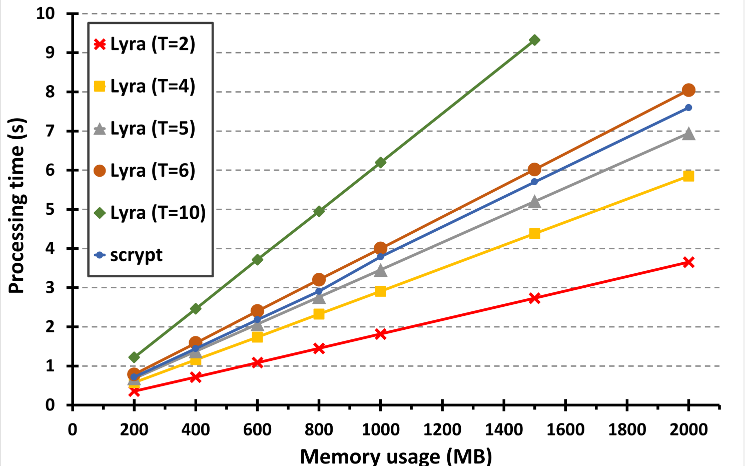

Fig. 4. Lyra's performance for $C = 64$, $\rho = 1$, and different T and R settings.

6 Performance for recommended parameters

In a preliminary assessment of Lyra's performance in software, we used a reference implementation of Blake2b's compression function [5] as the underlying sponge's f function. The implementations employed are available at `www.lyra-kdf.net`.

The results are depicted in Figure 4, in which Lyra is parameterized with $C = 64$, $\rho = 1$ 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, Lyra can run in less 1 s while using 200 MB of memory (with $R = 6.4 \cdot 10^4$), or in less than 5 s with 1GB (with $R = 3.2 \cdot 10^5$). All tests were performed on an Intel Core i5-24000 (3.10 GHz Quad Core, 64 bits) equipped with 8 GB of DRAM, running Windows 7 64 bits and using the MinGW64 compiler with `-O3` optimization.

Figure 4 also compares Lyra with the scrypt "optimized non-SSE2" implementation publicly available at `www.tarsnap.com/scrypt.html`, using the parameters suggested by scrypt's author in [36] (namely, $b = 8192$ and $p = 1$). The "non-SSE2" version of scrypt was chosen aiming at a fair comparison, since our preliminary Lyra implementation does not explore SSE2 instructions either. The results obtained show that, in order to achieve a memory usage and processing time similar to that of scrypt, Lyra should be parameterized with $T \approx 6$.

6.1 Expected attack costs

Considering that the cost of DDR3 SO-DIMM memory chips is currently around U$5.00/GB [45], Table 1 shows the cost added by Lyra with $T = 5$ when an attacker tries to crack a password in 1 year using the above reference hardware, for different password strengths. 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, ignoring costs related to wiring and energy consumption and supposing that testing a password takes the same amount of time as in our testbed. We refer the reader to [33, Appendix A] for a discussion on how to compute the entropy of passwords.

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 higher due to the higher cost per GB of such memory devices. All in all, the numbers provided in Table 1 are not intended as absolute values, but rather a reference on how much extra protection one could expect from using Lyra, since this additional memory-related cost is the main advantage of KDFs that explore memory usage when compared with those that do not.

Finally, when compared with existing solutions that do explore memory usage, Lyra is advantageous due to the elevated processing costs of attack venues in which attackers try to avoid memory-related costs, effectively discouraging such approaches. Indeed, considering the final Wandering phase alone and $T = 5$, the additional processing cost of a memory-free attack against Lyra is approximately $(6.4 \cdot 10^4)^6 = 6.9 \cdot 10^{28}$ calls to σ if the algorithm operates with 200 MB, or $(3.2 \cdot 10^5)^6 = 1.1 \cdot 10^{33}$ for a memory usage of 1GB. For the same memory usage settings, the total cost of a similar memory-free attack against scrypt would be approximately $(2 \cdot 10^5)^2 = 4 \cdot 10^{10}$ and $(1 \cdot 10^6)^2 = 1 \cdot 10^{12}$ calls to *BlockMix*, whose processing time is quite similar to that of σ for the parameters used in our experiments. As expected, such elevated processing costs are prone to discourage attack venues that try to avoid the memory costs of Lyra by means of extra processing.

Password	Memory usage (MB)				
entropy (bits)	200	600	1,000	1,500	2,000
35	877.2	6.9k	19.3k	43.5k	77.5k
40	28.1k	221.6k	616.5k	1.4M	2.5M
45	898.3k	7.1M	19.7M	44.6M	79.3M
50	28.8M	226.9M	631.3M	1.4B	2.5B
55	919.9M	7.3B	20.2B	45.7B	81.2B

Table 1. Memory-related cost (in U$) added by Lyra with $T = 6$, for attackers trying to break passwords in a 1-year period using an Intel Core i5-2400.

7 Conclusions

We presented Lyra, a password-based key derivation scheme 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, Lyra 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 Lyra's memory usage allows legitimate users to thwart attacks using parallel platforms. This can be accomplished by raising the 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 memory-processing 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 Lyra an appealing KDF alternative.

As future work, we plan to provide detailed performance analyses of Lyra and Lyra$_p$ in different (software and hardware) platforms, using different underlying sponges.

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 grant 303163/2009-7, as well as by the São Paulo Research Foundation (FAPESP) under grant 2011/21592-8.

References

1. Andreeva, E., Mennink, B., Preneel, B.: The Parazoa family: Generalizing the Sponge hash functions. IACR Cryptology ePrint Archive **2011**, 28 (2011)
2. Apple: iOS security. Tech. rep., Apple Inc. (2012). `http://images.apple.com/ipad/business/docs/iOS_Security_May12.pdf`
3. 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, vol. 5084, pp. 470–488. Springer-Verlag, Berlin, Heidelberg (2008). DOI 10.1007/978-3-540-71039-4_30
4. Aumasson, J.P., Guo, J., Knellwolf, S., Matusiewicz, K., Meier, W.: Differential and invertibility properties of BLAKE. In: Fast Software Encryption, pp. 318–332. Springer (2010). URL `http://eprint.iacr.org/2010/043.pdf`
5. Aumasson, J.P., Neves, S., Wilcox-OHearn, Z., Winnerlein, C.: BLAKE2: simpler, smaller, fast as MD5. `https://blake2.net/blake2_20130129.pdf` (2013)
6. Bellare, M., Ristenpart, T., Tessaro, S.: Multi-instance security and its application to password-based cryptography. In: Advances in Cryptology (CRYPTO 2012), *LNCS*, vol. 7417, pp. 312–329. Springer Berlin Heidelberg (2012). DOI 10.1007/978-3-642-32009-5_19

7. Bernstein, D.: The Salsa20 family of stream ciphers. In: M. Robshaw, O. Billet (eds.) New Stream Cipher Designs, pp. 84–97. Springer-Verlag, Berlin, Heidelberg (2008). DOI 10.1007/978-3-540-68351-3_8. URL `http://dx.doi.org/10.1007/978-3-540-68351-3_8`
8. 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`
9. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: Cryptographic sponge functions - version 0.1. http://keccak.noekeon.org/ (2011)
10. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: The Keccak SHA-3 submission. Submission to NIST (Round 3) (2011). URL `http://keccak.noekeon.org/Keccak-submission-3.pdf`
11. 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, pp. 553–567 (2012). DOI 10.1109/SP.2012.44
12. Chakrabarti, S., Singbal, M.: Password-based authentication: Preventing dictionary attacks. Computer **40**(6), 68–74 (2007). DOI 10.1109/MC.2007.216
13. Chang, S.j., 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). URL `http://nvlpubs.nist.gov/nistpubs/ir/2012/NIST.IR.7896.pdf`
14. 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, pp. 225–236. IEEE Computer Society, Washington, DC, USA (2010). DOI 10.1109/MICRO.2010.36
15. 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), *HICSS'04*, vol. 7, pp. 170–179. IEEE Computer Society, Washington, DC, USA (2004). URL `\url{http://dl.acm.org/citation.cfm?id=962755.963150}`
16. Crew, B.: New carnivorous harp sponge discovered in deep sea. Nature (2012). DOI 10.1038/nature.2012.11789. Available online: `http://www.nature.com/news/new-carnivorous-harp-sponge-discovered-in-deep-sea-1.11789`
17. Daemen, J., Rijmen, V.: A new MAC construction ALRED and a specific instance ALPHA-mac. In: Fast Software Encryption – FSE'05, pp. 1–17 (2005). DOI 10.1007/11502760_1
18. Daemen, J., Rijmen, V.: Refinements of the ALRED construction and MAC security claims. Information Security, IET **4**(3), 149–157 (2010). DOI 10.1049/iet-ifs.2010.0015
19. 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), pp. 485–485. IEEE (2008). DOI 10.1109/HICSS.2008.484
20. Dürmuth, M., Güneysu, T., Kasper, M.: Evaluation of standardized password-based key derivation against parallel processing platforms. In: Computer Security–ESORICS 2012, *LNCS*, vol. 7459, pp. 716–733. Springer Berlin Heidelberg (2012). DOI 10.1007/978-3-642-33167-1_41

21. Florencio, D., Herley, C.: A large scale study of web password habits. In: Proc. of the 16th International Conference on World Wide Web, pp. 657–666. Alberta, Canada (2007)
22. 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), pp. 47–56. ACM, New York, NY, USA (2012). DOI 10.1145/2145694.2145704
23. 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), 91–98 (2009). DOI 10.1145/1506409.1506429
24. 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, *LNCS*, vol. 5628, pp. 230–237. Springer Berlin / Heidelberg (2009). DOI 10.1007/978-3-642-03549-4_14
25. 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), 179–195 (2006). DOI 10.1007/s11227-006-5682-5
26. Kaliski, B.: PKCS#5: Password-Based Cryptography Specification version 2.0 (RFC 2898) (2000). URL http://tools.ietf.org/html/rfc2898
27. 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, pp. 121–134. Springer-Verlag, London, UK, UK (1998)
28. Khronos Group: The OpenCL Specification – Version 1.2 (2012)
29. Marechal, M.: Advances in password cracking. Journal in Computer Virology **4**(1), 73–81 (2008). DOI 10.1007/s11416-007-0064-y
30. 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), pp. 783–786. IEEE (2010). URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5709204
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 (2002). http://csrc.nist.gov/publications/fips/fips198/fips-198a.pdf
32. NIST: Special Publication 800-18 – Recommendation for Key Derivation Using Pseudorandom Functions. National Institute of Standards and Technology, U.S. Department of Commerce (2009). http://csrc.nist.gov/publications/nistpubs/800-108/sp800-108.pdf
33. NIST: Special Publication 800-63-1 – Electronic Authentication Guideline. National Institute of Standards and Technology, U.S. Department of Commerce (2011). http://csrc.nist.gov/publications/nistpubs/800-63-1/SP-800-63-1.pdf
34. Nvidia: CUDA C programming guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/ (2012)
35. Nvidia: Tesla Kepler family product overview. http://www.nvidia.com/content/tesla/pdf/Tesla-KSeries-Overview-LR.pdf (2012)
36. Percival, C.: Stronger key derivation via sequential memory-hard functions. In: BSDCan 2009 – The Technical BSD Conference (2009). URL http://www.bsdcan.org/2009/schedule/attachments/87_scrypt.pdf

37. PHC: Password hashing competition. https://password-hashing.net/ (2013)
38. Provos, N., Mazières, D.: A future-adaptable password scheme. In: Proc. of the FREENIX track: 1999 USENIX annual technical conference (1999)
39. Schneier, B.: Description of a new variable-length key, 64-bit block cipher (Blowfish). In: Fast Software Encryption, Cambridge Security Workshop, pp. 191–204. Springer-Verlag, London, UK (1994)
40. SciEngines: Rivyera s3-5000. http://sciengines.com/products/computers-and-clusters/rivyera-s3-5000.html
41. SciEngines: Rivyera v7-2000t. http://sciengines.com/products/computers-and-clusters/v72000t.html
42. Simplicio Jr, 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**, 165–180 (2009). DOI 10.1002/sec.66
43. Simplicio Jr, M.A., Barreto, P.S.L.M.: Revisiting the security of the ALRED design and two of its variants: Marvin and LetterSoup. IEEE Transactions on Information Theory **58**(9), 6223–6238 (2012). DOI 10.1109/TIT.2012.2203093
44. 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). URL http://www.ru.nl/publish/pages/578936/thesis.pdf
45. TrendForce: DRAM contract price (jan.15 2013). http://www.trendforce.com/price (visited on Apr.22, 2013) (2013)
46. TrueCrypt: TrueCrypt: Free open-source on-the-fly encryption – documentation. http://www.truecrypt.org/docs/ (2012)
47. 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, pp. 391–405. IEEE Computer Society, Washington, DC, USA (2009). DOI 10.1109/SP.2009.8
48. Yao, F., Yin, Y.: Design and analysis of password-based key derivation functions. IEEE Transactions on Information Theory **51**(9), 3292–3297 (2005). DOI 10.1109/TIT.2005.853307
49. Yuill, J., Denning, D., Feer, F.: Using deception to hide things from hackers: Processes, principles, and techniques. Journal of Information Warfare **5**(3), 26–40 (2006)

Appendix A: PBKDF2

The Password-Based Key Derivation Function version 2 (PBKDF2) algorithm [26] was originally proposed in 2000 as part of RSA Laboratories' PKCS#5. It is nowadays present in several security tools, such as TrueCrypt [46] and Apple's iOS for encrypting user passwords [2], and has been formally analyzed in several circumstances [48, 6].

Basically, PBKDF2 (see Algorithm 3) iteratively applies the underlying pseudorandom function $Hash$ to the concatenation of pwd and a variable U_i, i.e., it makes $U_i = Hash(pwd, U_{i-1})$ for each iteration $1 \leqslant i \leqslant T$. The initial value U_0 corresponds to the concatenation of the user-provided $salt$ and a variable l, where l corresponds to the number of required output blocks. The l-th block of

Algorithm 3 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  $K \leftarrow K \mid\mid T[i]$  end if
end for
return $K$
```

the k-long key is then computed as $K_l = U_1 \oplus U_2 \oplus \ldots \oplus U_T$, where k is the desired key length.

PBKDF2 allows users to control its total running time by configuring the T parameter. Since the key derivation 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 key derivation's processing time is bcrypt [38]. The scheme is based on a customized version of the 64-bit cipher algorithm Blowfish [39], called *EksBlowfish* ("expensive key schedule blowfish").

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

EksBlowfishSetup starts by copying the first digits of the number π into the subkeys and S-boxes S_i (see Algorithm 4). Then, it updates the subkeys and S-boxes by invoking $ExpandKey(salt, pwd)$, for a 128-bit salt value. Basically, this function (1) cyclically XORs the password with the current subkeys, and then (2) iteratively blowfish-encrypts one of the halves of the salt, the resulting ciphertext being XORed with the salt's other half and also replacing the next two subkeys (or S-Boxes, after all subkeys are replaced). After all subkeys and S-Boxes are updated, bcrypt alternately calls $ExpandKey(0, salt)$

Algorithm 4 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 $\pi$ 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 $T \,\|\, salt \,\|\, ctext$
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
```

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: On the algorithm's name

The name Lyra comes from *Chondrocladia lyra*, a recently discovered type of sponge [16]. While most sponges are harmless, this harp-like sponge is carnivorous, using its branches to ensnare its prey, envelope it in membrane and completely digest it.

Lyra'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. ☺