EWERTON RODRIGUES ANDRADE

LYRA2: PASSWORD HASHING SCHEME WITH IMPROVED SECURITY AGAINST TIME-MEMORY TRADE-OFFS

LYRA2: UM ESQUEMA DE HASH DE SENHAS COM MAIOR SEGURANÇA CONTRA TRADE-OFFS ENTRE PROCESSAMENTO E MEMÓRIA

Tese apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Doutor em Ciências.

São Paulo
2016

EWERTON RODRIGUES ANDRADE

LYRA2: PASSWORD HASHING SCHEME WITH IMPROVED SECURITY AGAINST TIME-MEMORY TRADE-OFFS

LYRA2: UM ESQUEMA DE HASH DE SENHAS COM MAIOR SEGURANÇA CONTRA TRADE-OFFS ENTRE PROCESSAMENTO E MEMÓRIA

Tese apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Doutor em Ciências.

Área de Concentração:

Engenharia de Computação

Orientador:

Prof. Dr. Marcos A. Simplicio Junior

São Paulo
2016

Catalogação-na-publicação

Andrade, Ewerton Rodrigues
Lyra2: Password Hashing Scheme with improved security against time memory trade-offs (Lyra2: Um Esquema de Hash de Senhas com maior segurança contra trade-offs entre processamento e memória) / E. R. Andrade -- São Paulo, 2016.
135 p.

Tese (Doutorado) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais.

1.Metodologia e técnicas de computação 2.Segurança de computadores 3.Criptologia 4.Algoritmos 5.Esquemas de Hash de Senhas I.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Computação e Sistemas Digitais II.t.

RESUMO

Para proteger-se de ataques de força bruta, sistemas modernos de autenticação baseados em senhas geralmente empregam algum Esquema de Hash de Senhas (*Password Hashing Scheme* - PHS). Basicamente, um PHS é um algoritmo criptográfico que gera uma sequência de bits pseudo-aleatórios a partir de uma senha provida pelo usuário, permitindo a este último configurar o custo computacional envolvido no processo e, assim, potencialmente elevar os custos de atacantes testando múltiplas senhas em paralelo. Esquemas tradicionais utilizados para esse propósito são o PBKDF2 e bcrypt, por exemplo, que incluem um parâmetro configurável que controla o número de iterações realizadas pelo algoritmo, permitindo ajustar-se o seu tempo total de processamento. Já os algoritmos scrypt e Lyra, mais recentes, permitem que usuários não apenas controlem o tempo de processamento, mas também a quantidade de memória necessária para testar uma senha. Apesar desses avanços, ainda há um interesse considerável da comunidade de pesquisa no desenvolvimento e avaliação de novas (e melhores) alternativas. De fato, tal interesse levou recentemente à criação de uma competição com esta finalidade específica, a *Password Hashing Competition* (PHC). Neste contexto, o objetivo do presente trabalho é propor uma alternativa superior aos PHS existentes. Especificamente, tem-se como alvo melhor o algoritmo Lyra, um PHS baseado em esponjas criptográficas cujo projeto contou com a participação dos autores do presente trabalho. O algoritmo resultante, denominado Lyra2, preserva a segurança, eficiência e flexibilidade do Lyra, incluindo a habilidade de configurar do uso de memória e tempo de processamento do algoritmo, e também a capacidade de prover um uso de memória superior ao do scrypt com um tempo de processamento similar. Entretanto, ele traz importantes melhorias quando comparado ao seu predecessor: (1) permite um maior nível de segurança contra estratégias de ataque envolvendo trade-offs entre tempo de processamento e memória; (2) inclui a possibilidade de elevar os custos envolvidos na construção de plataformas de hardware dedicado para ataques contra o algoritmo; (3) e provê um equilíbrio entre resistância contra ataques de canal colateral ("*side-channel*") e ataques que se baseiam no uso de dispositivos de memória mais baratos (e, portanto, mais lentos) do que os utilizados em computadores controlados por usuários legítimos. Além da descrição detalhada do projeto do algoritmo, o presente trabalho inclui também uma análise detalhada de sua segurança e de seu desempenho em diferentes plataformas. Cabe notar que o Lyra2, conforme aqui descrito, recebeu uma menção de reconhecimento especial ao final da competição PHC previamente mencionada.

Palavras-chave: derivação de chaves, senhas, autenticação de usuários, segurança, esponjas criptográficas.

ABSTRACT

To protect against brute force attacks, modern password-based authentication systems usually employ mechanisms known as Password Hashing Schemes (PHS). Basically, a PHS is a cryptographic algorithm that generates a sequence of pseudorandom bits from a user-defined password, allowing the user to configure the computational costs involved in the process aiming to raise the costs of attackers testing multiple passwords trying to guess the correct one. Traditional schemes such as PBKDF2 and bcrypt, 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. The more recent scrypt and Lyra algorithms, on the other hand, allow users to control both processing time and memory usage. Despite these advances, there is still considerable interest by the research community in the development of new (and better) alternatives. Indeed, this led to the creation of a competition with this specific purpose, the Password Hashing Competition (PHC). In this context, the goal of this research effort is to propose a superior PHS alternative. Specifically, the objective is to improve the Lyra algorithm, a PHS built upon cryptographic sponges whose project counted with the authors' participation. The resulting solution, called Lyra2, preserves the security, efficiency and flexibility of Lyra, including: the ability to configure the desired amount of memory and processing time to be used by the algorithm; and (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 includes tweaks for increasing the costs involved in the construction of dedicated hardware to attack the algorithm; (3) it balances resistance against side-channel threats and attacks relying on cheaper (and, hence, slower) storage devices. Besides describing the algorithm's design rationale in detail, this work also includes a detailed analysis of its security and performance in different platforms. It is worth mentioning that Lyra2, as hereby described, received a special recognition in the aforementioned PHC competition.

Keywords: Password-based key derivation, passwords, authentication, security, cryptographic sponges.

CONTENTS

List of Figures viii

List of Tables x

List of Acronyms xi

List of Symbols xii

1 Introduction 13

1.1 Motivation 14

1.2 Goals and Original Contributions 16

1.3 Methods 17

1.4 Document Organization 18

2 Background 19

2.1 Hash-Functions 19

2.1.1 Security 20

2.2 Cryptographic Sponges 21

2.2.1 Basic Structure 21

2.2.2 The duplex construction 22

2.2.3 Security 23

2.3 Password Hashing Schemes (PHS) 23

2.3.1 Attack platforms 25

2.3.1.1 Graphics Processing Units (GPUs) 25

2.3.1.2 Field Programmable Gate Arrays (FPGAs) . . . 26

3 Related Works **28**

3.1 Pre-PHC Schemes 28

3.1.1 PBKDF2 29

3.1.2 Bcrypt 30

3.1.3 Scrypt 32

3.1.4 Lyra 34

3.2 Schemes from PHC 37

3.2.1 Argon2 38

3.2.2 battcrypt 40

3.2.3 Catena 40

3.2.4 POMELO 42

3.2.5 yescrypt 43

4 Lyra2 **45**

4.1 Structure and rationale 47

4.1.1 Bootstrapping 47

4.1.2 The Setup phase 48

4.1.3 The Wandering phase 52

4.1.4 The Wrap-up phase 54

4.2 Strictly sequential design 54
4.3 Configuring memory usage and processing time 57
4.4 On the underlying sponge 57
4.4.1 A dedicated, multiplication-hardened sponge: BlaMka. . . 59
4.5 Practical considerations 60

5 Security analysis **63**

5.1 Low-Memory attacks 64
5.1.1 Preliminaries 66
5.1.2 The Setup phase 67
5.1.2.1 Storing only what is needed: 1/2 memory usage . 68
5.1.2.2 Storing less than what is needed: 1/4 memory usage 69
5.1.2.3 Storing less than what is needed: 1/8 memory usage 71
5.1.2.4 Storing less than what is needed: generalization . 75
5.1.2.5 Storing only intermediate sponge states 78
5.1.3 Adding the Wandering phase: consumer-producer strategy 81
5.1.3.1 The first $R/2$ iterations of the Wandering phase with 1/2 memory usage. 82
5.1.3.2 The whole Wandering phase with 1/2 memory usage 85
5.1.3.3 The whole Wandering phase with less than 1/2 memory usage 87
5.1.4 Adding the Wandering phase: sentinel-based strategy . . . 88

5.1.4.1 On the (low) scalability of the sentinel-based strategy . . . 91

5.2 Slow-Memory attacks . . . 94

5.3 Cache-timing attacks . . . 96

5.4 Garbage-Collector attacks . . . 99

5.5 Security analysis of BlaMka. . . . 100

5.6 Summary . . . 101

6 Performance for different settings **103**

6.1 Benchmarks for Lyra2 . . . 103

6.2 Benchmarks for BlaMka. . . . 108

6.3 Benchmarks for Lyra2 with BlaMka . . . 112

6.4 Expected attack costs . . . 113

6.5 Summary . . . 114

7 Final Considerations **116**

7.1 Publications and other results . . . 116

7.2 Future works . . . 118

References **119**

Appendix A. Naming conventions **129**

Appendix B. Further controlling Lyra2's bandwidth usage **130**

Appendix C. An alternative design for BlaMka: avoiding latency **132**

LIST OF FIGURES

1 Overview of the sponge construction $Z = [f, \texttt{pad}, b](M, \ell)$. Adapted from (BERTONI *et al.*, 2011a). 21

2 Overview of the duplex construction. Adapted from (BERTONI *et al.*, 2011a). 22

3 Handling the sponge's inputs and outputs during the Setup (left) and Wandering (right) phases in Lyra2. 50

4 BlaMka's multiplication-hardened (right) and Blake2b's original (left) permutations. 59

5 The Setup phase. 68

6 Attacking the Setup phase: storing 1/2 of all rows. 68

7 Attacking the Setup phase: storing 1/4 of all rows. 70

8 Attacking the Setup phase: recomputing $M[6_A]$ while storing 1/8 of all rows and keeping $M[F]$ in memory. 74

9 Attacking the Setup phase: storing only sponge states. 78

10 Reading and writing cells in the Setup phase. 80

11 An example of the Wandering phase's execution. 81

12 Tree representing the dependence among rows in Lyra2. 86

13 Tree representing the dependence among rows in Lyra2 with $T = 2$: using ϵ' sentinels per level. 92

14 Performance of SSE-enabled Lyra2, for $C = 256$, $\rho = 1$, $p = 1$, and different T and R settings, compared with SSE-enabled scrypt. 104

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. 105

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. 106

17 Performance of SSE-enabled Lyra2 with BlaMka, for $C = 256$, $\rho = 1$, $p = 1$, and different T and R settings, compared with SSE-enabled scrypt and memory-hard PHC finalists 112

18 Different permutations: Blake2b's original permutation (left), BlaMka's $G_{\texttt{tls}}$ multiplication-hardened permutation (middle) and BlaMka's latency-oriented multiplication-hardened permutation $G_{\texttt{tls}^{\oplus}}$ (right). 132

19 Improving the latency of G. 134

LIST OF TABLES

1 Indices of the rows that feed the sponge when computing $M[row]$ during Setup (hexadecimal notation). 52

2 Security overview of the PHSs considered the state of art. 102

3 PHC finalists: calls to underlying primitive in terms of their time and memory parameters, T and M, and their implementations. . 107

4 Data related of the tests performed in CPU, executing just one round of G function (i.e., 256 bits of output). 109

5 Data related of the initial tests performed in FPGA, executing just one round of G function (i.e., 256 bits of output). 111

6 Data related of the initial tests performed in dedicated hardware (that present advantage against CPU), executing just one round of G function (i.e., 256 bits of output). 111

7 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. 114

LIST OF ACRONYMS

GPU	Graphics Processing Unit
FPGA	Field-Programmable Gate Array
KDF	Key Derivation Function
PHS	Password Hashing Scheme
PHC	Password Hashing Competition
ASIC	Application Specific Integrated Circuit
HMAC	Hash-based Message Authentication Code
RAM	Random Access Memory
TMTO	Time-Memory trade-offs
PBKDF2	Password-Based Key Derivation Function 2
bcrypt	Blowfish crypt
XOR	Excusive-OR operation
SHA	Secure Hash Algorithm
AES	Advanced Encryption Standard
CUDA	Compute Unified Device Architecture
OpenCL	Open Computing Language
SIMD	Single Instruction, Multiple Data
SSE	Streaming SIMD Extensions
AVX	Advanced Vector Extensions

LIST OF SYMBOLS

$\oplus$	bitwise Exclusive-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

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 (CHAKRABARTI; SINGBAL, 2007; CONKLIN; DIETRICH; WALZ, 2004). For better or for worse, and despite the existence of many proposals for their replacement (BONNEAU *et al.*, 2012), 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) (NIST, 2009). Typically, the output of the PHS is employed in one of two manners (PERCIVAL, 2009): 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 (PERCIVAL, 2009; KALISKI, 2000).

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 (NIST, 2011). Indeed, a study from 2007 with 544,960 passwords from real users has shown an average entropy of approximately 40.5 bits (FLORENCIO; HERLEY, 2007), 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 (CHAKRABARTI; SINGBAL, 2007; HERLEY; OORSCHOT; PATRICK, 2009), 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 (DÜRMUTH; GÜNEYSU; KASPER, 2012; SPRENGERS, 2011) or custom hardware (DÜRMUTH; GÜNEYSU; KASPER, 2012; MARECHAL, 2008).

1.1 Motivation

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 (CHAKRABARTI; SINGBAL, 2007). For this reason, modern password hashing solutions usually employ mechanisms for increasing the *cost* of brute force attacks. Schemes such as PBKDF2 (KALISKI, 2000) and bcrypt (PROVOS; MAZIÈRES,

1999), 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 (PERCIVAL, 2009), 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. Since this may raise the RAM costs of password cracking to unbearable levels, attackers may try to trade memory for processing time, discarding (parts of) the memory used and recomputing the discarded information when (and only when) it becomes necessary (PERCIVAL, 2009). The exploitation of such time-memory trade-offs (TMTO) leads to the hereby-called *low-memory attacks*. Another approach that might be used by attackers trying to reduce the costs of password cracking is to use low-cost (and, thus, slower) storage devices for keeping all memory used in the legitimate process, using the spare budget to run more tests in parallel and, thus, compensating the lower speed of each test; we call this approach a *slow-memory attack*.

Besides the need for protection against low- and slow-memory attacks, there is also interest in the development of solutions that are safe against side-channel attacks, in especial the so-called *cache-timing attacks*. Basically, a cache-timing attack is possible if the attacker can observe a machine's timing behavior by monitoring its access to cache memory (e.g., the occurrence of cache-misses), building a profile of such occurrences for a legitimate password hashing process (FORLER; LUCKS; WENZEL, 2013; BERNSTEIN, 2005). Then, at least in theory, if the password being tested does not match the observed cache-timing behavior, the test could be aborted earlier, saving resources. Although this class of attack has not been effectively implemented in the context of PHSs, it has been shown to be effective, for example, against certain implementations of the Advanced Encryp-

tion Standard (AES) (NIST, 2001a) and RSA (RIVEST; SHAMIR; ADLEMAN, 1978).

The considerable interest by the research community in developing new (and better) password hashing alternatives has recently even led to the creation of a cryptographic competition with this specific purpose, the Password Hashing Competition (PHC) (PHC, 2013).

1.2 Goals and Original Contributions

Aiming to address this need for stronger alternatives, our early studies led to the proposal of Lyra (ALMEIDA *et al.*, 2014), a mode of operation of cryptographic sponges (BERTONI *et al.*, 2007; BERTONI *et al.*, 2011a) for password hashing. In this research, we propose an improved version of Lyra, called simply Lyra2.

Basically, Lyra2 preserves the flexibility and efficiency of Lyra, including:

1. The ability to configure the desired amount of memory and processing time to be used by the algorithm;
2. The capacity of providing a higher memory usage than what is obtained with scrypt for a similar processing time.

In addition, it brings important security improvements when compared to its predecessor:

1. It allows a higher security level against attack venues involving time-memory trade-offs (TMTO);
2. It includes tweaks to increase the costs involved in the construction of dedicated hardware for attacking the algorithm (e.g., FPGAs or ASICs);

3. It balances resistance against side-channel threats and attacks relying on cheaper (and, hence, slower) storage devices.

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, Lyra2 allows for a higher memory usage than its counterparts, further raising the costs of any possible attack venue.

1.3 Methods

The method adopted in this work is the applied research based on the hypothesis-deduction approach, i.e., by using scientific references to define the problem, specify the solution hypotheses and, finally, evaluate them (WAZLAWICK, 2008).

For accomplishing this, the work was separated according to the following steps:

- *Literature research*: survey of existing PHS, based on the analysis of academic articles and technical manuals. This included a comparison between existing solutions and the evaluation of their internal structures, security and performance, making it possible to determine attractive approaches to create a novel algorithm;

- *Design of algorithm*: using as basis the literature research, this step consists in the proposal of a novel PHS, called Lyra2. This new algorithm preserves the flexibility of existing functions (including its predecessor, Lyra), but

provides higher security. This step also involves the development of a reference implementation for allowing validation of its viability and comparison with existing PHS solutions;

- *Evaluation*: comparison between the structures of the current PHSs and the Lyra2 in order to verify that, (1) by construction, Lyra2's security is higher than that provided by existing PHS, and (2) its performance is at least as good as that of alternative solutions;

- *Thesis writing*: creation of a thesis encompassing the obtained results and analyses.

These steps are rather sequential, with frequent iterations between them (e.g., performance measurements usually lead to improvements to the algorithm's design).

1.4 Document Organization

The rest of this document is organized as follows. Chapter 2 describes the basic notation and outlines the concept of hash functions, cryptographic sponges and password hashing schemes, describing the main requirements of theses algorithms. Chapter 3 discusses the related work. Chapter 4 introduces Lyra2's core and its design rationale, while Chapter 5 analyzes its security. Chapter 6 presents our benchmark results and comparisons with existing PHS. Finally, Chapter 7 encloses our concluding remarks, main results and plans for future work.

2 BACKGROUND

To allow a better comprehension of the concepts explored in this document, it is necessary to clearly understand some basic concepts involved in the area of Cryptography. This is the main goal of this chapter, which covers the basic services provided by hash functions and how they can be implemented using the concept of cryptographic sponges. In addition, it also summarizes the characteristics, utilization and main security aspects of Password Hashing Schemes (PHS), which are the focus of this research.

In what follows and throughout this document, we use the notation and symbols shown on "List of Symbols" (page xii).

2.1 Hash-Functions

Let H be a function, we call H as Hash-Function if $H\colon \{0,1\}^* \mapsto \{0,1\}^h$, where $h \in \mathbb{N}$ (TERADA, 2008). In other words, a Hash-Function H is a one-way and non-invertible transformation that maps an arbitrary-length input x to a fixed h length output $y = H(x)$, called the hash-value, or simply the hash, of x. Therefore, the hash can be seen as a "digest" of x.

Hash-functions can be used to verify the integrity of a message x: since a modification in x will result in a different hash, any user can verify that the modified x does not map to $H(x)$ (SIMPLICIO JR, 2010). In this case, the integrity of $H(x)$, must be ensured somehow, or attackers could replace x by x'

at the same time that they replace $H(x)$ by $H(x')$, misleading the verification process.

We note that there is also a more generic definition for hash functions without the one-way requirement (MENEZES *et al.*, 1996), but for the purposes of this document such alternative is not considered because password hashing schemes, which are the focus of the discussion, requires the a H that is not easily invertible.

2.1.1 Security

Since hash-functions are "many-to-one" functions, the existence of collisions (pairs of inputs x and x' that are mapped to the same output $y = H(x)$) is unavoidable. Indeed, supposing that all input messages have a length of at most t bits, that the outputs are h-bit long and that all 2^h outputs are equiprobable, then 2^{t-h} inputs will map to each output, and two input picked at random will yield to the same output with a probability of 2^{-h}. To prevent attackers from using this property to their advantage, secure hash algorithms must satisfy at least the following three requirements:

- *First Pre-image Resistance*: given a hash $y = H(x)$, it is computationally infeasible to find any x having that hash-value, i.e., it is computationally infeasible to "invert" the hash-function

- *Second Pre-image Resistance*: given x and its corresponding hash $y = H(x)$, it is computationally infeasible to "find any other input" x' that maps to the same hash, i.e., it is computationally infeasible find a x' such that $x' \neq x$ and $y = H(x') = H(x)$.

- *Collision Resistance*: it is computationally infeasible to "find any two distinct inputs" x and x' that map to the same hash-value, i.e., it is computationally infeasible to find $H(x) = H(x')$ where $x \neq x'$.

2.2 Cryptographic Sponges

The concept of *cryptographic sponges* was formally introduced by Bertoni *et al.* in (BERTONI *et al.*, 2007) and is described in detail in (BERTONI *et al.*, 2011a). The elegant design of sponges has also motivated the creation of more general structures, such as the Parazoa family of functions (ANDREEVA; MENNINK; PRENEEL, 2011). Indeed, their flexibility is probably among the reasons that led Keccak (BERTONI *et al.*, 2011b), one of the members of the sponge family, to be elected as the new Secure Hash Algorithm (SHA-3).

2.2.1 Basic Structure

In a nutshell, different to the hash functions, 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

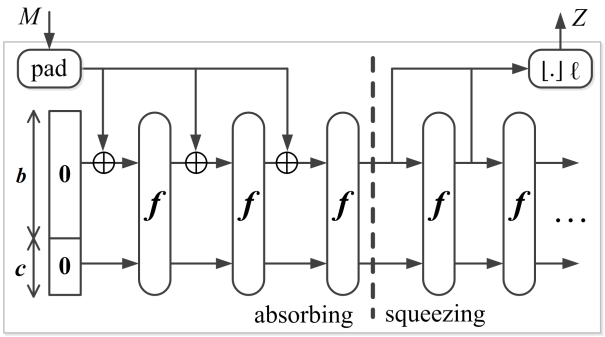


Figure 1: Overview of the sponge construction $Z = [f, \texttt{pad}, b](M, \ell)$. Adapted from (BERTONI *et al.*, 2011a).

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 (BERTONI *et al.*, 2011b)).

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

2.2.2 The duplex construction

A similar structure derived from the sponge concept is the *Duplex construction* (BERTONI *et al.*, 2011a), 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 x must be short enough to fit in a single b-bit block after padding, and the output length ℓ must satisfy $\ell \leqslant b$.

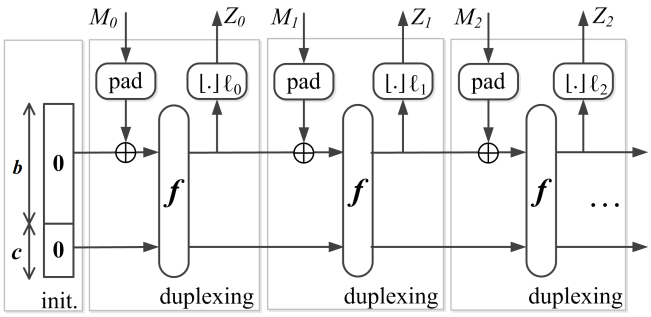


Figure 2: Overview of the duplex construction. Adapted from (BERTONI *et al.*, 2011a).

2.2.3 Security

The fundamental attacks against cryptographic sponge functions which allow it to be distinguished from a random oracle are called *primary attacks* (BERTONI *et al.*, 2011a). In order to be considered secure, a sponge function must resist to such attacks, which implies that the following operations must be computationally infeasible (BERTONI *et al.*, 2011a):

- Finding a path (a sequence of bytes to be absorbed by the sponge) P leading to a given internal state s.
- Finding two different paths leading to the same internal state s.
- Finding the internal state s for a given output Z.

2.3 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 (KELSEY *et al.*, 1998). 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 (NIST, 2011; WEIR *et al.*, 2009)).

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* (KALISKI, 2000; KELSEY *et al.*, 1998). 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 (KALISKI, 2000).

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.

2.3.1 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 (DÜRMUTH; GÜNEYSU; KASPER, 2012).

2.3.1.1 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 (NVIDIA, 2014) and OpenCL (KHRONOS GROUP, 2012)). As a result, they became more intensively employed for more general purposes, including password cracking (DÜRMUTH; GÜNEYSU; KASPER, 2012; SPRENGERS, 2011).

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 (NVIDIA, 2012). Its computational power can also be further expanded by using the host machine's resources (NVIDIA, 2014), 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.

2.3.1.2 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 (DANDASS, 2008; KAKAROUNTAS *et al.*, 2006)). 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 (MARECHAL, 2008).

When compared to GPUs, FPGAs may also be advantageous due to the latter's considerably lower energy consumption (CHUNG *et al.*, 2010; FOWERS

et al., 2012), which can be further reduced if its circuit is synthesized in the form of custom logic hardware (ASIC) (CHUNG *et al.*, 2010).

A recent example of password cracking using FPGAs is presented in (DÜRMUTH; GÜNEYSU; KASPER, 2012). Using a RIVYERA S3-5000 cluster (SCIENGINES, 2013a) 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) (DÜRMUTH; GÜNEYSU; KASPER, 2012, 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 (SCIENGINES, 2013a) 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 (SCIENGINES, 2013b). 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 (SCIENGINES, 2013b). 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 RELATED WORKS

Following the call for candidates made by the Password Hashing Competition (PHC), several new Password Hashing Schemes have emerged in the last years. To be more specific, 24 new schemes were proposed, two of which voluntarily gave up the competition (PHC, 2013); later, out of the 22 remaining proposals, only 9 were selected for the final phase of the PHC (PHC, 2015c). In what follows, we describe the main password hashing solutions available in the literature and also give a brief overview of the PHC's finalists that, like Lyra2, allow both memory usage and processing time to be configured. For conciseness, however, we do not cover all details of each PHC algorithm, but only the main characteristics that are useful for the discussion. Nevertheless, we refer the interested reader to the PHC official website (PHC, 2013) for details on each submission to the competition.

3.1 Pre-PHC Schemes

Arguably, the main password hashing solutions available in the literature before the start of PHC were (PHC, 2013): PBKDF2 (KALISKI, 2000), bcrypt (PROVOS; MAZIÈRES, 1999), scrypt (PERCIVAL, 2009) and Lyra2's predecessor (simply called Lyra) (ALMEIDA *et al.*, 2014). These schemes are described in what follows.

3.1.1 PBKDF2

The Password-Based Key Derivation Function version 2 (PBKDF2) algorithm (KALISKI, 2000) was originally proposed in 2000 as part of RSA Laboratories' Public-Key Cryptography Standards series, namely PKCS#5. It is nowadays present in several security tools, such as TrueCrypt (TRUECRYPT, 2012), Apple's iOS for encrypting user passwords (Apple, 2012) and Android operating system for filesystem encryption, since version 3.0 (AOSP, 2012), and has been formally analyzed in several circumstances (YAO; YIN, 2005; BELLARE; RISTENPART; TESSARO, 2012).

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

PBKDF2 allows users to control its total running time by configuring the T parameter. Since the password hahsing process is strictly sequential (one cannot

Algorithm 1 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

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.

3.1.2 Bcrypt

Another solution that allows users to configure the password hashing processing time is bcrypt (PROVOS; MAZIÈRES, 1999). The scheme is based on a customized version of the 64-bit cipher algorithm Blowfish (SCHNEIER, 1994), called *EksBlowflish* ("expensive key schedule blowfish").

Algorithm 2 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 $T \,\|\, salt \,\|\, ctext$
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

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 2). 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). For example, in the first iteration, the first 64 bits of the salt are encrypted, and then the result is XORed with its second half and replaces the first two subkeys; this new set of subkeys is used in the subsequent encryption.

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, i.e., does not allow the users the amount of memory used by the algorithm, 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. This latter issue is not a serious limitation, not only because larger passwords are unlikely to be "human-memorizable", but also because this could be overcome by pre-hashing the password to the required 56 bytes before the call to the bcrypt algorithm. Nonetheless, this does impairs the scheme's flexibility.

3.1.3 Scrypt

The design of scrypt (PERCIVAL, 2009) 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

Algorithm 3 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
        $M_i \leftarrow X$; $X \leftarrow \text{BLOCKMIX}(X)$
    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

operations and for which a parallel implementation cannot asymptotically obtain a significantly lower cost. Informally, this means that if the number of operations and the amount of memory used in the regular operation of the algorithm are both $\mathcal{O}(R)$, where R is a system parameter, then any attack trying to exploit time-memory trade-offs (TMTO) should always lead to a $\Omega(R^2)$ time-memory product, limiting the attacker's capability of using strategies for reducing the algorithm's total memory usage. For example, the complexity of a memory-free attack – i.e., an attack for which the memory usage is reduced to $\mathcal{O}(1)$ – becomes $\Omega(R^2)$, which should compel attackers to use more memory. For conciseness, we refer the reader to (PERCIVAL, 2009) for a more formal definition of the memory-hardness concept.

The following steps compose scrypt's operation (see Algorithm 3). First, it initializes p b-long memory blocks B_i. This is done using the PBKDF2 algorithm with HMAC-SHA-256 (NIST, 2002b) as underlying hash function and a single iteration. Then, each B_i is processed (incrementally or in parallel) by the sequential memory-hard *ROMix* function. Basically, *ROMix* initializes an array M of R b-long elements by iteratively hashing B_i. It then visits R positions of M at random, updating the internal state variable X during this (strictly sequential) process in order to ascertain that those positions are indeed available in memory.

The hash function employed by *ROMix* is called *BlockMix*, which emulates a function having arbitrary (b-long) input and output lengths; this is done using the Salsa20/8 (BERNSTEIN, 2008) 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 (AUMASSON *et al.*, 2008) are not expected to put the security of scrypt in hazard (PERCIVAL, 2009), 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. However, sponges' intrinsic properties make some of scrypt's operations unnecessary: for example, since sponges support inputs and outputs of any length, the whole *BlockMix* structure could be replaced.

3.1.4 Lyra

Inspired by scrypt's design, Lyra (ALMEIDA *et al.*, 2014) 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.

Lyra's steps as described in (ALMEIDA *et al.*, 2014) are detailed in Algorithm 4. Basically, Lyra builds upon (reduced-round) operations of a cryptographic sponge for (1) building a memory matrix, (2) visiting its rows in a pseudorandom fashion, as many times as defined by the user, and then (3) providing the desired number of bits as output. More precisely, the first part of the algorithm, called the *Setup Phase* (lines 1 – 8), 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). Without resetting the sponge's internal state, the algorithm enters then the *Wandering Phase* (lines 9 – 19), in which $(T \cdot R)$ rows are visited in a pseudorandom fashion, aiming to ensure that the whole memory matrix is still available in memory. Every row visited in this manner has all of its cells read and combined with the output of the underlying sponge's (reduced) duplexing operation $Hash.duplexing_{\rho}$ (line 14). Finally, in the *Wrap-up Phase* (lines 20 – 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 stateful, full-round sponge employed in this last stage ensures that the whole process is both non-invertible and of sequential nature.

While Lyra2 also builds upon reduced-round sponges for achieving high performance, it also addresses some shortcomings of Lyra's design. First, 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. Second, Lyra's duplexing operations performed during the Wandering phase involve only one pseudorandomly-picked row, which is read and

Algorithm 4 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
$Hash.absorb(\texttt{pad}(salt \,\|\, pwd))$ ▷ 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

written upon. As it turns out, one can add extra rows to this process with little impact on performance on modern platforms, as existing memory devices have enough bandwidth to support a higher number of memory reads/writes. Raising the amount of memory accesses also have positive results on security, for two reasons: (1) if an attacker tries to trade memory usage for extra processing, a potentially larger number of rows will have to be recomputed for performing each duplexing operation; and (2) attackers trying to run multiple instances of the password hashing algorithm (or recomputations in an attack exploiting time-memory trade-offs) will need to account for such increased bandwidth usage. Lyra2's design addresses both of these issues, besides introducing a few other improvements (e.g., resistance to side-channel attacks).

3.2 Schemes from PHC

The Password Hashing Competition (PHC) was created aiming to evaluate novel PHS designs and ideas in terms of security, performance and flexibility (PHC, 2013). In its first round, 22 candidates were submited to the competition: AntCrypt, Argon, battcrypt, Catena, Centrifuge, EARWORM, Gambit, Lanarea, Lyra2, Makwa, MCS_PHS, Omega Crypt, Parallel, PolyPassHash, POMELO, Pufferfish, RIG, Schvrch, Tortuga, TwoCats, Yarn and yescrypt. Beside these, two other schemes (Catfish and M3lcrypt) were submitted, but withdrawn before the initial evaluation process.

After the first round evaluation, 9 finalists were announced as potential winners of the competition, 6 of which are memory-hard algorithms (PHC, 2015c): Argon, battcrypt, Catena, POMELO, yescrypt and Lyra2. This selection was based on many criteria (PHC, 2015c): defense against GPU/FPGA/ASIC attackers; defense against time-memory trade-offs; defense against side-channel leaks; defense against cryptanalytic attacks; elegance and simplicity of design; quality of the documentation; quality of the reference implementation; general soundness and simplicity of the algorithm; and originality and innovation. These criteria have no particular order, and the panel also took into account some specific applications like web-service authentication, client login, key derivation, or usage in embedded devices, presented by some candidates.

In the specific case of Lyra2, the algorithm was announced as a finalist due to its elegant sponge-based design, with a single external primitive, and its detailed security analysis (PHC, 2015c). At the end of the competition, this property, together with the adopted approach to side-channel resistance that also takes into account slow-memory attacks, led to a "special recognition" for Lyra2 (PHC, 2015b). In this occasion, Argon2 (which was not among the original candidates,

but was accepted in the second round as a "new candidate" nevertheless) was announced as the PHC winner, and other three candidates received a special recognition; namely: Catena, for its agile framework approach and side-channel resistance; Makwa, for its unique delegation feature and its factoring-based security; and yescrypt, for its rich feature set and easy upgrade path from scrypt (PHC, 2015b).

In what follows, we briefly describe the other memory-hard finalists. For conciseness, however we do not analyze every detail of the algorithms, but only the main aspects that allow the reader to grasp the reason behind the numbers obtained in the comparative performance assessment presented later in Section 6.1.

3.2.1 Argon2

The Argon2 scheme was announced as an evolution of its predecessor, Argon. Although these algorithms have nothing in common, the PHC panel decided to accept Argon2 in the last phase of the competition after internal discussions and consultation to the other teams participating in the PHC, including the Lyra2 team (SIMPLICIO JR, 2015).

Contrasting with all other finalists, Argon2 displays more than one mode of operation which means that it works in a distinct way depending on its parameters. Namely, Argon2d's memory access pattern depends on the user's password, while Argon2i adopts a password-independent memory access (BIRYUKOV; DINU; KHOVRATOVICH, 2016); as a result, Argon2i displays high resistance against side-channel attacks, while Argon2d focus on resistance against slow-memory attacks. Besides these two operation modes, the authors also provide a hybrid operation in its official repository, called Argon2id (PHC, 2015a), which was designed after the end of the PHC as a recommended tweak for the algorithm. This mode combines a password-independent memory access pattern

when it fills the memory at the beginning of the algorithm's execution, and then revisits the memory in a password-dependent manner in the remainder of the process, which is actually very similarly to what is done in Lyra2. Although the multiple modes of operation may actually be confusing to users, since even specialists not always agree on how much side-channel vs. slow-memory resistance is necessary in a given practical scenario, this approach leads to a quite flexible design.

Another characteristic of the Argon2 scheme that shows that Lyra2 contributed to its final design is that it adopts as underlying cryptographic function the BlaMka multiplication-hardened sponge (BIRYUKOV; DINU; KHOVRATOVICH, 2016, Appendix A), which is actually one of the original results of this thesis, presented in Section 4.4.1

Like and Lyra2 (and also Lyra), Argon2 was designed to thwart attacks involving Time-Memory trade-offs (TMTO), imposing large processing penalties to attackers who try to run the algorithm with less memory than a legitimate user. According to the security analysis presented by Argon2's authors, for a memory reduction of approximately 1/6, the penalty should be approximately $(279601 \cdot T \cdot R)/6 \approx 2^{15.5} \cdot T \cdot R$ for Argon2i and $(4753217 \cdot T \cdot R)/6 \approx 2^{19.6} \cdot T \cdot R$ for Argon2d – where T is the algorithm's time parameter and R is its memory parameter – (BIRYUKOV; DINU; KHOVRATOVICH, 2016).

Argon2 can also be configured to use any amount of memory (e.g., it does not impose that the memory sizes must be a power of two, a limitation that appears in some other candidates). Nonetheless, the memory parameter should be larger than $8p$ and multiple of $4p$, where p is its degree of parallelism.

3.2.2 battcrypt

Battcrypt (Blowfish All The Things [crypt]) is a simplified scrypt and targets server-side application (THOMAS, 2014). Internally, it uses the Blowfish block cipher (SCHNEIER, 1994) in the CBC mode of operation (NIST, 2001b), as well as the hash function SHA-512 (NIST, 2002a). According to Battcrypt's authors, Blowfish is used because it is well-studied and included in PHP implementations (THOMAS, 2014).

Despite its simple design, Battcrypt does not have an in-depth security analysis. Namely, Battcrypt's authors provide only a discussion on the scheme's security in terms of the underlying Blowfish primitive, concluding that, if the latter is broken, the same applies to battcrypt too. There is, however, no security analysis concerning its resistance to TMTO attacks, besides the complete absence of mechanisms for protecting the algorithm against side-channel attacks.

Another shortcoming of Battcrypt is that its memory usage cannot be easily fine-tuned, as its running time depends on the time parameter T using a quite convoluted equation, namely $2^{\lfloor T/2 \rfloor \cdot ((T \mod 2)+2)}$ (THOMAS, 2014).

3.2.3 Catena

Catena was designed with an specific goal in mind: provide a memory-hard PHS with high resistance against side-channel attacks. To accomplish this goal, the algorithm avoids any password-dependent code branching, meaning that the order in which its internal memory is initialized and visited is completely deterministic, instead of pseudo-random. Specifically, this protects Catena against the so-called cache-timing attacks, in which the access to cache memory is monitored and used in the recovery of secret information (FORLER; LUCKS; WENZEL, 2013; BERNSTEIN, 2005).

On the positive side, Catena displays a quite simple and elegant design, which makes it easy to understand and implement. Its authors also provide a quite complete security analysis, relying on the fact that Catena's structure is based on a special type of graph called "*Bit-Reversal Graph*" (LENGAUER; TARJAN, 1982). This particular graph type allows the security of Catena to be formally demonstrated (at least in part) using the pebble-game theory (COOK, 1973; DWORK; NAOR; WEE, 2005), allowing time-memory trade-offs (TMTO) against the algorithm to be tightly calculated. Specifically, according to Catena's analysis in (FORLER; LUCKS; WENZEL, 2013; FORLER; LUCKS; WENZEL, 2014)[1]: the complexity of a memory-free attack against Catena is conjectured to be $\Theta(R^{T+1})$; in attacks where the attacker choose the amount of memory to be used, the complexity is conjectured as $\Theta(R^{T+1}/M^T)$, where R denotes the total memory used by legitimate users and M the memory used by the attacker.

On the negative side, Catena does not allow a flexible choice of parameters, as the memory size must always be a power of two. In addition, the algorithm as originally presented to the PHC was quite slow, although in its last version it also allows the usage of a reduced-round sponge as proposed in Lyra (a feature also incorporated in Lyra2). Finally, and as further discussed in Section 5.3, providing full resistance against cache-timing attacks facilitates other types of attacks that can benefit from a purely deterministic memory visitation pattern. Therefore, by focusing on one (not necessarily most probable) attack venue, Catena ends up failing to provide a compromise between the different attack strategies at the disposal of password crackers.

Despite these shortcomings, Catena has the merit of bringing several interesting ideas concerning the usage of a PHS. Probably the most useful one is the concept of "server relief protocol", which allows a remote authentication server

[1]Note: to standardize the notation hereby employed, here we interpret Catena's *garlic* as R and its *depth* as T

to offload (most of) the computational effort involved in the password hashing process to the client (FORLER; LUCKS; WENZEL, 2014), leading to a more scalable authentication system. Another interesting idea is the concept of client-independent update, a feature that allows the defender to increase the PHS's security parameters at any time, even for inactive accounts (FORLER; LUCKS; WENZEL, 2014), by re-hashing values already stored at the server's database. We note, however, that while these features are very relevant and described in detail for Catena, they can also be easily incorporated into any PHS (FORLER; LUCKS; WENZEL, 2013).

3.2.4 POMELO

POMELO has a quite simple design and, thus, is easy to implement (WU, 2015). Interestingly, this PHS does not adopt any existing cryptographic function as underlying primitive, but operates on 8-byte words and uses three state update functions developed specifically for this scheme. The first function is a simple non-linear feedback function and the other two provide simple random memory access over data (HATZIVASILIS; PAPAEFSTATHIOU; MANIFAVAS, 2015).

According to POMELO's author, these tree functions protect POMELO against pre-image attacks (i.e., attempts to invert the hash for obtaining the password) and also TMTO attempts (WU, 2015). In addition, and similarly to what is done in Lyra2, POMELO's design is such that it provides a compromise between resistance against cache-timing attacks and to other attacks that might take advantage of purely deterministic memory visitation patterns. We note, however, that even though the scheme's authors provide a quite complete security analysis in its specification manual, the underlying POMELO functions have been target of criticism for not having a formal proof of their security (PHC, 2015d).

Besides these potential doubts on the security of its primitives, POMELO also has one unusual security claim: protection against what the authors call "SIMD attacks", i.e., attacks that take advantage of SIMD instructions in modern platforms. Since most computer platforms do have support to SIMD, this is rather a limitation for *legitimate users*, who cannot take full advantage of available commodity hardware, than for attackers, who can build their own hardware platforms with whichever optimization they might find. In addition, while the authors claim that POMELO is resistant to GPU and dedicated hardware attacks (WU, 2015). Finally, POMELO's design makes it difficult to fine tune its memory usage and processing time, as both grow exponentially with its T and R parameters.

3.2.5 yescrypt

The yescrypt scheme is strongly based on scrypt and, as the latter, allows legitimate users to adjust its memory and time costs to desired levels, using the R and T parameters. However, and unlike most PHC candidates, it provides a wide range of parameters besides R and T, including the ability to indicate: whether it should be only read from memory after initializing it; if it should read from an additional, read-only memory device in addition to its internal memory; if it should operate in a scrypt-compatible mode; among many others. Furthermore, yescrypt uses many cryptographic primitives in its design, namely: SHA-256 (NIST, 2002a), HMAC (NIST, 2002b), Salsa20/8 (BERNSTEIN, 2008), and PBKDF2 (KALISKI, 2000) itself This adds a lot of complexity to yescrypt's design, making it very hard to understand and, thus, analyze.

Indeed, its authors do not provide an in-depth security analysis about the algorithm, but rather high level claims about how its design strategies should thwart TMTO attempts as well as attacks using GPUs and dedicated hardware.

The algorithm also has no mechanisms from protecting against side channel attacks, as all memory visitations are password-dependent, and does not allow the memory usage to be fine tunned, as the scheme requires its R parameter to be a power of two (PESLYAK, 2015).

Despite its complexity, probably one of the main contributions of yescrypt's design is the introduction of the "multiplication-hardening" concept (COX, 2014; PESLYAK, 2015), which translates to the adoption of integer multiplications among the PHS's internal operations as a way to protect against attacks based on dedicated hardware. The reason is that, as verified in several benchmarks available in the literature (SHAND; BERTIN; VUILLEMIN, 1990; SODERQUIST; LEESER, 1995), the performance gain offered by hardware implementations of the multiplication operation is not much higher than what is obtained with software implementations running on commodity 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. With these observations in mind, yescrypt iteratively performs several multiplications for each memory position it visits, and then mixes the result using one round of Salsa20/8.

4 LYRA2

As any PHS, Lyra2 takes as input a salt and a password, creating a pseudo-random output that can then be used as key material for cryptographic algorithms or as an authentication string (NIST, 2009). 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 by the user, allowing Lyra2's execution time to be fine-tuned according to the target platform's resources.

In this chapter, we describe the core of the Lyra2 algorithm in detail and discuss its design rationale and resulting properties. Later, in Appendix B, we discuss a possible variant of the algorithm that may be useful in a different scenario.

Algorithm 5 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)
$params \leftarrow len(k) \,\|\, len(pwd) \,\|\, len(salt) \,\|\, T \,\|\, R \,\|\, C$
$H.absorb(\texttt{pad}(pwd \,\|\, salt \,\|\, params))$ ▷ 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**
        $rand \leftarrow H_\rho.duplex(M[row^1][col] \boxplus M[prev^0][col] \boxplus M[prev^1][col], b)$
        $M[row^0][C-1-col] \leftarrow M[prev^0][col] \oplus rand$
        $M[row^1][col] \leftarrow M[row^1][col] \oplus \mathrm{rot}(rand)$ ▷ 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**
    $row^0 \leftarrow \mathrm{lsw}(rand) \bmod R$; $row^1 \leftarrow \mathrm{lsw}(\mathrm{rot}(rand)) \bmod R$ ▷ Picks pseudorandom rows
    for $(col \leftarrow 0$ **to** $C-1)$ **do** ▷ **Columns Loop**: updates $M[row^{0,1}]$
        ▷ Picks pseudorandom columns
        $col^0 \leftarrow \mathrm{lsw}(\mathrm{rot}^2(rand)) \bmod C$; $col^1 \leftarrow \mathrm{lsw}(\mathrm{rot}^3(rand)) \bmod C$
        $rand \leftarrow H_\rho.duplex(M[row^0][col] \boxplus M[row^1][col] \boxplus M[prev^0][col^0] \boxplus M[prev^1][col^1], b)$
        $M[row^0][col] \leftarrow M[row^0][col] \oplus rand$ ▷ Updates first pseudorandom row
        $M[row^1][col] \leftarrow M[row^1][col] \oplus \mathrm{rot}(rand)$ ▷ 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

4.1 Structure and rationale

Lyra2's steps are shown in Algorithm 5. As highlighted in the pseudocode's comments, its operation is composed by four sequential phases: Bootstrapping, Setup, Wandering and Wrap-up. Along the description, we assume that all operations are done using little-endian convention; they should, thus, be adapted accordingly for big-endian architectures (this applies basically to the rot operation).

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 `pad10*1` described in (BERTONI *et al.*, 2011a), 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 $pwd = 0^u$, $salt = 0^v$ 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 (DAEMEN; RIJMEN, 2005; DAEMEN; RIJMEN, 2010; SIMPLICIO JR *et al.*, 2009; SIMPLICIO JR; BARRETO, 2012). 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 $Lyra2$ 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 for saving memory and then recomputed right before they are used. 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

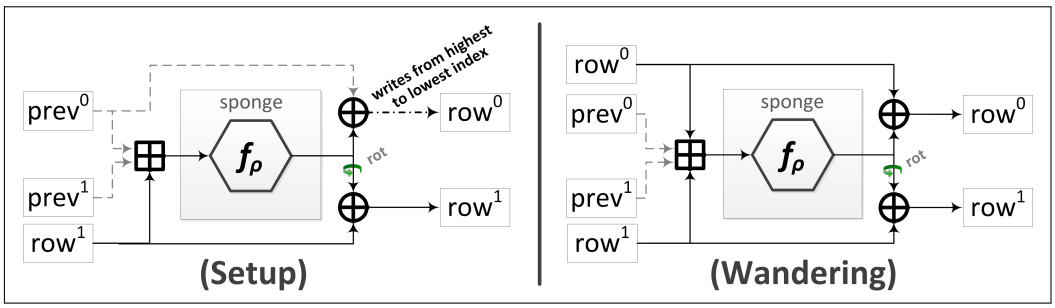


Figure 3: Handling the sponge's inputs and outputs during the Setup (left) and Wandering (right) phases in Lyra2.

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 1):

- 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 5, using the auxiliary $sqrt = \sqrt{wnd}$ variable to facilitate this computation.

- 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 5, once again using the auxiliary $sqrt = 2\sqrt{wnd/2}$ variable.

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

Table 1 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 5.1.2.4, 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 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 (BERTONI *et al.*, 2011b), does not

seem to be the best alternative for this purpose. This happens because Keccak excels in hardware rather than in software performance (GAJ *et al.*, 2012). 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 (AUMASSON *et al.*, 2013), a slightly tweaked version of Blake (AUMASSON *et al.*, 2010b). Blake itself displays a security level similar to that of Keccak (CHANG *et al.*, 2012), and its compression function has been shown to be a good permutation (AUMASSON *et al.*, 2010a; MING; QIANG; ZENG, 2010) and to have a strong diffusion capability (AUMASSON *et al.*, 2010b) even with a reduced number of rounds (JI; LIANGYU, 2009; SU *et al.*, 2010), while Blake2b is believed to retain most of these security properties (GUO *et al.*, 2014).

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 (AUMASSON *et al.*, 2013), 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 5) should have at least 512 bits, leaving up to 512 bits for the bitrate b. After this first absorb, 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, following the "multiplication-hardening" concept (COX, 2014; PESLYAK, 2015) briefly mentioned in Section 3.2.5 when discussing the yescrypt PHC candidate. More precisely, this approach is of interest whenever a legitimate user prefers to rely on a function that provides further protection against dedicated hardware platforms, while maintaining a high efficiency on platforms such as CPUs.

For this purpose the Blake2b structure may itself be adapted to integrate multiplications, which is done in the hereby proposed BlaMka algorithm. More precisely, Blake2b's G function (see the left side of Figure 4) relies on sequential additions, rotations and XORs (ARX) for attaining bit diffusion and creating a mutual dependence between those bits (AUMASSON *et al.*, 2010a; MING; QIANG; ZENG, 2010). If, however, the additions employed are replaced by another permutation that includes multiplications and still provides at least the same capacity of diffusion, its security should not be negatively affected.

$$
\begin{aligned}
a &\leftarrow a + b \\
d &\leftarrow (d \oplus a) \ggg 32 \\
c &\leftarrow c + d \\
b &\leftarrow (b \oplus c) \ggg 24 \\
a &\leftarrow a + b \\
d &\leftarrow (d \oplus a) \ggg 16 \\
c &\leftarrow c + d \\
b &\leftarrow (b \oplus c) \ggg 63
\end{aligned}
$$

(a) Blake2b's G function, based on the addition operation.

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

(b) BlaMka's $G_{\texttt{tls}}$ function, based on a truncated latin square (`tls`).

Figure 4: BlaMka's multiplication-hardened (right) and Blake2b's original (left) permutations.

One suggestion, originally made by Samuel Neves (one of the authors of Blake2) (NEVES, 2014), was to replace the additions of integers x and y by something like the latin square function (WALLIS; GEORGE, 2011) $\texttt{ls}(x, y) = x+y+2\cdot x\cdot y$. This would lead to an structure quite similar to what is done in the NORX authenticated encryption scheme (AUMASSON; JOVANOVIC; NEVES, 2014), but in the additive field. 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, which can be seen as a truncated version of the latin square. Some alternatives (one of which is described in Appendix C of this document) have been evaluated, but the end result is $\texttt{tls}(x, y) = x+y+2\cdot\text{lsw}(x)\cdot\text{lsw}(y)$, as shown in the right side of Figure 4 for the $G_{\texttt{tls}}$ function. This `tls` operation 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 (i.e., it still is a permutation).

The resulting structure, whose security and performance are further analyzed later in Sections 5.5 and 6.2, is indeed quite promising for usage in password hashing schemes Specifically, it provides better performance on hardware than on software platforms than Blake2 itself, justifying its early adoption as the default underlying sponge of Argon2's official implementation (PHC, 2015a) and also of Lyra2.

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 3) 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 (PERCIVAL, 2009), 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 5). 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 (HALDERMAN *et al.*, 2009; YUILL; DENNING; FEER, 2006).

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 5 using multiple instances of the cryptographic sponge H, so they cannot significantly speed up the process of testing a password by means of multiple processing cores; (3) once initialized, the memory matrix is expected to remain available during most of the password hashing process, meaning that the optimal operation of Lyra2 requires enough (fast) memory to hold its contents.

For better performance, a legitimate user is likely to store the whole memory matrix in volatile memory, facilitating its access in each of the several iterations of the algorithm. An attacker running multiple instances of Lyra2, on the other hand, may decide not to do the same, but to keep a smaller part of the matrix in fast memory aiming to reduce the memory costs per password guess. Even though this alternative approach inevitably lowers the throughput of each individual instance of Lyra2, the goal with this strategy is to allow more guesses to be independently tested in parallel, thus potentially raising the overall throughput of the process.

There are basically two methods for accomplishing this. The first is what we call a *Low-Memory attack*, which consists of trading memory for processing time,

i.e., discarding (parts of) the matrix and recomputing the discarded information from scratch, when (and only when) it becomes necessary. The second it to use low-cost (and, thus, slower) storage devices, such as magnetic hard disks, which we call a *Slow-Memory attack*.

In what follows, we discuss both attack venues and evaluate their relative costs, as well as the drawbacks of such alternative approaches. Our goal with this discussion is to demonstrate how Lyra2's design discourages attackers from making such memory-processing trade-offs while testing many passwords in parallel. Consequently, the algorithm limits the attackers' ability to take advantage of highly parallel platforms, such as GPUs and FPGAs, for password cracking.

In addition the above attacks, the security analysis hereby presented also discusses the so-called *Cache-Timing attacks* (FORLER; LUCKS; WENZEL, 2013), 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. It also evaluates Lyra2 in terms of *Garbage-Collector attacks* (FORLER *et al.*, 2014), an attack that explores vulnerabilities of the memory management during the derivation process. Finally, we present a preliminary analysis of the BlaMka sponge-based hash function proposed in Section 4.4.1.

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 3). 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 3), 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 3), 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 (COOK, 1973; DWORK; NAOR; WEE, 2005) as done in (FORLER; LUCKS; WENZEL, 2013; DZIEMBOWSKI; KAZANA; WICHS, 2011) — 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 5) 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 5. 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 5). 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 $\tau \geqslant 1$.

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

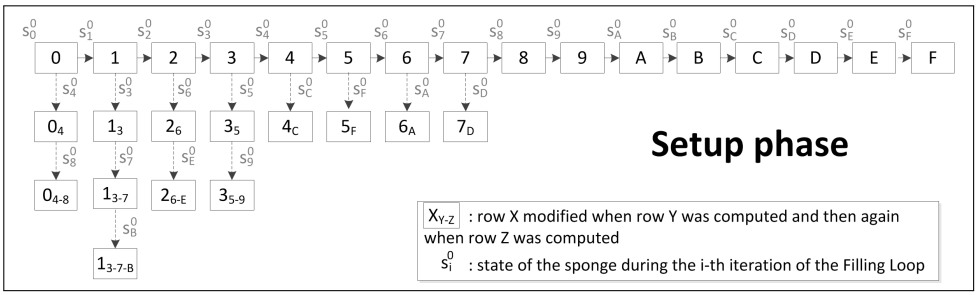


Figure 5: The Setup phase.

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.

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

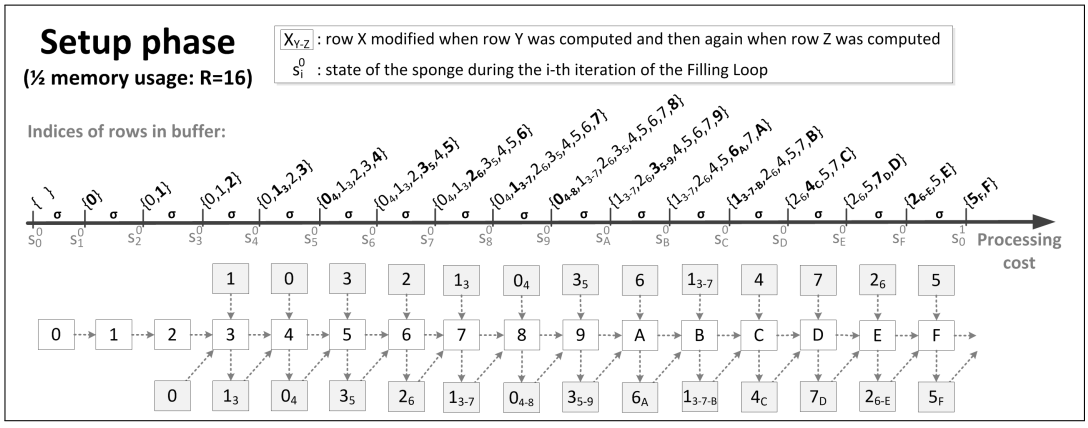


Figure 6: Attacking the Setup phase: storing 1/2 of all rows. The most recently modified rows in each iteration are marked in bold.

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. □

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,

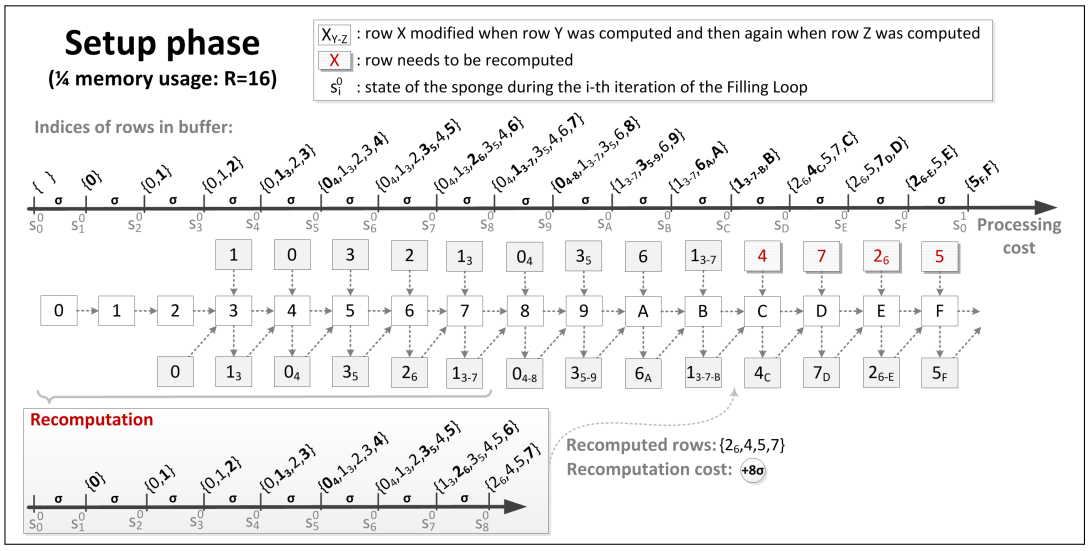


Figure 7: Attacking the Setup phase: storing 1/4 of all rows. The most recently modified rows in each iteration are marked in bold.

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

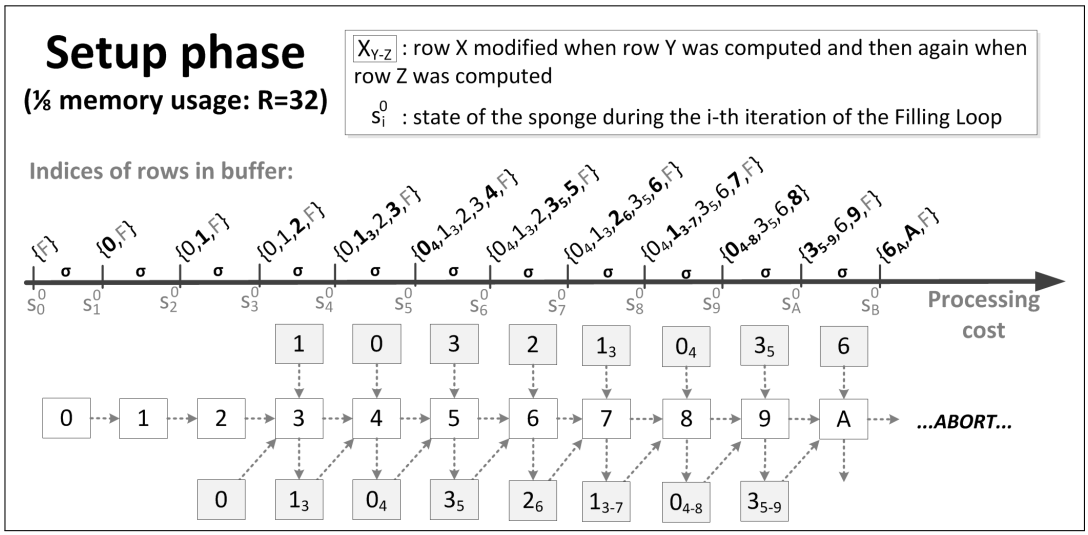


Figure 8: Attacking the Setup phase: recomputing $M[6_A]$ while storing $1/8$ of all rows and keeping $M[F]$ in memory. The most recently modified rows in each iteration are marked in bold.

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. 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:

$$\begin{aligned} CostSetup_0(m) &= 3m/2 \quad \triangleright\ 1/4 \text{ memory usage scenario } (n = 0) \\ CostSetup_n(m) &= \overbrace{CostSetup_{n-1}(m/2)}^{V_1^0} + \overbrace{m/2}^{V_2^0} + \\ &\quad \overbrace{\underbrace{(3 \cdot CostSetup_{n-1}(m/2)/4)}_{\text{approximate cost of each execution}} \cdot \underbrace{(2^{n+1})}_{\text{number of executions}}}^{\text{Re-executions of } V_1^0} \end{aligned} \tag{5.1}$$

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 5.1, 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 5.1 assumes that only 2^{n+1} re-executions of V_1^0 are required, on the other hand, is likely to become an oversimplification as R and n grow. The reason is that factor 4 discussed in section 5.1.2.3 is unlikely to compensate factor 3 in these cases. After all, as the memory available drops, it should become harder for the attacker to spare some space for rows that are not immediately needed.

The theoretical upper limit for the number of times V_1^0 would have to be executed during V_2^0 when the memory usage is m would then be $m/4$: this corresponds to a hypothetical scenario in which, unless promptly consumed, no row required by V_2^0 remains in the buffer during V_1^0; then, since V_2^0 revisits rows from V_1^0 in an alternating pattern, approximately a pair of rows can be recovered with each execution of V_1^0, as the next row required is likely to have already been computed and discarded in that same execution.

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

$$\begin{aligned} CostSetup_0(m) &= 3m/2 \qquad \triangleright\ 1/4 \text{ memory usage scenario } (n=0) \\ CostSetup_n(m) &= \overbrace{CostSetup_{n-1}(m/2)}^{V_1^0} + \overbrace{m/2}^{V_2^0} + \\ &\quad \overbrace{\underbrace{(3 \cdot CostSetup_{n-1}(m/2)/4)}_{\substack{\text{approximate cost of} \\ \text{each execution}}} \cdot \underbrace{(m/4)}_{\substack{\text{number of} \\ \text{executions}}}}^{\text{Re-executions of } V_1^0} \end{aligned} \tag{5.2}$$

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 5.2 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 5.1 (for a lower bound) and 5.2 (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 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

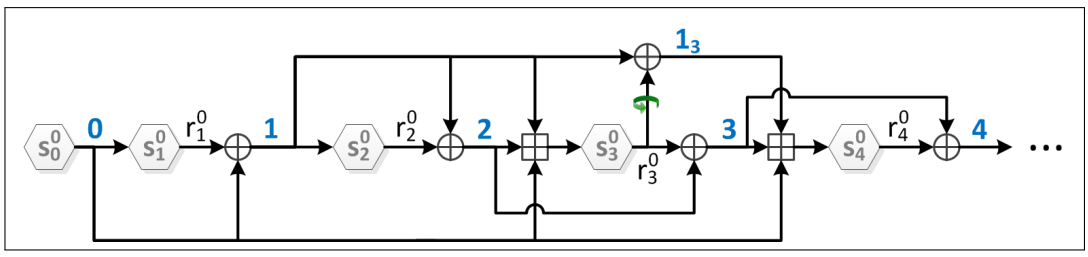


Figure 9: Attacking the Setup phase: storing only sponge states.

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 $M[2][C-2]$

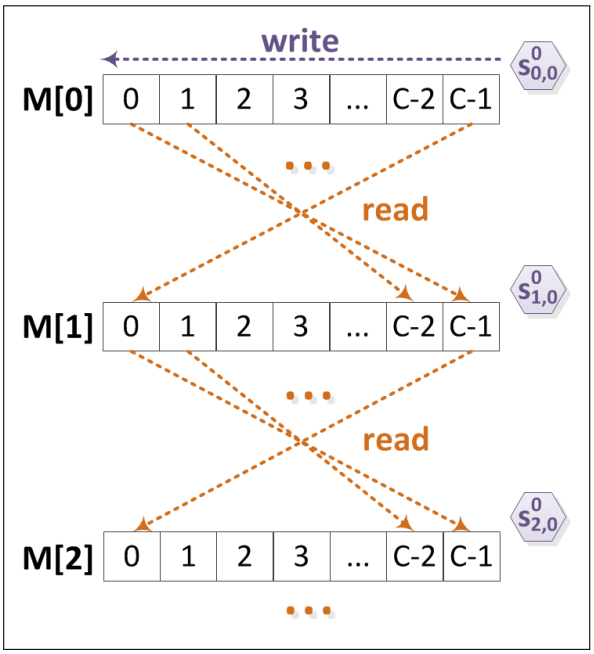


Figure 10: Reading and writing cells in the Setup phase.

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 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 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,

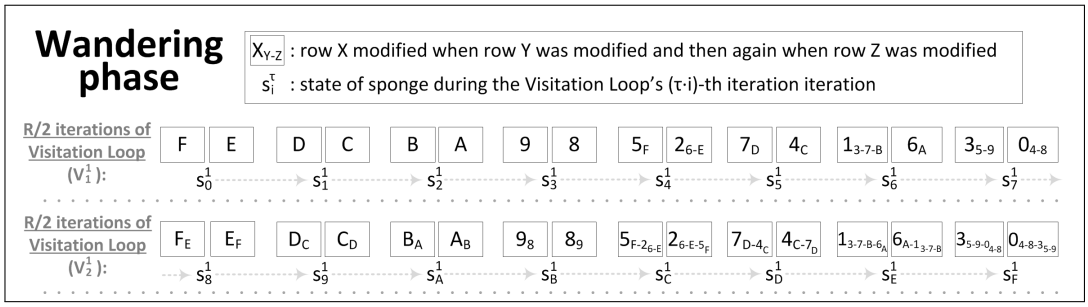


Figure 11: An example of the Wandering phase's execution.

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_2^0 are needed to compute each row from V_1^1. On the other hand, keeping $M[i \geqslant 3R/4]$ is unlikely to be much advantageous now, because that would reduce the attacker's ability to bufferize rows from V_1^1.

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

$$CostWander^*_\ell = \overbrace{\underbrace{(1/4)(R/2)}_{\substack{25\% \text{ of} \\ \text{iterations}}} \cdot 1}^{\substack{\text{no re-execution of} \\ \text{previous levels}}} + \overbrace{\underbrace{(3/4)(R/2)}_{\substack{75\% \text{ of} \\ \text{iterations}}} \cdot CostWander_{\ell-1}/2}^{\substack{\text{re-executions of} \\ \text{previous levels}}} \tag{5.3}$$

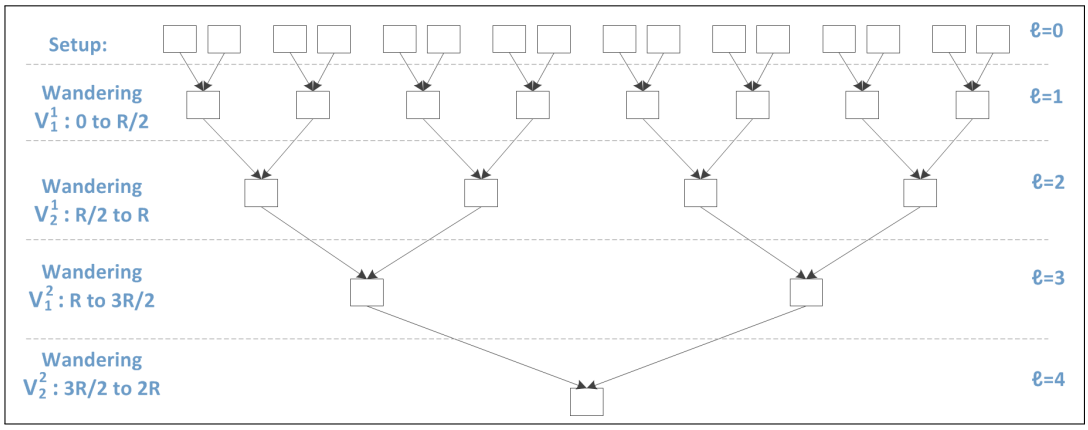


Figure 12: Tree representing the dependence among rows in Lyra2.

The value of $CostWander_{\ell-1}$ in Equation 5.3 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:

$$\begin{aligned} CostWander_0 &= R \quad \triangleright \text{ The Setup phase} \\ CostWander_\ell &= \underbrace{R/2}_{\substack{\text{internal} \\ \text{computations}}} + \underbrace{(3/2) \cdot CostWander_{\ell-1}}_{\substack{\text{re-executions of} \\ \text{previous level } (\ell-1)}} = R \cdot (2(3/2)^\ell - 1) \end{aligned} \tag{5.4}$$

Combining Equations 5.3 and 5.4 with Lemma 1, we get that the cost (in number of executions of CL) of running Lyra2 with half of the prescribed memory usage for a given T would be roughly:

$$\begin{aligned} CostLyra2_{(1/2)}(R,T) &= R + CostWander^*_1 + \cdots + CostWander^*_{2T} \\ &= (T+4) \cdot (R/4) + (3R^2/4) \cdot ((3/2)^{2T} - (T+2)/2) \\ &= \mathcal{O}((3/2)^{2T} R^2) \end{aligned} \tag{5.5}$$

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 5.3 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 5.1 and 5.2, 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 5.4: on average, $CostWander_\ell$ will involve $(1 + 2^{n+2})/2 \approx 2^{n+1}$ calls to $CostWander_{\ell-1}$.

Combining these observations, we arrive at

$$CostWander^*_{\ell,n} = \overbrace{\underbrace{(R/2) \cdot (1/2^{n+2})}_{1/2^{n+2} \text{ of iterations}} \cdot 1}^{\text{no re-execution of previous levels}} + \overbrace{\underbrace{(R/2) \cdot (1 - 1/2^{n+2})}_{\text{all other iterations}} \cdot (CostWander_{\ell-1,n})/2}^{\text{re-executions of previous levels}} \tag{5.6}$$

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

$$\begin{aligned} CostWander_{0,n} &= CostSetup_n(R) \quad \triangleright \text{ The Setup phase} \\ CostWander_{\ell,n} &= \overbrace{R/2}^{\text{internal computations}} + \overbrace{(2^{n+1}) CostWander_{\ell-1,n}}^{\text{re-executions of previous level}} \\ &= (R/2) \cdot (1 - (2^{n+1})^\ell)/(1 - 2^{n+1}) + (2^{n+1})^\ell \cdot CostSetup_n(R) \end{aligned} \tag{5.7}$$

for the deterministic re-executions of level ℓ.

Equations 5.6 and 5.7 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:

$$\begin{aligned} CostLyra2_{(1/2^{n+2})}(R,T) &= (CostSetup_n(R) + CostWander^*{}_{1,n} + \cdots + \\ &\quad CostWander^*{}_{2T,n})\sigma \\ &\approx \mathcal{O}((R^2)(2^{2nT}) + R \cdot CostSetup_n(R) \cdot 2^{2nT}) \end{aligned} \tag{5.8}$$

Since, as suggested in Section 5.1.2.4, the upper bound $CostSetup_n = \mathcal{O}(R^{n+1})$ given by Equation 5.2 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"

[1]This is analogous to the attack presented in (KHOVRATOVICH; BIRYUKOV; GROBSCHÄDL, 2014) for the version of Lyra2 originally submitted to the Password Hashing Competition as "V1"

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 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

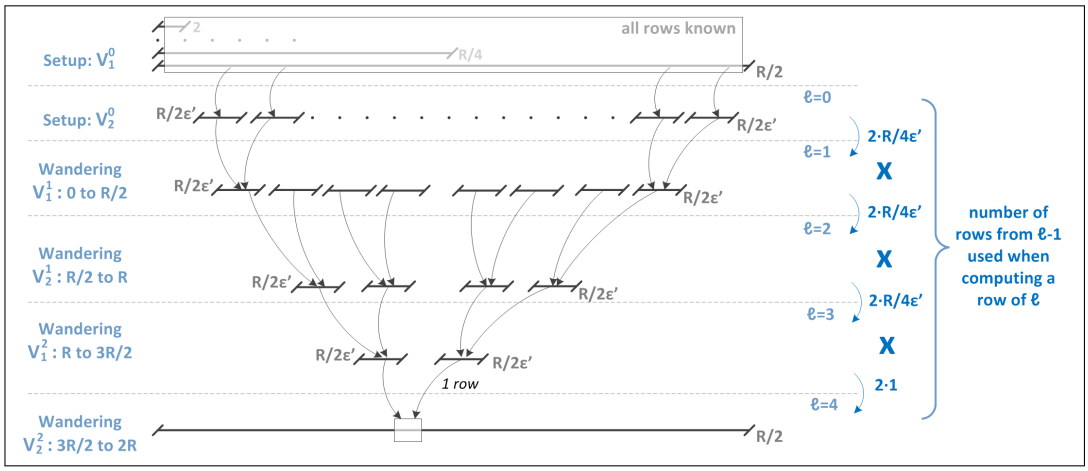


Figure 13: Tree representing the dependence among rows in Lyra2 with $T = 2$: using ϵ' sentinels per level.

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 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 6, 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 (NVIDIA, 2014, 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) (FORLER; LUCKS; WENZEL, 2013; BERNSTEIN, 2005). This class of attacks has been shown to be effective, for example, against certain implementations of the Advanced Encryption Standard (AES) (NIST, 2001a) and RSA (RIVEST; SHAMIR; ADLEMAN, 1978), allowing the recovery of the secret key employed by the algorithms (BERNSTEIN, 2005; PERCIVAL, 2005).

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 (FORLER; LUCKS; WENZEL, 2013). 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 (RISTENPART *et al.*, 2009). This happens because such environments potentially create the required conditions for making cache-timing measurements (RISTENPART *et al.*, 2009), 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 (FORLER; LUCKS; WENZEL, 2013), according to which the server sends only the salt to the client (preferably using a secure channel), who responds with $x = \text{PHS}(pwd, salt)$; 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 (MOWERY; KEELVEEDHI; SHACHAM, 2012), 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 (RISTENPART *et al.*, 2009) 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.

5.4 Garbage-Collector attacks

A garbage-collector attack is another type of side-channel attack, in which the attacker has access to the internal memory of the target's machine *after* a legitimate password hashing process is finished (FORLER *et al.*, 2014). In this context, the goal of the attacker is to find a valid password candidate based on the collected data, instead of trying all the possibilities (brute-force attack), exploiting the machine's memory management mechanisms as applied to the PHS's internal state and/or password-dependent values written in memory (FORLER *et al.*, 2014).

Specifically in the context of password hashing, garbage-collector attacks are a threat especially against memory-hard solutions that do not overwrite its internal memory after filling it with password-dependent data. In this case, the memory parts that may have been written on disk (e.g., due to memory swapping) or are still in RAM leave a useful trace that can be used in the discovery of the original password: attackers can execute the password ahshing over candidate passwords until they obtain the same memory section resulting from the correct password; if the contents of those regions match, then the password is probably correct and the whole hashing process should proceed until the end; otherwise, the test can be aborted earlier.

Albeit interesting and potentially powerful, the uefulness of such attacks against memory-hard solutions that overwrite its internal memory is quite limited. After all, when the memory visitation and overwritting is password-dependent, as in the case of Lyra2, the attacker cannot determine with a reasonable degree

of certainty how many times a given memory region has been overwritten until a given point of the processing. Hence, even if some speficic memory region is available due to swapping, it is not easy to determine when the corresponding value was obtained unless the entirety of the algorithm is run, which prevents the early abortion of the test. Therefore, since Lyra2 overwrites the sponge state multiple times and rewrite the memory matrix's rows and columns again and again, garbage-collector attacks do not apply to the algorithm, a claim that is corroborated by third-party analysis (FORLER *et al.*, 2014).

5.5 Security analysis of BlaMka.

Whereas the previous analysis does not depend on the underlying sponge, it is important also to evaluate the security of BlaMka as opposed to Blake2b, in especial considering that the latter has been intensively analyzed during the SHA-3 competition. Fortunately, in terms of security, a preliminary analysis of the $\texttt{tls}(x, y) = x + y + 2 \cdot \text{lsw}(x) \cdot \text{lsw}(y)$ operation adopted in BlaMka's $G_{\texttt{tls}}$ permutation indicates that its diffusion capability is at least as high as that provided by the simple word-wise addition employed by Blake2b's original G function. This observation comes from the assessment of XOR-differentials over `tls`, defined in (AUMASSON; JOVANOVIC; NEVES, 2015) 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 XOR-differentials that hold for all 65536 pairs (x, y), both for `tls` 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). Hence, the most common XOR-differentials have the same probability for both `tls` and regular additions.

However, when we analyze the XOR-differentials with second highest probability, we observe that the addition operation displays 168 XOR-differentials that hold for 50% of all (x, y) pairs, while the `tls` operation hereby described has only 48 of such XOR-differentials. XOR-differentials with lower, but still high probabilities are also less frequent for `tls` than for an addition — 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 `tls`.

Therefore, although tests with a larger number of bits would be required to confirm this analysis, by adopting the multiplication-hardened `tls` operation as part of its underlying $G_{\texttt{tls}}$ permutation, BlaMka is expected to be at least as secure as Blake2b when it comes to its diffusion capability. Interestingly, given the similarity between $G_{\texttt{tls}}$ and NORX's own structure, it is quite possible that analyses of this latter scheme can also apply to the construction hereby described.

5.6 Summary

This chapter provided a security analysis of Lyra2, showing how it's design thwarts different attack strategies. Table 2 summarizes the discussion, providing a comparison with other relevant Password Hashing Schemes available in the literature and discussed in Section 3. We note that, among the results appearing in this table, only the data related to Lyra2 were actually analyzed and presented in this work, while the remainder of the data shown were col-

Algorithm	Time-Memory trade-offs TMTO	Slow Memory	Side Channel	Hardware and GPUs	Garbage Collector
Argon2i	for $R' = R/6$ $\approx 2^{15.5} \cdot T \cdot R$	—	✓	✓	✓
Argon2d	for $R' = R/6$ $\approx 2^{19.6} \cdot T \cdot R$	✓	✗	✓	✓
battcrypt	—	✓	✗	✓	✓
Catena	$\mathcal{O}(1)$ $\Theta(R^{T+1})$	—	✓	✓	✓
Lyra2 [ours]	for $R' = R/2^{n+2}$, where $n \geq 0$ $\mathcal{O}(2^{2nT}R^{2+n/2})$, for $n \gg 1$	✓	!	✓	✓
POMELO	—	✓	!	✓	✓
yescrypt	$\mathcal{O}(1)$ $\mathcal{O}(R^{T+1})$	✓	✗	✓	✓[2]
scrypt	$\mathcal{O}(1)$ $\mathcal{O}(R^2)$	✓	✗	!	✗

✓- Has protection; ✗- Has no protection; ! - Partial protection; — - Nothing declared.

Table 2: Security overview of the PHSs considered the state of art.

lected on the reference guides, manuals and articles describing and/or analyzing these solutions, including also third-party analysis. Hence, we refer the interested reader to the original sources for details on each of these algorithms: Argon2 (BIRYUKOV; DINU; KHOVRATOVICH, 2016), battcrypt (THOMAS, 2014), Catena(FORLER; LUCKS; WENZEL, 2013), Pomelo (WU, 2015), yescrypt (PESLYAK, 2015), scrypt (PERCIVAL, 2009), and garbage collector attacks (FORLER *et al.*, 2014).

[2]Provides protection when in its "read-write" mode (`YESCRYPT_RW`) (FORLER *et al.*, 2014).

6 PERFORMANCE FOR DIFFERENT SETTINGS

In our assessment of Lyra2's performance, we used an SSE-enabled implementation of Blake2b's compression function (AUMASSON *et al.*, 2013) as the underlying sponge's f function of Algorithm 5. 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 5) 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>.

6.1 Benchmarks for Lyra2

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

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 (PERCIVAL, 2009), 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 $T \approx 6$.

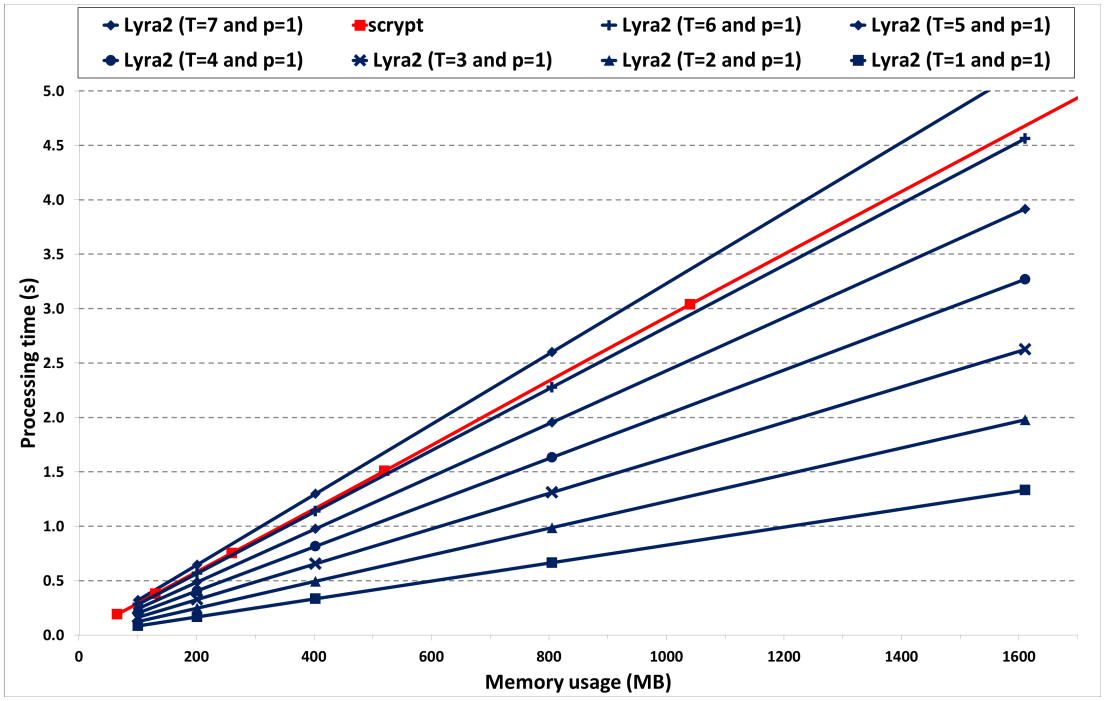


Figure 14: Performance of SSE-enabled Lyra2, for $C = 256$, $\rho = 1$, $p = 1$, and different T and R settings, compared with SSE-enabled scrypt.

We also performed tests aiming to compare the performance of Lyra2 and the other 5 memory-hard PHC finalists: Argon2, 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 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 (BROZ, 2014), are depicted in Figure 15, which shows that Lyra2 is a very competitive solution in terms of performance.

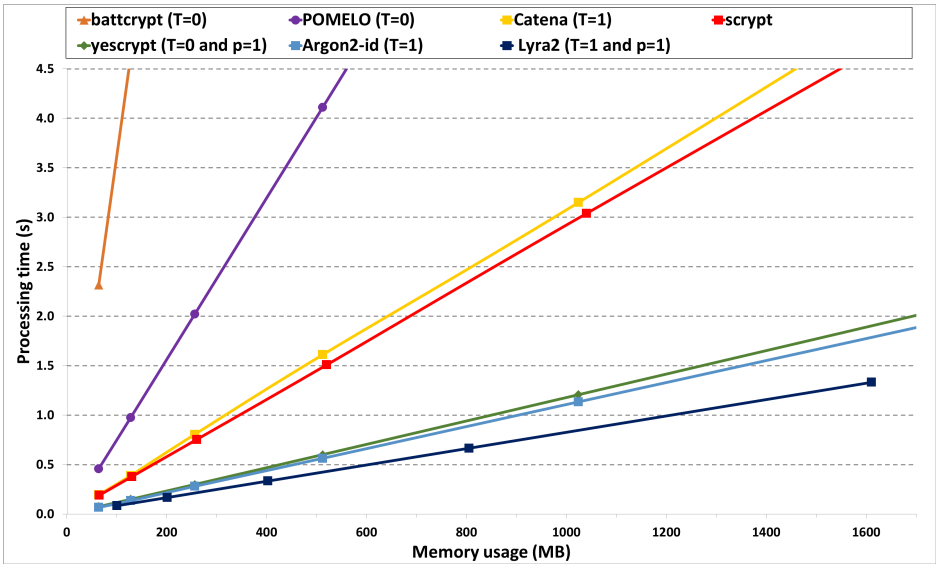


Figure 15: Performance of SSE-enabled Lyra2, for $C = 256$, $\rho = 1$, $p = 1$, and different T and R settings, compared with SSE-enabled scrypt and memory-hard PHC finalists with minimum parameters.

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

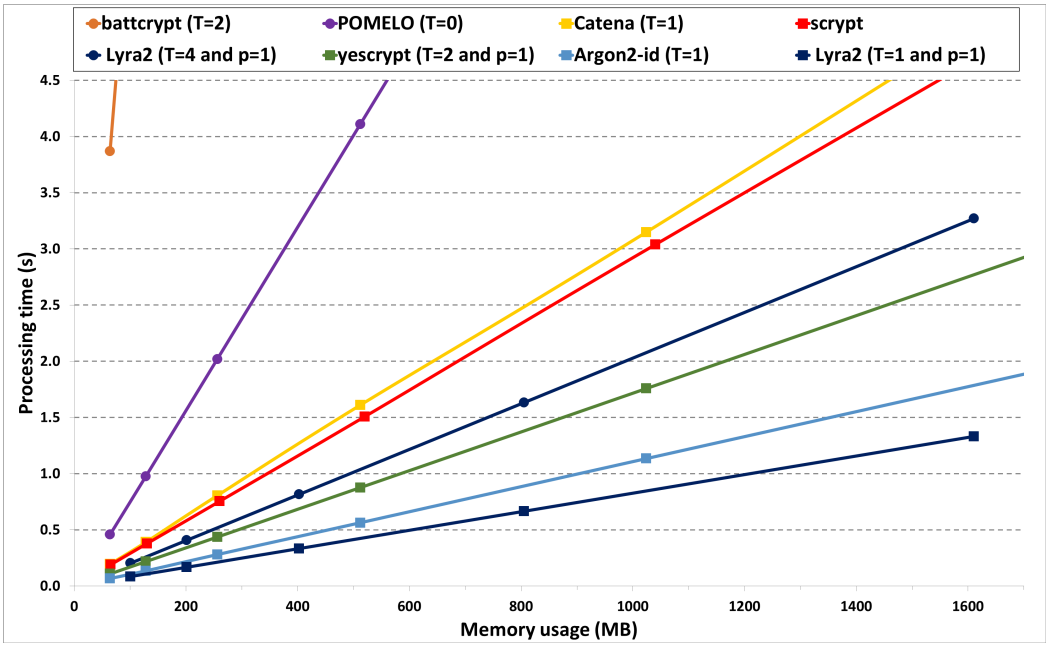


Figure 16: Performance of SSE-enabled Lyra2, for $C = 256$, $\rho = 1$, $p = 1$ and different T and R settings, compared with SSE-enabled scrypt and memory-hard PHC finalists with a similar number of calls to the underlying function (comparable configurations are marked with the same symbol, ■ or •).

Algorithm	Calls to underlying primitive	SIMD instructions
Argon2	$2 \cdot T \cdot M$	Yes
battcrypt	$\{2^{\lfloor T/2 \rfloor} \cdot [(T \mod 2) + 2] + 1\} \cdot M$	No
Catena[1]	$(T + 1) \cdot M$	Yes
Lyra2	$(T + 1) \cdot M$	Yes
POMELO	$\left(3 + 2^{2T}\right) \cdot M$	No
yescrypt	$T \cdot M$	Yes

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

comes from Table 3, which shows how each memory-hard PHC finalist handles the time- and memory-cost parameters (respectively, T and M), based on the analysis of the documentation provided by their authors (PHC, 2013; PESLYAK, 2015; PHC wiki, 2014). 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.

In addition of these results, a third-party implementation (GONÇALVES, 2014) has shown that it is possible to reduce Lyra2's processing time even more in modern processors equipped with AVX2 (Advanced Vector Extensions 2) instructions . With that study, the authors produced a piece of code that runs 30% faster than our SSE-optimized implementation (GONÇALVES; ARANHA, 2005).

[1]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 $(T + 1) \cdot M$.

6.2 Benchmarks for BlaMka.

In terms of performance, our analysis indicate that the throughput of BlaMka when compared with Blake2b drops less in CPUs than in dedicated hardware, meaning that it is more advantageous for regular users than it is for attackers. This analysis is based on benchmarks of Blake2b's G and BlaMka's $G_{\texttt{tls}}$ functions performed in CPUs, comparing the results with hardware implementations made by Jonatas F. Rossetti under the supervision of Prof. Wilson V. Ruggiero.

For all implementations, we measured the average throughput of one round of G and $G_{\texttt{tls}}$ (i.e., for 256 bits of input/output), which does not correspond to the entire Blake2b or BlaMka algorithms but should be enough for a comparative analysis between these permutation functions. In addition, for simplicity, along the discussion we ignore ancillary (e.g, memory-related) operations that should appear in any algorithm using either Blake2b or BlaMka as underlying hash function. As later discussed in Section 6.3, this simplification leads to numbers implying an impact much more significant than those actually observed in algorithms involving a large amount of memory-related operations, such as Lyra2 itself. Nevertheless, the resulting analysis is useful to evaluate how the adoption of BlaMka or Blake2b as underlying hash function impacts the "processing" portion of an algorithm's total execution time.

Table 4 shows the benchmark results obtained with software implementations of G and $G_{\texttt{tls}}$. All tests were performed on an Intel Xeon E5-2430 (2.20 GHz with 12 Cores, 64 bits) equipped with 48 GB of DRAM and running Ubuntu 14.04 LTS 64 bits. The source code was compiled using gcc 4.9.3 with `-O3` optimization, and SIMD (Stands for Single Instruction, Multiple Data) instructions were not used, in order to implement these functions in the most generic way possible.

Microarchitecture	CPU Family	Model	G	Implem.	T. Power (W)	Max. Frequency (MHz)	Throughput (Gbps)	Throughput Ratio
Sandy Bridge	Intel Xeon	E5-2430	Blake2b	Generic	95	2200[2]	51.578	1
			BlaMka	Generic			21.465	0.416

Table 4: Data related of the tests performed in CPU, executing just one round of G function (i.e., 256 bits of output).

As shown in Table 4's rightmost column, BlaMka's $G_{\texttt{tls}}$ function has approximately 42% of the throughput capacity of the original Blake2b's G permutation function in CPUs. Hence, ignoring ancillary operations, any algorithm using BlaMka as underlying hash function should observe a similar impact on its throughput, whether it is a legitimate or attacker's implementation.

The hardware descriptions for G and $G_{\texttt{tls}}$, on their turn, were made directly from definition of the algorithms described on Figure 4, using basic combinational and sequential components, such as registers, multiplexers, adders, multipliers, and 64-bit XORs. The rotation and shift left operations present on BlaMka were implemented as simple changes in wire connections, as it is known in advance how many bits should be rotated and left shifted. Two versions were developed for the purposed of benchmarking. In the first, denoted simply Generic", no special adder or multiplier were used, leaving for the implementation tools the task of choosing some special algorithm. In the second, denoted "Opt" to indicate further optimizations, the hardware descriptions were made using specific combinational multipliers — namely, Dadda Multipliers (DADDA, 1965; DADDA, 1976) — and Carry Save Adders, using registers along the critical path to reduce latency.

Both Generic and Opt descriptions were implemented in FPGA using Xilinx ISE Design Suite (XILINX, 2013) and Xilinx Vivado Design Suite (XILINX, 2016) for multiple FPGA families of different fabrication technologies (Tables 5 and 6). For the implementations on ASICs, the Synopsys Design Compiler tool

[2]During the tests the TurboBoost was turned off.

(SYNOPSYS, 2010) was used in a 90nm cell library (Table 6). These tools were used with default implementation parameters, aiming to achieve the most generic results possible.

Tables 5 and 6 show the results obtained for FPGA and ASICs implementations, respectively. In those tables, one can see that the throughput ratio of BlaMka instead of Blake2b is in most cases worst than the one obtained with CPUs. Namely, in the FPGA-based implementations, BlaMka's throughput from 29.5% to 58.9% of Blake2b's throughput instead of 42% observed in software, but in 9 out of 12 them BlaMka's multiplication-hardened permutation was higher on hardware than on software. The ASICs' implementations, on the other hand, could cope considerably better with the $G_{\texttt{tls}}$ structure, so BlaMka's throughput became 45.3% to 89.5% of Blake2b's in this platform.

Nevertheless, in practice we need to take account of the *area* employed by the implementations, since, from an attackers' perspective, doubling the area of one implementation only makes sense if the resulting throughput more than doubles; otherwise, it would be more beneficial to have two instances of the algorithm running in parallel testing different passwords, as this would result in twice as much area with twice the throughput. The rightmost column of Tables 5 and 6, the "Relative Throughput", take this observation into account, as they indicate how much throughput is lost with the adoption of BlaMka instead of Blake2b assuming the same amount of silicon area is available for implementing both algorithms. Specifically, BlaMka's relative throughput ranges from 4% to 38.3% of the throughput observed with Blake2b, remaining, thus, below the 42% obtained in software. Therefore, according to this analysis, the adoption of BlaMka as underlying sponge by legitimate users with software-based implementations of the algorithm does improve defense against attackers with access to dedicated hardware.

FPGA (ISE Design Suite)

Technology	FPGA Family	G	Implem.	Area		T. Power	Max. Frequency	Throughput		Relative
				Slices	Ratio	(mW)	(MHz)	Gbps	Ratio	Throughput
65nm	Virtex 5	Blake2b	Generic	237	1	1529	257.003	4.112	1	1
		BlaMka	Generic	383	1.616	1280	75.689	1.211	0.295	0.182
		BlaMka	Opt	2030	8.565	1511	88.511	1.416	0.344	0.040
28nm	Kintex 7	Blake2b	Generic	359	1	372	364.299	5.829	1	1
		BlaMka	Generic	433	1.206	232	109.397	1.750	0.300	0.249
		BlaMka	Opt	2306	6.423	463	142.450	2.279	0.391	0.061
45nm	Spartan 6	Blake2b	Generic	239	1	45	141.663	2.267	1	1
		BlaMka	Generic	206	0.862	94	46.757	0.748	0.330	0.383
		BlaMka	Opt	1592	6.661	327	104.373	1.336	0.589	0.088

FPGA (Vivado Design Suite)

Technology	FPGA Family	G	Implem.	Area		T. Power	Max. Frequency	Throughput		Relative
				Slices	Ratio	(mW)	(MHz)	Gbps	Ratio	Throughput
16nm	Kintex Ultra SCALE+	Blake2b	Generic	144	1	690	500.250	8.004	1	1
		BlaMka	Generic	210	1.458	692	167.842	2.685	0.335	0.230
		BlaMka	Opt	1216	8.444	766	194.818	3.117	0.389	0.046
20nm	Kintex Ultra SCALE	Blake2b	Generic	148	1	958	280.741	4.492	1	1
		BlaMka	Generic	218	1.473	956	99.433	1.591	0.354	0.240
		BlaMka	Opt	1184	8.000	1024	135.538	2.169	0.483	0.060
28nm	Virtex 7	Blake2b	Generic	139	1	273	219.443	3.511	1	1
		BlaMka	Generic	188	1.353	276	100.291	1.605	0.457	0.338
		BlaMka	Opt	994	7.151	317	84.545	1.353	0.385	0.054

Table 5: Data related of the initial tests performed in FPGA, executing just one round of G function (i.e., 256 bits of output).

FPGA (ISE Design Suite)

Technology	FPGA Family	G	Implem.	Area		T. Power	Max. Frequency	Throughput		Relative
				Slices	Ratio	(mW)	(MHz)	Gbps	Ratio	Throughput
90nm	Spartan 3E	Blake2b	Generic	508	1	162	98.097	1.570	1	1
		BlaMka	Generic	787	1.549	159	44.470	0.711	0.453	0.292
		BlaMka	Opt	3567	7.022	160	57.894	0.741	0.472	0.067

ASIC (Synopsys Design Compiler)

Technology	Family	G	Implem.	Area		T. Power	Max. Frequency	Throughput		Relative
				μm^2	Ratio	(μW)	(MHz)	Gbps	Ratio	Throughput
90nm	SAED EDK 90nm	Blake2b	Generic	40191	1	222	239.808	3.837	1	1
		BlaMka	Generic	107088	2.664	485	214.592	3.433	0.895	0.336
		BlaMka	Opt	142911	3.556	637	245.700	3.145	0.820	0.231

Table 6: Data related of the initial tests performed in dedicated hardware (that present advantage against CPU), executing just one round of G function (i.e., 256 bits of output).

6.3 Benchmarks for Lyra2 with BlaMka

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 reported in Section 6.1; nevertheless, the impact is unlikely to be as high as observed in Section 6.2, since memory-related operations play an important role in the algorithm's total execution time. To assess the impacts of BlaMka over Lyra2's efficiency, we conducted some benchmarks as described in what follows.

Figure 17 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 than yescrypt and Argon2 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 $T = 2$ and Lyra2 with $T = 1$).

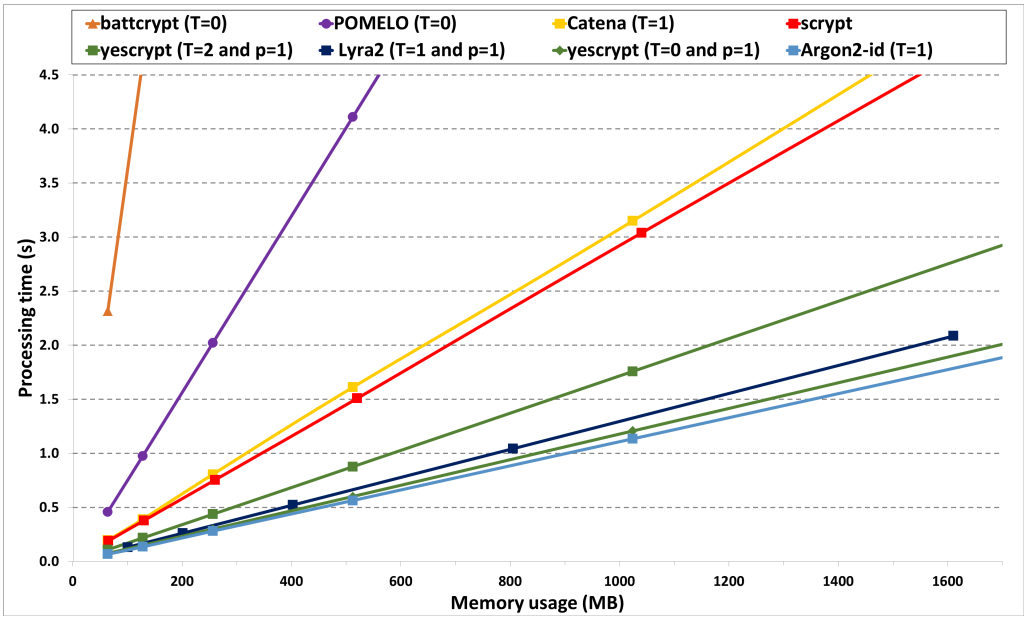


Figure 17: Performance of SSE-enabled Lyra2 with BlaMka G function, for $C = 256$, $\rho = 1$, $p = 1$, and different T and R settings, compared with SSE-enabled scrypt and memory-hard PHC finalists (configurations with a similar number of calls to the underlying function are marked with the same symbol, ■).

6.4 Expected attack costs

Considering that the cost of DDR3 SO-DIMM memory chips is currently around U$8.6/GB (TRENDFORCE, 2015), Table 7 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 (NIST, 2011, 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 7 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.

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
Where: k (kilo) = $\times 1,000$; M (Million) = $\times 1,000,000$; B (Billion) = $\times 1,000,000,000$.								

Table 7: 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.

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 5.8 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 $(3/2)^{2\cdot 5} \cdot (2^{14})^2 \approx 2^{34}\sigma$ if the algorithm operates regularly with 400 MB, or $(3/2)^{2\cdot 5} \cdot (2^{16})^2 \approx 2^{38}\sigma$ 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.

6.5 Summary

This chapter discussed the performance of Lyra2, showing that it is quite competitive with other PHS solutions considered among the state-of-the-art, even surpassing some of them, allowing legitimate users to use a large amount of memory while keeping the algorithm's execution time within reasonable levels.

This positive result is, at least in part, is due to the adoption of a reduced-round cryptographic sponge as underlying function, which allows more memory positions to be covered in a same amount of time than what would be possible with the application of a full-round sponge. The use of a higher bitrate during the most of the algorithm's execution (768 bits in our tests) also contributes to this higher speed without decreasing its security against possible cryptanalytic attacks to a low level, as even so the sponge's capacity can be kept quite high (256 bits in our tests).

The choice of software-oriented cryptographic sponges is similarly important, as it not only leads to high performance, but also gives more advantage to legitimate users than to attackers.

Finally, Lyra2's memory matrix was designed to allows legitimate users to take advantage of memory hierarchy features, such as caching and prefetching, which optimizes the portion of the execution time related to memory operations even with common hardware available in software-oriented platforms. These properties can also be combined with optimization strategies from modern processors, such as vectorized instructions (e.g., SSE and AVX). The end result is a fast and flexible design, which leads to a considerable cost of password-cracking hardware, comparable (and not uncommonly superior) to the best results available in the current state-of-the-art.

7 FINAL CONSIDERATIONS

In this document, 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.

7.1 Publications and other results

The following adoptions, publications, presentations, submissions and envisioned papers are a direct or indirect result of the research effort carried out during this thesis:

- *PHC special recognition*: Lyra2 received a special recognition for its elegant sponge-based design, and alternative approach to side-channel resistance combined with resistance to slow-memory attacks (PHC, 2015b).

- *BlaMka's adoption*: the winner of the PHC, Argon2, adopts BlaMka by default as its permutation function (BIRYUKOV; DINU; KHOVRATOVICH, 2016).

- *Lyra2's adoptions*: the Vertcoin electronic currency announced that it is migrating from scrypt to Lyra2 due to the excellent performance against custom mining hardware, as well as to the latter's ability to fine tune memory usage and processing time Lyra2 (A432511, 2014; DAY, 2014). Furthermore, a few months ago one of the major bitcoin mining software on GPU, the Sgminer, added support to Lyra2 in its distribution package (CRYPTO MINING, 2015).

- *Journal Articles*: in (ALMEIDA *et al.*, 2014), we present the Lyra algorithm, describing the preliminary ideas that gave rise to Lyra2; in (ANDRADE *et al.*, 2016), we describe Lyra2 and provide a brief security and performance analysis of the algorithm.

- *Extended abstract*: in (ANDRADE; SIMPLICIO JR, 2014b), we present the initial ideas and results of Lyra2, and took the opportunity of the event to discuss these results with other graduate students from the graduate program of Electrical Engineering (Computer Engineering area) at Escola Politecnica; in (ANDRADE; SIMPLICIO JR, 2014a), we gave an oral presentation at LatinCrypt'14 with some preliminary results; and in (ANDRADE; SIMPLICIO JR, 2016) we present the current status of our project.

- *Award*: recently, we received a best PhD project award due Lyra2's design

and analysis (ANDRADE; SIMPLICIO JR, 2016; ICISSP, 2016).

7.2 Future works

As future work, we plan to provide a more detailed performance and security analysis of Lyra2 with different underlying sponges and in different platforms. Among those sponges, in especial we intend to evaluate multiplication-hardened solutions, including BlaMka, since this kind of solutions would increase the protections against attacks performed with dedicated hardware.

Another interesting subject of research involves adapting Lyra2's design to allow parallel tasks during the password hashing process. This approach would allow legitimate users to take advantage of multi-core architectures (either in CPU or in GPU) for accelerating the password hashing process in their own platforms and, hence, potentially raise the memory usage for some target processing time.

Finally, we can also evaluate the consequences to adopt a password independent approach on the Lyra2 structure, aiming make it totally resistant against cache-timing attacks, evaluating the impacts of such approach when experimentally performing a slow-memory attack.

REFERENCES

A432511. *PoW Algorithm Upgrade: Lyra2 – Vertcoin*. 2014. <https://vertcoin.org/pow-algorithm-upgrade-lyra2/>. Accessed: 2015-05-06.

ALMEIDA, L. C.; ANDRADE, E. R.; BARRETO, P. S. L. M.; SIMPLICIO JR, M. A. Lyra: Password-Based Key Derivation with Tunable Memory and Processing Costs. *Journal of Cryptographic Engineering*, Springer Berlin Heidelberg, v. 4, n. 2, p. 75–89, 2014. ISSN 2190-8508. See also <http://eprint.iacr.org/2014/030>.

ANDRADE, E. R.; SIMPLICIO JR, M. A. *Lyra2: a password hashing schemes with tunable memory and processing costs*. 2014. Third International Conference on Cryptology and Information Security in Latin America, LATINCRYPT'14. Florianópolis, Brazil.<http://latincrypt2014.labsec.ufsc.br/>. Accessed: 2015-05-06.

_____. *Lyra2: Um Esquema de Hash de Senhas com custos de memória e processamento ajustáveis*. São Paulo, SP, Brazil: III WPG-EC. <http://www2.pcs.usp.br/wpgec/2014/index.htm>. Accessed: 2016-04-20., 2014. 53-55 p.

_____. Lyra2: Efficient Password Hashing with high security against Time-Memory Trade-Offs. In: INSTITUTE FOR SYSTEMS AND TECHNOLOGIES OF INFORMATION. *Doctoral Consortium – Proceedings of 2nd International Conference on Information Systems Security and Privacy, ICISSP 2016*. Rome, Italy: INSTICC. <http://www.icissp.org/?y=2016>. Accessed: 2016-04-20., 2016.

ANDRADE, E. R.; SIMPLICIO JR, M. A.; BARRETO, P. S. L. M.; SANTOS, P. C. F. d. Lyra2: efficient password hashing with high security against time-memory trade-offs. *IEEE Transactions on Computers*, PP, n. 99, 2016. ISSN 0018-9340. See also <http://eprint.iacr.org/2015/136>.

ANDREEVA, E.; MENNINK, B.; PRENEEL, B. *The Parazoa Family: Generalizing the Sponge Hash Functions*. 2011. Cryptology ePrint Archive, Report 2011/028. <http://eprint.iacr.org/2011/028>. Accessed: 2014-07-18.

AOSP. *Android Security Overview*. 2012. Android Open Source Project – <http://source.android.com/tech/security/index.html>. Accessed: 2015-02-13.

Apple. *iOS Security*. U.S., 2012. <https://www.apple.com/br/ipad/business/docs/iOS_Security_Oct12.pdf>. Accessed: 2015-06-08.

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*. Berlin, Heidelberg: Springer-Verlag, 2008. v. 5084, p. 470–488. ISBN 978-3-540-71038-7.

AUMASSON, J.-P.; GUO, J.; KNELLWOLF, S.; MATUSIEWICZ, K.; MEIER, W. Differential and Invertibility Properties of BLAKE. In: HONG, S.; IWATA, T. (Ed.). *Fast Software Encryption*. Berlin, Germany: Springer Berlin Heidelberg, 2010, (Lecture Notes in Computer Science, v. 6147). p. 318–332. ISBN 978-3-642-13857-7. See also <http://eprint.iacr.org/2010/043>.

AUMASSON, J.-P.; HENZEN, L.; MEIER, W.; PHAN, R. *SHA-3 proposal BLAKE (version 1.3)*. 2010. <https://131002.net/blake/blake.pdf>.

AUMASSON, J.-P.; JOVANOVIC, P.; NEVES, S. NORX: Parallel and Scalable AEAD. In: KUTYLOWSKI, M.; VAIDYA, J. (Ed.). *Computer Security – ESORICS 2014*. Berlin, Germany: Springer International Publishing, 2014, (LNCS, v. 8713). p. 19–36. ISBN 978-3-319-11211-4. See also <https://norx.io/>.

_____. Analysis of NORX: Investigating Differential and Rotational Properties. In: ARANHA, D. F.; MENEZES, A. (Ed.). *Progress in Cryptology – LATINCRYPT 2014*. Berlin, Germany: Springer International Publishing, 2015, (LNCS, v. 8895). p. 306–324. ISBN 978-3-319-16294-2. See also <https://eprint.iacr.org/2014/317>.

AUMASSON, J.-P.; NEVES, S.; WILCOX-O'HEARN, Z.; WINNERLEIN, C. *BLAKE2: simpler, smaller, fast as MD5*. 2013. <https://blake2.net/>. Accessed: 2014-11-21.

BELLARE, M.; RISTENPART, T.; TESSARO, S. Multi-instance Security and Its Application to Password-Based Cryptography. In: SAFAVI-NAINI, R.; CANETTI, R. (Ed.). *Advances in Cryptology – CRYPTO 2012*. Berlin, Germany: Springer Berlin Heidelberg, 2012, (LNCS, v. 7417). p. 312–329. ISBN 978-3-642-32008-8.

BERNSTEIN, D. The Salsa20 family of stream ciphers. In: ROBSHAW, M.; BILLET, O. (Ed.). *New Stream Cipher Designs*. Berlin, Heidelberg: Springer-Verlag, 2008. p. 84–97. ISBN 978-3-540-68350-6.

BERNSTEIN, D. J. *Cache-timing attacks on AES*. Chicago, 2005. <http://cr.yp.to/antiforgery/cachetiming-20050414.pdf>. Accessed: 2014-07-01.

BERTONI, G.; DAEMEN, J.; PEETERS, M.; ASSCHE, G. V. *Sponge functions*. 2007. (ECRYPT Hash Function Workshop 2007). <http://sponge.noekeon.org/SpongeFunctions.pdf>. Accessed: 2015-06-09.

_____. *Cryptographic sponge functions – version 0.1*. 2011. <http://sponge.noekeon.org/CSF-0.1.pdf>. Accessed: 2015-06-09.

_____. *The Keccak SHA-3 submission*. 2011. Submission to NIST (Round 3). <http://keccak.noekeon.org/Keccak-submission-3.pdf>. Accessed: 2015-06-09.

BIRYUKOV, A.; DINU, D.; KHOVRATOVICH, D. *Argon2: the memory-hard function for password hashing and other applications*. v1.3 of argon2. Luxembourg, 2016. <https://github.com/P-H-C/phc-winner-argon2/blob/master/argon2-specs.pdf>. Accessed: 2016-04-20.

BONNEAU, J.; HERLEY, C.; OORSCHOT, P. C. V.; STAJANO, F. The Quest to Replace Passwords: A Framework for Comparative Evaluation of Web Authentication Schemes. In: *Security and Privacy (SP), 2012 IEEE Symposium on*. San Francisco, CA: IEEEXplore, 2012. p. 553–567. ISSN 1081-6011.

BROZ, M. *Another PHC candidates "mechanical" tests – Public archives of PHC list*. 2014. <http://article.gmane.org/gmane.comp.security.phc/2237>. Accessed: 2014-11-27.

CAPCOM. *Blanka – Capcom Database*. 2015. <http://capcom.wikia.com/wiki/Blanka>. Accessed: 2015-12-18.

CHAKRABARTI, S.; SINGBAL, M. Password-based authentication: Preventing dictionary attacks. *Computer*, v. 40, n. 6, p. 68–74, june 2007. ISSN 0018-9162.

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*. Washington, DC, USA: US Department of Commerce, National Institute of Standards and Technology, 2012. <http://nvlpubs.nist.gov/nistpubs/ir/2012/NIST.IR.7896.pdf>. Accessed: 2015-03-06.

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*. Washington, DC, USA: IEEE Computer Society, 2010. (MICRO'43), p. 225–236. ISBN 978-0-7695-4299-7.

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)*. Washington, DC, USA: IEEE Computer Society, 2004. (HICSS'04, v. 7), p. 170–179. ISBN 0-7695-2056-1. <http://dl.acm.org/citation.cfm?id=962755.963150>.

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. p. 29–33.

COX, B. *TwoCats (and SkinnyCat): A Compute Time and Sequential Memory Hard Password Hashing Scheme*. v0. Chapel Hill, NC, 2014. <https://password-hashing.net/submissions/specs/TwoCats-v0.pdf>. Accessed: 2014-12-11.

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>. Accessed: 2013-12-21.

CRYPTO MINING. *Updated Windows Binary of sgminer 5.1.1 With Fixed Lyra2Re Support – Crypto Mining Blog*. 2015. <http://cryptomining-blog.

com/4535-updated-windows-binary-of-sgminer-5-1-1-with-fixed-lyra2re-support/>. Accessed: 2015-05-06.

DADDA, L. Some schemes for parallel multipliers. *Alta Frequenza*, v. 34, p. 349–356, 1965.

_____. On parallel digital multipliers. *Alta Frequenza*, v. 45, p. 574–580, 1976.

DAEMEN, J.; RIJMEN, V. A new MAC construction ALRED and a specific instance ALPHA-mac. In: GILBERT, H.; HANDSCHUH, H. (Ed.). *Fast Software Encryption – FSE'05*. Berlin, Germany: Springer Berlin Heidelberg, 2005, (LNCS, v. 3557). p. 1–17. ISBN 978-3-540-26541-2.

_____. Refinements of the ALRED construction and MAC security claims. *Information Security, IET*, v. 4, n. 3, p. 149–157, 2010. ISSN 1751-8709.

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)*. Waikoloa, HI: IEEEXplore, 2008. p. 485–485. ISSN 1530-1605.

DAY, T. *Vertcoin (VTC) plans algorithm change to Lyra2 – Coinbrief*. 2014. <http://coinbrief.net/vertcoin-algorithm-change-lyra2/>. Accessed: 2015-05-06.

DÜRMUTH, M.; GÜNEYSU, T.; KASPER, M. Evaluation of Standardized Password-Based Key Derivation against Parallel Processing Platforms. In: *Computer Security – ESORICS 2012*. Berlin, Germany: Springer Berlin Heidelberg, 2012, (LNCS, v. 7459). p. 716–733. ISBN 978-3-642-33166-4.

DWORK, C.; NAOR, M.; WEE, H. Pebbling and Proofs of Work. In: *Advances in Cryptology – CRYPTO 2005*. Berlin, Germany: Springer Berlin Heidelberg, 2005. (LNCS, v. 3621), p. 37–54. ISBN 978-3-540-28114-6.

DZIEMBOWSKI, S.; KAZANA, T.; WICHS, D. Key-Evolution Schemes Resilient to Space-Bounded Leakage. In: *Advances in Cryptology – CRYPTO 2011*. Berlin, Germany: Springer Berlin Heidelberg, 2011. (LNCS, v. 6841), p. 335–353. ISBN 978-3-642-22791-2.

FLORENCIO, D.; HERLEY, C. A Large-scale Study of Web Password Habits. In: *Proceedings of the 16th International Conference on World Wide Web*. New York, NY, USA: ACM, 2007. p. 657–666. ISBN 978-1-59593-654-7.

FORLER, C.; LIST, E.; LUCKS, S.; WENZEL, J. *Overview of the Candidates for the Password Hashing Competition - And Their Resistance Against Garbage-Collector Attacks*. 2014. Cryptology ePrint Archive, Report 2014/881. <http://eprint.iacr.org/2014/881>. Accessed: 2016-04-23.

FORLER, C.; LUCKS, S.; WENZEL, J. *Catena: A Memory-Consuming Password Scrambler*. 2013. Cryptology ePrint Archive, Report 2013/525. <http://eprint.iacr.org/2013/525>. Accessed: 2014-03-03.

_____. *The Catena Password Scrambler: Submission to the Password Hashing Competition (PHC)*. v1. Weimar, Germany, 2014. <https://password-hashing.net/submissions/specs/Catena-v1.pdf>. Accessed: 2014-10-09.

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. p. 47–56. ISBN 978-1-4503-1155-7.

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*. 2012. Cryptology ePrint Archive, Report 2012/368. <http://eprint.iacr.org/2012/368>. Accessed: 2014-03-07.

GONÇALVES, G. P. *Github of Lyra2 implementation targeting modern CPU architectures and compilers*. 2014. Github ©. <https://github.com/eggpi/lyra2>. Accessed: 2016-04-21.

GONÇALVES, G. P.; ARANHA, D. F. Implementação eficiente em software da função lyra2 em arquiteturas modernas. In: *Anais do XV Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais*. Florianópolis, SC, Brazil: Sociedade Brasileira de Computação (SBC), <http://sbseg2015.univali.br/anais/AnaisSBSeg2015Completo.pdf>. Accessed: 2015-04-22, 2005. (SBSEG 2015), p. 388–397. ISSN 2176-0063.

GUO, J.; KARPMAN, P.; NIKOLIĆ, I.; WANG, L.; WU, S. Analysis of BLAKE2. In: *Topics in Cryptology (CT-RSA 2014)*. Berlin, Germany: Springer International Publishing, 2014. (LNCS, v. 8366), p. 402–423. ISBN 978-3-319-04851-2. See also <https://eprint.iacr.org/2013/467>.

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*, ACM, New York, NY, USA, v. 52, n. 5, p. 91–98, maio 2009. ISSN 0001-0782.

HATZIVASILIS, G.; PAPAEFSTATHIOU, I.; MANIFAVAS, C. *Password Hashing Competition – Survey and Benchmark*. 2015. Cryptology ePrint Archive, Report 2015/265. <http://eprint.iacr.org/2015/265>. Accessed: 2015-05-18.

HERLEY, C.; OORSCHOT, P. V.; PATRICK, A. Passwords: If We're So Smart, Why Are We Still Using Them? In: *Financial Cryptography and Data Security*. Berlin, Germany: Springer Berlin Heidelberg, 2009. (LNCS, v. 5628), p. 230–237. ISBN 978-3-642-03548-7.

ICISSP. *Previous Awards*. 2016. International Conference on Information Systems Security and Privacy – website. <http://www.icissp.org/PreviousAwards.aspx>. 2016-04-20.

JI, L.; LIANGYU, X. *Attacks on round-reduced BLAKE*. 2009. Cryptology ePrint Archive, Report 2009/238. <http://eprint.iacr.org/2009/238>. Accessed: 2014-06-22.

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*, v. 37, n. 2, p. 179–195, 2006. ISSN 0920-8542.

KALISKI, B. *PKCS#5: Password-Based Cryptography Specification version 2.0 (RFC 2898)*. RSA Laboratories. Cambridge, MA, USA, 2000. <http://tools.ietf.org/html/rfc2898>. Accessed: 2014-03-12.

KELSEY, J.; SCHNEIER, B.; HALL, C.; WAGNER, D. Secure Applications of Low-Entropy Keys. In: *Proc. of the 1st International Workshop on Information Security*. London, UK, UK: Springer-Verlag, 1998. (ISW '97), p. 121–134. ISBN 3-540-64382-6.

KHOVRATOVICH, D.; BIRYUKOV, A.; GROBSCHÄDL, J. *Tradeoff cryptanalysis of password hashing schemes*. 2014. PasswordsCon'14. See also <https://www.cryptolux.org/images/4/4f/PHC-overview.pdf>.

KHRONOS GROUP. *The OpenCL Specification – Version 1.2*. Beaverton, OR, USA, 2012. <www.khronos.org/registry/cl/specs/opencl-1.2.pdf>. Accessed: 2014-11-11.

LENGAUER, T.; TARJAN, R. E. Asymptotically Tight Bounds on Time-space Trade-offs in a Pebble Game. *J. ACM*, ACM, New York, NY, USA, v. 29, n. 4, p. 1087–1130, oct 1982. ISSN 0004-5411.

MARECHAL, M. Advances in password cracking. *Journal in Computer Virology*, Springer-Verlag, v. 4, n. 1, p. 73–81, 2008. ISSN 1772-9890.

MENEZES, A. J.; OORSCHOT, P. C. V.; VANSTONE, S. A.; RIVEST, R. L. *Handbook of Applied Cryptography*. 1. ed. Boca Raton, FL, USA: CRC Press, 1996. ISSN 0-8493-8523-7.

MING, M.; QIANG, H.; ZENG, S. Security Analysis of BLAKE-32 Based on Differential Properties. In: IEEE. *2010 International Conference on Computational and Information Sciences (ICCIS)*. Chengdu, 2010. p. 783–786.

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. p. 19–24. ISBN 978-1-4503-1665-1.

NEVES, S. *Re: A review per day - Lyra2 – Public archives of PHC list*. 2014. <http://article.gmane.org/gmane.comp.security.phc/2045>. Accessed: 2014-12-20.

_____. *Re: Compute time hardness (pwxform,blake,blamka)*. 2015. <http://article.gmane.org/gmane.comp.security.phc/2696>. Accessed: 2016-04-22.

NIST. *Federal Information Processing Standard (FIPS 197) – Advanced Encryption Standard (AES)*. National Institute of Standards and Technology, U.S. Department of Commerce. Gaithersburg, MD, USA, 2001. <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>. Accessed: 2016-04-15.

_____. *Special Publication SP 800-38A – Recommendations for Block Cipher Modes of Operation, Methods and Techniques*. National Institute of Standards and Technology, U.S. Department of Commerce. Gaithersburg, MD, USA, 2001. <http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf>. Accessed: 2016-04-15.

_____. *Federal Information Processing Standard (FIPS PUB 180-2) – Secure Hash Standard (SHS)*. National Institute of Standards and Technology, U.S. Department of Commerce. Gaithersburg, MD, USA, 2002. <http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf>. Accessed: 2015-06-25.

_____. *Federal Information Processing Standard (FIPS PUB 198) – The Keyed-Hash Message Authentication Code*. National Institute of Standards and Technology, U.S. Department of Commerce. Gaithersburg, MD, USA, 2002. <http://csrc.nist.gov/publications/fips/fips198/fips-198a.pdf>. Accessed: 2014-04-28.

_____. *Special Publication 800-18 – Recommendation for Key Derivation Using Pseudorandom Functions*. National Institute of Standards and Technology, U.S. Department of Commerce. Gaithersburg, MD, USA, 2009. <http://csrc.nist.gov/publications/nistpubs/800-108/sp800-108.pdf>. Accessed: 2013-11-02.

_____. *Special Publication 800-63-1 – Electronic Authentication Guideline*. National Institute of Standards and Technology, U.S. Department of Commerce. Gaithersburg, MD, USA, 2011. <http://csrc.nist.gov/publications/nistpubs/800-63-1/SP-800-63-1.pdf>. Accessed: 2014-03-30.

NVIDIA. *Tesla Kepler Family Product Overview*. 2012. <http://www.nvidia.com/content/tesla/pdf/Tesla-KSeries-Overview-LR.pdf>. Accessed: 2013-10-03.

_____. *CUDA C Programming Guide (v6.5)*. 2014. <http://docs.nvidia.com/cuda/cuda-c-programming-guide/>. Accessed: 2013-10-03.

PERCIVAL, C. Cache missing for fun and profit. In: *Proc. of BSDCan 2005*. Ottawa, Canada: University of Ottawa, 2005.

_____. Stronger key derivation via sequential memory-hard functions. In: *BSDCan 2009 – The Technical BSD Conference*. Ottawa, Canada: University of Ottawa, 2009. See also: <http://www.bsdcan.org/2009/schedule/attachments/87_scrypt.pdf>. Accessed: 2013-12-09.

PESLYAK, A. *yescrypt - a Password Hashing Competition submission*. v1. Moscow, Russia, 2015. <https://password-hashing.net/submissions/specs/yescrypt-v0.pdf>. Accessed: 2015-05-22.

PHC. *Password Hashing Competition*. 2013. <https://password-hashing.net/>. Accessed: 2015-05-06.

_____. *Github of Argon2*. 2015. Github ©. <https://github.com/p-h-c/phc-winner-argon2>. Accessed: 2016-04-22.

_____. *Password Hashing Competition*. 2015. <https://password-hashing.net/#phc>. Accessed: 2016-04-23.

_____. *PHC status report*. 2015. <https://password-hashing.net/report-finalists.html>. Accessed: 2016-04-23.

_____. *Public archives of PHC list*. 2015. <http://dir.gmane.org/gmane.comp.security.phc>. Accessed: 2015-05-18.

PHC wiki. *Password Hashing Competition – wiki*. 2014. <https://password-hashing.net/wiki/>. Accessed: 2015-05-06.

PROVOS, N.; MAZIÈRES, D. A future-adaptable password scheme. In: *Proc. of the FREENIX track: 1999 USENIX annual technical conference*. Monterey, California, USA: USENIX, 1999.

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*. New York, NY, USA: ACM, 2009. (CCS '09), p. 199–212. ISBN 978-1-60558-894-0.

RIVEST, R. L.; SHAMIR, A.; ADLEMAN, L. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, ACM, New York, NY, USA, v. 21, n. 2, p. 120–126, Feb 1978. ISSN 0001-0782.

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. p. 191–204. ISBN 3-540-58108-1.

SCIENGINES. *RIVYERA S3-5000*. 2013. <http://www.sciengines.com/products/discontinued/rivyera-s3-5000.html>. Accessed: 2015-05-02.

_____. *RIVYERA V7-2000T*. 2013. <http://sciengines.com/products/computers-and-clusters/v72000t.html>. Accessed: 2015-05-02.

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*. New York, NY, USA: ACM, 1990. (SPAA'90), p. 138–145. ISBN 0-89791-370-1.

SIMPLICIO JR, M. A. *Message Authentication Algorithms for Wireless Sensor Networks*. Tese (Doutorado) — Escola Politécnica da Universidade de São Paulo (Poli-USP), São Paulo, September 2010. Available from: <http://www.teses.usp.br/teses/disponiveis/3/3141/tde-11082010-114456/>. Accessed: 2015-03-17.

SIMPLICIO JR, M. A. *Re: Competition process*. 2015. <http://article.gmane.org/gmane.comp.security.phc/2784>. Accessed: 2016-04-23.

SIMPLICIO JR, M. A.; BARBUDA, P.; BARRETO, P. S. L. M.; CARVALHO, T.; MARGI, C. The MARVIN Message Authentication Code and the LETTERSOUP Authenticated Encryption Scheme. *Security and Communication Networks*, v. 2, p. 165–180, 2009.

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*, v. 58, n. 9, p. 6223–6238, 2012.

SODERQUIST, P.; LEESER, M. An area/performance comparison of subtractive and multiplicative divide/square root implementations. In: *Proc. of the 12th Symposium on Computer Arithmetic, 1995*. Bath: IEEEXplore, 1995. p. 132–139.

SOLAR DESIGNER. *New developments in password hashing: ROM-port-hard functions*. 2012. <http://www.openwall.com/presentations/ZeroNights2012-New-In-Password-Hashing/ZeroNights2012-New-In-Password-Hashing.pdf>. Accessed: 2013-03-23.

SPRENGERS, M. *GPU-based Password Cracking: On the Security of Password Hashing Schemes regarding Advances in Graphics Processing Units*. Dissertação (Mestrado) — Radboud University Nijmegen, 2011. <http://www.ru.nl/publish/pages/578936/thesis.pdf>. Accessed: 2014-02-12.

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*. Berlin, Germany: Springer Berlin Heidelberg, 2010, (LNCS, v. 6467). p. 124–139. ISBN 978-3-642-17618-0.

SYNOPSYS. *Design Compiler 2010*. 2010. <http://www.synopsys.com/Tools/Implementation/RTLSynthesis/DesignCompiler/Pages/default.aspx>. Accessed: 2016-04-25.

TERADA, R. *Segurança de dados: Criptografia em redes de computador*. 2, revisada e ampliada. ed. São Paulo, SP, Brazil: Blucher, 2008. ISBN 978-85-212-0439-8.

THOMAS, S. *battcrypt (Blowfish All The Things)*. v0. Lisle, IL, USA, 2014. <https://password-hashing.net/submissions/specs/battcrypt-v0.pdf>. Accessed: 2015-05-18.

TRENDFORCE. *DRAM Contract Price (Jan.13 2015)*. 2015. Http://www.trendforce.com/price (visited on Jan.13, 2015).

TRUECRYPT. *TrueCrypt: Free open-source on-the-fly encryption – Documentation*. 2012. <http://www.truecrypt.org/docs/>. Accessed: 2014-09-01.

WALLIS, W. D.; GEORGE, J. *Introduction to Combinatorics*. Florence, Kentucky, USA: Taylor & Francis, 2011. (Discrete Mathematics and Its Applications). ISBN 9781439806234.

WAZLAWICK, R. S. *Metodologia de pesquisa para ciência da computação*. Rio de Janeiro: Elsevier, 2008. 184 p. ISSN 978-85-352-3522-7.

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*. Washington, DC, USA: IEEE Computer Society, 2009. (SP'09), p. 391–405. ISBN 978-0-7695-3633-0.

WU, H. *POMELO – A Password Hashing Algorithm (Version 3)*. v3. Nanyang Ave, Singapore, 2015. <https://password-hashing.net/submissions/specs/POMELO-v3.pdf>. Accessed: 2015-05-18.

XILINX. *ISE Design Suite*. 2013. <http://www.xilinx.com/products/design-tools/ise-design-suite.html>. Accessed: 2016-04-25.

_____. *Vivado Design Suite*. 2016. <http://www.xilinx.com/products/design-tools/vivado.html>. Accessed: 2016-04-25.

YAO, F. F.; YIN, Y. L. Design and Analysis of Password-Based Key Derivation Functions. *IEEE Transactions on Information Theory*, v. 51, n. 9, p. 3292–3297, 2005. ISSN 0018-9448.

YUILL, J.; DENNING, D.; FEER, F. Using deception to hide things from hackers: Processes, principles, and techniques. *Journal of Information Warfare*, v. 5, n. 3, p. 26–40, 2006. ISSN 1445-3312.

APPENDIX A. NAMING CONVENTIONS

The name "Lyra" comes from *Chondrocladia lyra*, a recently discovered type of sponge (CREW, 2012). 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 (ALMEIDA *et al.*, 2014), 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 (CAPCOM, 2015) that comes from Brazil and, as such, is a compatriot of this document's authors. ☺

APPENDIX B. FURTHER CONTROLLING LYRA2'S BANDWIDTH USAGE

Even though not part of Lyra2's core design, the algorithm could be adapted for allowing the user to control the number of rows involved in each iteration of the Visitation Loop. The reason is that, while Algorithm 5 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 5 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 5.5 and 5.8 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 Equation 5.3.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 (SOLAR DESIGNER, 2012).

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 6). 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 do not explore further the ability of adjusting δ to a value different than 1.

APPENDIX C. AN ALTERNATIVE DESIGN FOR BLAMKA: AVOIDING LATENCY

One drawback of the BlaMka permutation function as described in Section 4.4.1 is that it has no instruction parallelism. In platforms where a legitimate user can execute instructions in parallel (e.g., in modern x86 processors) this can become disadvantage, as the design will not take full advantage of the hardware available for the legitimate user. This potential disadvantage can be addressed with a quite simple tweak, which consists in replacing the addition (+) introduced by BlaMka by a XOR operation.

This alternative structure is shown in Figure 18, which shows Blake2b's original G function (on the left) and the two multiplication-hardened modified functions proposed on this work, $G_{\texttt{tls}}$ as described in Section 4.4.1 and an alternative, XOR-based permutation hereby denoted called $G_{\texttt{tls}\oplus}$ aiming at reduced latency.

$$
\begin{aligned}
a &\leftarrow a + b \\
d &\leftarrow (d \oplus a) \ggg 32 \\
c &\leftarrow c + d \\
b &\leftarrow (b \oplus c) \ggg 24 \\
a &\leftarrow a + b \\
d &\leftarrow (d \oplus a) \ggg 16 \\
c &\leftarrow c + d \\
b &\leftarrow (b \oplus c) \ggg 63
\end{aligned}
$$

(a) Blake2 G function.

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

(b) BlaMka's $G_{\texttt{tls}}$ function.

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

(c) BlaMka's $G_{\texttt{tls}\oplus}$ function.

Figure 18: Different permutations: Blake2b's original permutation (left), BlaMka's $G_{\texttt{tls}}$ multiplication-hardened permutation (middle) and BlaMka's latency-oriented multiplication-hardened permutation $G_{\texttt{tls}\oplus}$ (right).

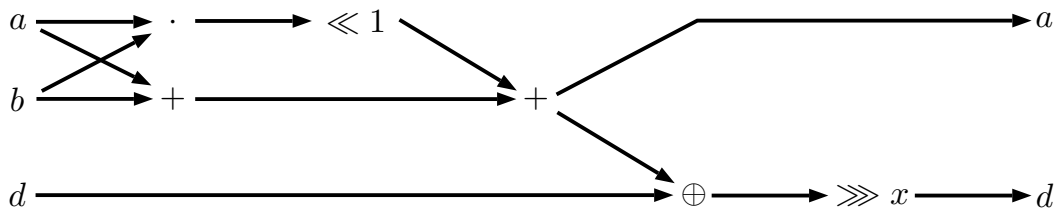
This trick was originally proposed on NORX – a Parallel and Scalable Authenticated Encryption Algorithm – (AUMASSON; JOVANOVIC; NEVES, 2014, Section 5.1), but during the PHC, Samuel Neves, one of the authors of this algorithm, suggested the possibility of using this approach also on BlaMka (NEVES, 2015).

With this modification, in principle a developer can trade a few extra instructions by reduced latency, shortening the execution's pipeline, by implementing it as follows:

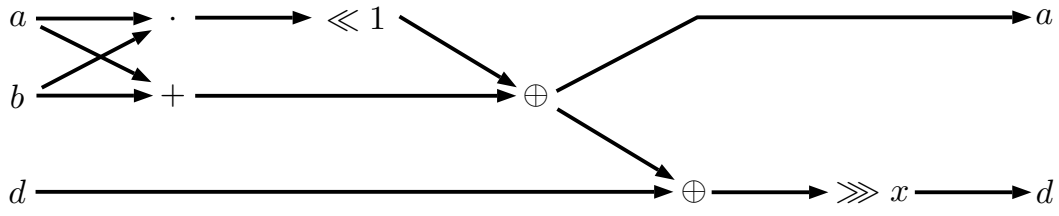
$$\begin{array}{rcl} a & \leftarrow & a + b \oplus 2 \cdot \mathrm{lsw}(a) \cdot \mathrm{lsw}(b) \\ d & \leftarrow & (d \oplus a) \ggg x \end{array} \quad \Longrightarrow \quad \begin{array}{rcl} t_0 & \leftarrow & a + b \\ t_1 & \leftarrow & \mathrm{lsw}(a) \cdot \mathrm{lsw}(b) \\ t_1 & \leftarrow & t_1 \ll 1 \\ a & \leftarrow & t_0 \oplus t_1 \\ d & \leftarrow & d \oplus t_0 \\ d & \leftarrow & d \oplus t_1 \\ d & \leftarrow & d \ggg x \end{array}$$

This tweak saves up to 1 cycle per instruction sequence, leading to 4 instead of 5 cycles for $G_{\texttt{tls}^{\oplus}}$, at the cost of 1 extra instruction (see Figure 19). In a sufficiently parallel architecture, this can save at least 4×8 cycles per round, since a single round of BlaMka has 8 calls to its underlying permutation.

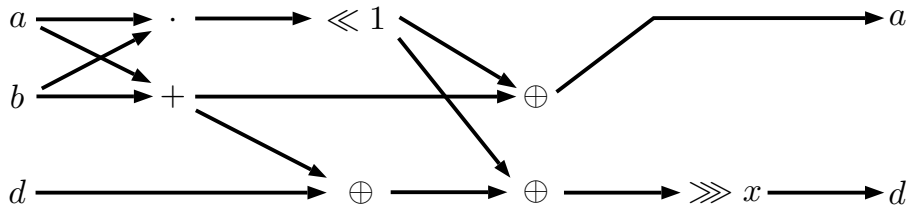
In our initial measurements, this modification represented a somewhat modest gain of performance in legitimate platforms, which reached up to 6% in Lyra2 when compared with BlaMka using the hereby adopted $G_{\texttt{tls}}$ permutation. However, our tests revealed that the ability to obtain this gain would heavily depend on the target architecture, coding, compiler, and type of vectorization. Therefore, as it would be difficult to have the same gain in practice for all legitimate platforms, while attackers could very well use dedicated hardware to take more advantage of it than legitimate users, we preferred not to adopt this approach in the design of the BlaMka sponge.



(a) Naïve implementation of BlaMka's $G_{\texttt{tls}}$ instruction sequence.



(b) Naïve implementation of BlaMka $G_{\texttt{tls}^{\oplus}}$ instruction sequence.



(c) Latency-oriented version of BlaMka's $G_{\texttt{tls}^{\oplus}}$ instruction sequence.

Figure 19: Improving the latency of G.