LEONARDO DE CAMPOS ALMEIDA

LYRA: UMA FUNÇÃO DE DERIVAÇÃO DE CHAVES COM CUSTOS DE MEMÓRIA E PROCESSAMENTO CONFIGURÁVEIS

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Mestre em Engenharia Elétrica.

São Paulo
2016

LEONARDO DE CAMPOS ALMEIDA

LYRA: UMA FUNÇÃO DE DERIVAÇÃO DE CHAVES COM CUSTOS DE MEMÓRIA E PROCESSAMENTO CONFIGURÁVEIS

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Mestre em Engenharia Elétrica.

Área de Concentração:
Engenharia de Computação

Orientador:
Prof. Dr. Marcos Simplicio Júnior

São Paulo
2016

Este exemplar foi revisado e alterado em relação à versão original, sob responsabilidade única do autor e com a anuência de seu orientador.

São Paulo, 9 de maio de 2016.

Assinatura do autor

Assinatura do orientador

FICHA CATALOGRÁFICA

Almeida, Leonardo de Campos
Lyra: uma função de derivação de chaves com custos de memória e processamento configuráveis/ L. de Campos Almeida. – ed. rev. – São Paulo, 2016.
69 p.

Dissertação (Mestrado) — Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais (PCS).

1. Criptografia 2. Segurança de Redes 3. Memória RAM I. Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Computação e Sistemas Digitais (PCS). II. t.

Dedico este trabalho à minha esposa, com amor e admiração, por sua compreensão e apoio durante a execução deste trabalho.

AGRADECIMENTOS

Ao Professor Doutor Marcos Simplicio Júnior, pela atenção, apoio e dedicação, durante o processo de confecção deste trabalho, que contribuiu para meu crescimento científico e intelectual.
Aos doutorandos Ewerton Rodrigues de Andrade e Paulo Carlos Ferreira dos Santos pelo apoio e colaboração na execução do trabalho. À Escola Politécnica da USP, pela oportunidade de realização do curso.

RESUMO

Este documento apresenta o Lyra, um novo esquema de derivação de chaves, baseado em esponjas criptográficas. O Lyra foi projetado para ser estritamente sequencial, fornecendo um nível elevado de segurança mesmo contra atacantes que utilizem múltiplos núcleos de processamento, como uma GPU ou FPGA. Ao mesmo tempo possui uma implementação simples em software e permite ao usuário legítimo ajustar o uso de memória e tempo de processamento de acordo com o nível de segurança desejado. O Lyra é, então, comparado ao scrypt, mostrando que esta proposta fornece um nível se segurança mais alto, além de superar suas deficiências. Caso o atacante deseje realizar um ataque utilizando pouca memória, o tempo de processamento do Lyra cresce exponencialmente, enquanto no scrypt este crescimento é apenas quadrático. Além disto, para o mesmo tempo de processamento, o Lyra permite uma utilização maior de memória, quando comparado ao scrypt, aumentando o custo de ataques de força bruta.

ABSTRACT

This document presents Lyra, a password-based key derivation scheme based on cryptographic sponges. Lyra was designed to be strictly sequential, providing strong security even against attackers that use multiple processing cores, such as FPGAs or GPUs. At the same time, it is very simple to implement in software and allows legitimate users to tune its memory and processing costs according to the desired level of security. We compare Lyra with scrypt, showing how this proposal provides a higher security level and overcomes limitations of scrypt. If the attacker wishes to perform a low-memory attack against the algorithm, the processing cost grwos expontetialy, while in scrypt, this growth is only quadratic. In addition, for an identical processing time, Lyra allows for a higher memory usage than its counterparts, further increasing the cost of brute force attacks.

SUMÁRIO

Lista de Ilustrações

Lista de Tabelas

Lista de Abreviaturas e Siglas

Lista de Símbolos

1 Introdução **12**

1.1 Motivação 14

1.2 Objetivos 14

1.3 Justificativa 15

1.4 Métodos 16

1.5 Organização 17

2 Fundamentação Teórica **18**

2.1 Ataques de Força Bruta 18

2.2 Ataques Utilizando GPUs e FPGAs 19

2.2.1 GPUs 20

2.2.2 FPGAs 21

2.3 Funções Esponja 22

2.3.1 Construção Básica 22

2.3.2 Construção Duplex 23

3 Revisão da Literatura: Funções de Derivação de Chave 25

3.1 PBKDF2 25

3.2 Bcrypt 26

3.3 Scrypt 28

3.4 PHC 31

3.4.1 Catena 31

3.4.2 Yescrypt 32

3.4.3 Argon2 33

4 Lyra 34

4.1 Estrutura 34

4.1.1 *Setup* 34

4.1.2 *Wandering* 37

4.1.3 *Wrap-up* 38

4.2 Projeto estritamente sequencial 38

4.3 Configurando a quantidade de memória e tempo de processamento 41

4.4 Função esponja subjacente 42

4.5 Considerações Práticas 42

4.6 Paralelismo em plataformas legítimas 44

4.7 Análise de segurança 46

4.7.1 Ataque com pouca memória 47
4.7.1.1 Notações 49
4.7.1.2 Cenário 1: Armazenando todos os estados internos 50
4.7.1.3 Cenário 2: Armazenando poucos ou nenhum estado interno. 52
4.7.1.4 Resumo 53
4.7.2 Ataques com memória lenta 54
4.8 Desempenho com os parâmetros recomendados 55
4.8.1 LyraP 58
4.8.2 Expectativa de custo de ataques 60

5 Conclusões e Próximos Passos 63

Referências 65

LISTA DE ILUSTRAÇÕES

1 Construção básica da função esponja 24

2 Construção duplex 24

3 Exemplo simplificado da operação da fase *Wandering*. 50

4 Desempenho do Lyra comparado com scrypt 56

5 Desempenho do Lyra comparado com os finalistas do PHC 57

6 Desempenho do Lyra comparado com o Lyra2 58

7 Desempenho do $Lyra_p$ 59

LISTA DE TABELAS

1 Custo de memória do Lyra 61

LISTA DE ABREVIATURAS E SIGLAS

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

API Application Programming Interface

CPU Central Processing Unit

NIST National Institute of Standards and Technology

PKCS Public-Key Cryptography Standards

RAM Random Access Memory

NIST National Institute of Standards and Technology

LISTA DE SÍMBOLOS

$\oplus$	Operação XOR
$\|\|$	Concatenação
$\|x\|$	Quantidade de bits necessários para representar x

1 INTRODUÇÃO

Atualmente, a autenticação de usuários é um dos elementos vitais da segurança de computação. Os três principais mecanismos de autenticação são:

- Senhas (O que o usuário sabe);
- Dispositivos biométricos (O que o usuário é ou faz);
- Dispositivos físicos (O que o usuário possui).

Estes mecanismos não precisam ser utilizados de maneira independente, sendo possível combiná-los, melhorando a segurança do sistema de autenticação. De fato, sistemas que exigem maiores níveis de segurança podem utilizar mais de um fator para autenticar o usuário, como é o caso de alguns bancos, que exigem senha e um dispositivo físico para que o usuário consiga realizar operações em sua conta.

Na maioria dos sistemas computacionais, entretanto, a utilização de senhas continua sendo o método de autenticação mais utilizado, devido à sua conveniência e baixo custo (CHAKRABARTI; SINGBAL, 2007), (CONKLIN; DIETRICH; WALZ, 2004). Mesmo havendo tecnologias mais modernas e seguras, é bem provável que as senhas continuem a ser o principal fator utilizado na autenticação de usuários no futuro próximo (BONNEAU *et al.*, 2012). Porém, geralmente, são escolhidas senhas curtas e de fácil memorização por humanos, sendo menos resistentes a ataques (NIST, 2011). Por exemplo, um

estudo de 2007 com 544.960 senhas de usuário reais (FLORENCIO; HERLEY, 2007) mostra uma entropia média de aproximadamente 40,5 bits, comprovando esta escolha de senhas com um nível de segurança bem inferior aos 128 bits recomendados para o uso em sistemas modernos, a fim de dificultar ataques. Entretanto, senhas com maior complexidade são mais difíceis de serem memorizadas por humanos, fazendo com que, muitas vezes, o usuário opte por armazenar esta senha em papéis ou documentos, diminuindo sua segurança, visto que o atacante pode conseguir acesso físico a esta senha (CHAKRABARTI; SINGBAL, 2007).

Frequentemente, as senhas são transmitidas via rede, durante o procedimento de autenticação, sendo que em alguns casos esta senha é transmitida às claras, como no *telnet*, por exemplo. Também existem sistemas que guardam as senhas em arquivos, armazenados em um computador. Nestes dois casos, é possível que um atacante consiga ter acesso às senhas dos usuários diretamente, monitorando a rede ou acessando o sistema de arquivos do computador.

A fim de evitar este ataque, frequentemente, é utilizado o *hash* da senha, ao invés de texto às claras. Como o *hash* é não inversível, se o atacante tiver acesso a esta informação, não irá conseguir obter a senha do usuário. Porém, o atacante pode se valer de ataques de força bruta, testando diversas senhas, até descobrir a senha correta do usuário.

Para tornar tais ataques de força bruta mais custosos, pode-se aplicar fortalecimento de chaves sobre a senha. Este método consiste em aplicar uma função de derivação de chaves (*Key Derivation Function* – KDF), também conhecida como esquema de *hash* de senha (*Password Hashing Scheme* – PHS) que requer 2^s operações criptográficas para ser calculada, onde s é um parâmetro da função. Desta forma, o custo do ataque de força bruta, realizado

sobre uma senha de t bits de entropia, aumenta de 2^t para 2^{s+t} operações (PERCIVAL, 2009). A saída do PHS também pode ser utilizada como chave em operações criptográficas.

1.1 Motivação

Os PHSs mais antigos, como PBKDF2 (KALISKI, 2000) e bcrypt (PROVOS; MAZIÈRES, 1999), incluem um parâmetro configurável que controla a quantidade de iterações executadas, permitindo ao usuário configurar o tempo de processamento necessário para a computação da chave. Porém, caso o atacante disponha de uma plataforma equipada com múltiplos núcleos, como uma GPU ou FPGA, é possível executar múltiplas instâncias destes PHSs, diminuindo drasticamente o tempo necessário para descobrir uma senha através de um ataque de força bruta.

Uma proposta mais recente, o scrypt (PERCIVAL, 2009), permite que o usuário controle o tempo de execução e também o consumo de memória. Assim, ao aumentar a quantidade de memória necessária para computar a chave, aumenta-se também o custo do ataque de força bruta, pois o atacante deve dispor de uma GPU com maior quantidade de memória RAM ou aumentar a área de silício na FPGA. Porém, o scrypt também possui algumas limitações/ineficiências, como o acoplamento entre custo de tempo e memória, fazendo com que o aumento do custo de processamento eleve o custo de memória. Estes pontos fracos motivaram a busca por novas alternativas.

1.2 Objetivos

O objetivo deste projeto de pesquisa de mestrado é criar um novo PHS, chamado Lyra, que combine os pontos fortes das soluções dos PHS já exis-

tentes, como flexibilidade, parametrização de tempo de processamento e consumo de memória, mas que seja capaz de prover maior segurança contra ataques do que estes. Esta solução consiste em um novo modo de operação das funções esponja (BERTONI *et al.*, 2007; BERTONI *et al.*, 2011a), voltado para derivação de chave.

1.3 Justificativa

Com o aumento do poder computacional, os ataques à senhas se tornam mais elaborados e eficazes, fazendo com que seja necessário criar um PHS que consiga resistir a estes ataques. Este PHS deve conseguir aproveitar os pontos fortes dos esquemas já existentes, mitigando suas fraquezas.

Para desenvolver uma solução melhor que as existentes atualmente, o novo esquema deverá permitir que, para o mesmo tempo de processamento, seja possível aumentar o consumo de memória. Além disto, o custo de se realizar um ataque com custo reduzido de memória deve ser superior ao custo de se realizar o mesmo tipo de ataque contra o scrypt, um dos poucos PHSs existentes no momento em que esta dissertação estava sendo construída, que exploram o uso de memória. Assim, o scrypt servirá como a principal base de comparação do Lyra.

Com base na necessidade de prover uma solução de derivação de chaves mais segura e moderna, em 2013 foi criada a PHC (*Password Hashing Competition*) , com o objetivo de melhorar o estado da arte de PHSs, além de recomendar a utilização de proteções de senhas mais fortes. Em razão desta competição, surgiram novos PHSs. Foram propostos 24 novos esquemas, sendo que dois deles se retiraram da competição voluntariamente (PHC, 2013). O vencedor da competição foi o Argon2 (BIRYUKOV; DINU; KHOVRA-

TOVICH, 2015). Além destes novos esquemas, também foi apresentado o Lyra2 (**??**), que foi baseado no Lyra. Suas vantagens em relação ao Lyra são explorar maior uso de banda e outras formas de paralelismo, mantendo a estrutura básica que foi definida no Lyra. O Lyra2 foi um dos finalistas da competição, recebendo reconhecimento especial pelo seu desenho elegante.

1.4 Métodos

O método utilizado neste trabalho foi pesquisa aplicada, baseada em hipótese-dedução, utilizando referências científicas para definição do problema, especificação da hipótese de solução e sua avaliação.

O trabalho foi separado de acordo com as seguintes etapas:

- **Pesquisa bibliográfica**: realização de levantamento dos PHSs existentes, com base em leitura e análise de artigos técnicos e científicos. A partir de comparações entre as soluções existentes, avaliando suas estruturas internas, segurança e desempenho, sendo possível determinar a abordagens atrativas para a criação do algoritmo proposto;

- **Projeto do algoritmo**: criação de um novo PHS, chamado Lyra, que possua a mesma flexibilidade das funções existentes, mas com maior segurança. Criação da implementação de referência para validação e implementação otimizada para desempenho, possibilitando a comparação com as soluções existentes até o momento.

- **Comparação com as soluções existentes até o momento**: comparação entre as estruturas dos PHSs atuais e do Lyra, a fim de verificar se, por definição, a segurança obtida no Lyra é maior, quando comparada aos PHSs atuais; Realização de testes de desempenho.

- **Escrita da dissertação**: confecção da dissertação, englobando as soluções existentes até o momento, projeto do Lyra e comparação entre PHSs.

Estas etapas serão executadas em paralelo, quando possível.

1.5 Organização

Este documento está organizado em capítulos, distribuídos conforme a seguir.

O capítulo 2 fornece a base para o entendimento dos PHSs discutidos.

O capítulo 3 discute os PHSs existentes hoje.

O capítulo 4 consolida os resultados parciais obtidos até o momento.

O capítulo 5 finaliza a discussão e discute os próximos passos planejados.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo traz a fundamentação teórica para o entendimento do restante do texto, abordando os conceitos de funções de derivação de chave e ataques, utilizando diferentes plataformas, como GPUs e FPGAs. Esta seção também discute o conceito de funções esponja, que servem como base para a construção do algoritmo proposto.

2.1 Ataques de Força Bruta

Em alguns cenários, as senhas de usuários são armazenadas encriptadas em arquivos de texto. Caso um atacante consiga acesso a este arquivo, ele pode tentar descobrir as senhas por tentativa e erro, sem que seja possível bloquear o ataque, pois não há limitação na quantidade de tentativas. Este cenário é diferente do que ocorre em sistemas *online*, nos quais é possível bloquear a senha do usuário após algumas tentativas incorretas (CONKLIN; DIETRICH; WALZ, 2004).

Nas suas primeiras versões, o sistema operacional Unix armazenava o *hash* das senhas de usuários em arquivo, utilizando uma função não reversível, chamada *crypt*. Com o aumento do poder computacional disponível para ataques, tornou-se mais barato descobrir a senha de um determinado usuário, calculando o *hash* de todas as senhas possíveis e comparando com o valor armazenado no arquivo, até que se encontre a senha correta (PROVOS; MA-

ZIÈRES, 1999). Este tipo de ataque, no qual se calcula o *hash* de todas as senhas possíveis, a fim de descobrir uma senha encriptada, é chamado de ataque de força bruta.

Com o objetivo de fornecer maior segurança contra este tipo de ataque, surgiram as funções de derivação de chaves. Isto é feito através do aumento do custo computacional necessário para gerar a senha encriptada. Os primeiros esquemas de derivação de chaves conseguiam este aumento através de um parâmetro, que controla o número de execuções da função de *hash* interna. Assim, é possível parametrizar a função de derivação, de forma que a sua execução demore um tempo aceitável para o usuário legítimo, mas proibitivo para ataques de força bruta. Por exemplo, se um PHS demora 1 segundo para ser executada, calcular um espaço de senhas pequeno, de 100.000 senhas, demoraria aproximadamente 28 horas. Estes esquemas de derivação de senhas também utilizam outro recurso, conhecido como sal. O sal é um valor pseudo-aleatório, concatenado à senha, fazendo com que senhas iguais gerem *hashes* diferentes, dificultando o ataque através de tabelas com *hashes* pré-computados.

2.2 Ataques Utilizando GPUs e FPGAs

Plataformas que exploram paralelismo massivo e barato representam um perigo aos PHSs, pois passa a ser possível executar muitas instâncias em paralelo, reduzindo drasticamente o tempo necessário para realizar um ataque de força bruta. As principais plataformas, utilizadas para este fim, são as Unidades de Processamento Gráfico (GPUs) e hardware personalizado através de FPGAs (DÜRMUTH; GÜNEYSU; KASPER, 2012).

2.2.1 GPUs

Com a crescente demanda por renderização de gráficos com alta definição em tempo real, as GPUs passaram a ser equipadas com um grande número de núcleos de processamento, aumentando sua capacidade de paralelismo. Recentemente, as GPUs deixaram de ser plataformas específicas para processamento de gráficos e começaram a suportar linguagens padronizadas, como CUDA (Nvidia, 2012) e OpenCL (KHRONOS GROUP, 2012), que ajudaram a utilizar todo seu poder computacional. Com isto, as GPUs passaram a ser utilizadas para propósitos mais gerais, incluindo ataque à senhas (SPRENGERS, 2011; DÜRMUTH; GÜNEYSU; KASPER, 2012).

As GPUs modernas são equipadas com alguns milhares de núcleos de processamento, fazendo com que a execução de múltiplas *threads* em paralelo, se torne simples e barata. Assim, a GPU torna-se uma plataforma atrativa, quando o objetivo é testar múltiplas senhas em paralelo ou paralelizar instruções internas de um PHS. Por exemplo, a NVidia Tesla K20X possui 2.688 núcleos de processamento operando a 732 MHz, bem como 6 GB de DRAM compartilhada, com uma banda de 250 GB por segundo (NVIDIA, 2012). Supondo que esta GPU seja utilizada para atacar um PHS, que foi parametrizada para executar em um segundo e consumir menos de 2,33 MB de memória, é simples conceber uma implementação que consiga testar 2.688 senhas por segundo, sendo uma por núcleo. Porém, ao aumentar o consumo de memória, o número de testes por segundo diminuiria, devido à limitação de 6 GB de memória da GPU. Por exemplo, se um PHS sequencial exige 20 MB de memória, seria possível utilizar apenas 300 núcleos em paralelo, ou seja, 11% do total disponível.

2.2.2 FPGAs

Uma FPGA é um conjunto de blocos lógicos, configuráveis, ligados entre si e com elementos de memória, formando um circuito integrado de alto desempenho e programável. Estes dispositivos são configurados para realizar uma tarefa específica, sendo altamente otimizados para seu propósito. Geralmente, as FPGAs são soluções mais eficazes em termos de custos, quando comparadas às CPUs de propósito genérico. A FPGA ainda possui um menor consumo de energia em comparação às GPUs (CHUNG *et al.*, 2010; FOWERS *et al.*, 2012). Assim, o seu uso para ataques de força bruta contra senhas é algo interessante, tanto em relação ao desempenho, quanto ao custo.

Dürmuth, Güneysu e Kasper (2012) apresenta um exemplo recente de ataque à senhas, utilizando um *cluster* de 128 FPGAs, modelo RIVYERA S3-5000 (SCIENGINES, 2013a), para atacar o PHS PBKDF2-SHA-512, no qual foi possível testar 356.352 senhas por segundo, em uma arquitetura que permite processar 5,376 senhas em paralelo. Uma razão que tornou este resultado possível foi o baixo consumo de memória do PBKDF2, visto que a maior parte do processamento interno do SHA-512 é realizado dentro do *cache* do dispositivo. Se o PHS consumisse 20 MB de memória, o resultado seria um número menor de testes por segundo. Como cada uma das FPGAs utilizadas no *cluster* possui até 64MB de memória, seria possível processar apenas três senhas em paralelo por FPGA.

Um PHS que utilize 20MB de memória traria dificuldades, mesmo para um *cluster* mais poderoso como o RIVYERA V7-2000T (SCIENGINES, 2013b). Este *cluster* é equipado com até quatro FPGAs Xilinx Virtex-7, cada uma com 20 GB de memória, além de até 128 GB de memória compartilhada. Conside-

rando este PHS, não seria possível testar mais de 2.600 senhas em paralelo.

Devido ao grande consumo de memória deste PHS, não é possível sintetizar um *hardware* genérico para todas as senhas possíveis. Seria possível sintetizar FPGAs específicas para cada senha comum, porém, também seria necessário gerar um *hardware* para cada sal, tornando esta técnica inviável. As FPGAs, geralmente, fazem uso de *pipeline* para acelerar o processamento, porém, se este PHS realizar leituras e escritas de forma pseudoaleatória em sua memória interna, o *pipeline* não consegue prever qual o próximo trecho de memória que será acessado, perdendo sua eficiência.

2.3 Funções Esponja

O conceito de esponjas criptográficas foi introduzido formalmente por Bertoni *et al.* (2007), sendo mais amplamente detalhado em (BERTONI *et al.*, 2011a). O projeto elegante e flexível das esponjas para a construção de diversas funções criptográficas, em especial (mas não apenas) funções de hash, motivou ainda a criação de estruturas mais genéricas, como a família de funções Parazoa (ANDREEVA; MENNINK; PRENEEL, 2011). Recentemente, um dos membros da família esponja, Keccak, foi escolhido como o novo *Secure Hash Algorithm* (SHA-3), após um concurso público que durou anos (BERTONI *et al.*, 2011b).

2.3.1 Construção Básica

Basicamente, as esponjas criptográficas permitem a construção de funções de *hash* com tamanhos de entrada e saída arbitrários. Estas funções são baseadas na chamada construção esponja, um modo de operação iterado que utiliza uma permutação (ou outro tipo de transformação) de tamanho

fixo denotada f, bem como uma regra de preenchimento (*padding*) denotada pad. Como pode ser observado na figura 1, as funções esponja operam sobre um estado interno de w bits, inicialmente de valor zero, tendo como entrada uma mensagem M preenchida de acordo com a regra de pad e dividida em blocos de tamanho b.

A operação de uma esponja é composta por duas fases. Na primeira, denominada *absorb*, aplica-se f iterativamente sobre o estado interno da esponja e a entrada M. Já na segunda fase, denominada *squeeze*, a função f é aplicada iterativamente ao estado interno da esponja, gerando a saída por partes. A operação é finalizada quando todos os bits da entrada foram consumidos na fase *absorb* e mapeados na saída de tamanho l na fase *squeeze*. Geralmente, a função f é iterativa, sendo parametrizada por um número de rodadas (e.g., 24 para o Keccak com palavras de 64 bits (BERTONI *et al.*, 2011b)).

Os parâmetros w, b e c são chamados, respectivamente, de $comprimento$, $taxadebits$ e $capacidade$ da esponja. O estado interno da esponja é dividido em duas partes: a parte externa, de tamanho b, que interage diretamente com a entrada M; e a parte interna, de tamanho c, que é afetada apenas pela função f. Seguindo a convenção *little-endian* ao longo do documento, os bits menos significativos são aqueles que fazem parte do estado externo.

2.3.2 Construção Duplex

Uma outra maneira de operação das funções esponja é chamada de construção duplex, ilustrada na figura 2. A diferença com relação à construção básica da esponja é que esta última não retém o valor do estado interno entre chamadas, ao passo que a construção duplex o faz. A construção duplex recebe uma entrada de tamanho variável e fornece uma saída de tamanho va-

riável, que depende de todas as entradas recebidas anteriormente. Ou seja, apesar do estado interno ser inicializado com zeros quando a esponja é instanciada, esse estado é armazenado entre as chamadas à função duplex, ao invés de ser inicializado com zeros novamente.

Na construção duplex, o tamanho da entrada após o preenchimento, deve ser menor do que b, e o comprimento da saída deve respeitar a relação $l \leq b$ (BERTONI *et al.*, 2011a).

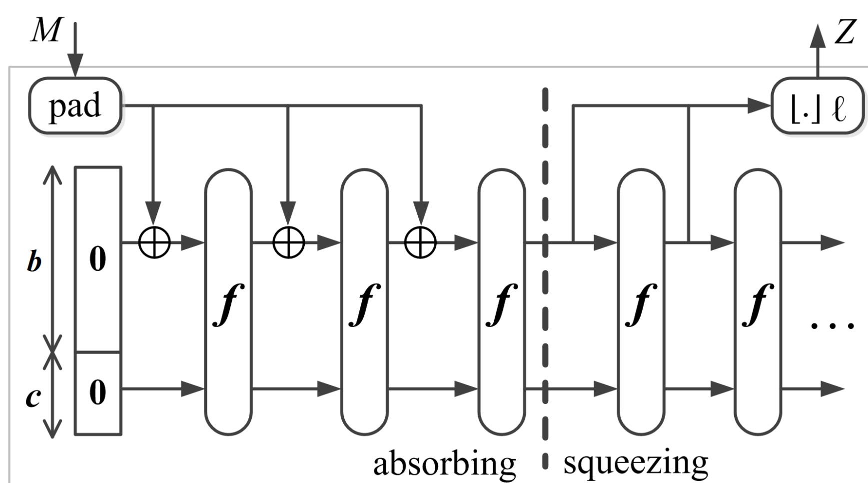

Figura 1: Construção básica da função esponja $Z = [f, \mathrm{pad}, b](M, \ell)$. Adaptado de (BERTONI *et al.*, 2011a).

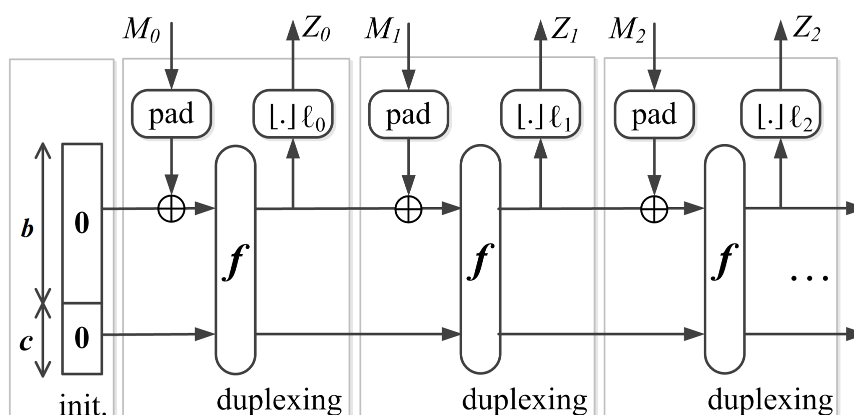

Figura 2: Construção duplex. Adaptado de (BERTONI *et al.*, 2011a).

3 REVISÃO DA LITERATURA: FUNÇÕES DE DERIVAÇÃO DE CHAVE

As seções a seguir explicam com maiores detalhes o projeto e modo de funcionamento dos algoritmos PBKDF2, bcrypt, scrypt e alguns dos algoritmos submetidos à PHC. Os três primeiros algoritmos foram escolhidos porque são PHSs bem conhecidos e utilizados atualmente.

3.1 PBKDF2

O algoritmo *Password-Based Key Derivation Function version 2* (PBKDF2) foi proposto originalmente no ano 2000, como parte do PKCS#5 do laboratório RSA (KALISKI, 2000). Este algoritmo está presente, atualmente, em diversas ferramentas de segurança, como TrueCrypt (TRUECRYPT, 2012) e o iOS da Apple, no qual é utilizado para criptografar senhas do usuário, além de possuir diversas análises formais (YAO; YIN, 2005; BELLARE; RISTENPART; TESSARO, 2012).

Como mostrado no Algoritmo 1, o PBKDF2 aplica a função pseudoaleatória subjacente $Hash$ à concatenação da senha $senha$ e uma variável U_i, executando $U_i = Hash(senha, U_{i-1})$ para cada iteração $1 \leqslant i \leqslant T$. O valor inicial U_0 corresponde à concatenação do sal fornecido pelo usuário e uma variável l, que corresponde ao número de blocos de saída. O bloco l da chave de comprimento k é calculado da seguinte forma: $K_l = U_1 \oplus U_2 \oplus \ldots \oplus U_T$, onde

Algorithm 1 PBKDF2.

```
Input: senha   ▷ A senha
Input: sal   ▷ O sal
Input: T   ▷ O parâmetro de custo
Output: K   ▷ A chave derivada da senha
 1: if k > (2^32 − 1) · h then
 2:     return Derived key too long.
 3: end if
 4: l ← ⌈k/h⌉ ;  r ← k − (l − 1) · h
 5: for i ← 1 to l do
 6:     U[1] ← PRF(senha, sal||INT(i))   ▷ INT(i): i codificado em 32 bits
 7:     T[i] ← U[1]
 8:     for j ← 2 to T do
 9:         U[j] ← PRF(senha, U[j − 1]) ;  T[i] ← T[i] ⊕ U[j]
10:     end for
11:     if i = 1 then  K ← T[1]  else  K ← K || T[i]  end if
12: end for
13: return K
```

k é o comprimento desejado da chave.

O PBKDF2 permite a configuração de seu tempo total de execução, através do parâmetro T. Como o processo de derivação de chave é estritamente sequencial, não sendo possível calcular U_i sem antes calcular U_{i-1}, sua estrutura interna não pode ser paralelizável. Porém, como o PBKDF2 utiliza pequenas quantidades de memória, o custo de implementar ataques de força bruta contra o algoritmo, utilizando múltiplas unidades de processamento, é razoavelmente baixo.

3.2 Bcrypt

Outra solução que permite ao usuário configurar o tempo de processamento da derivação da chave é o bcrypt (PROVOS; MAZIÈRES, 1999). Este esquema é baseado em uma versão do algoritmo de cifração de 64 bits, Blowfish (SCHNEIER, 1994), sendo chamado *EksBlowflish* (“expensive key schedule blowfish”).

Os dois algoritmos utilizam o mesmo processo de cifração, diferindo apenas na maneira como as sub-chaves e *S-boxes* são calculadas. A operação do bcrypt consiste em inicializar as sub-chaves e *S-boxes* com o sal e a senha,

Algorithm 2 Bcrypt.

```
Input: senha  ▷ A senha
Input: sal  ▷ O sal
Input: T  ▷ O parâmetro de custo de processamento
Output: K  ▷ A chave derivada da senha
 1: s ← InitState()  ▷ Copia os primeiros dígitos de π nas sub-chaves e S-boxes S_i
 2: s ←ExpandKey(s, sal, senha)
 3: for i ← 1 to 2^T do
 4:     s ←ExpandKey(s, 0, sal)
 5:     s ←ExpandKey(s, 0, senha)
 6: end for
 7: ctext ← "OrpheanBeholderScryDoubt"
 8: for i ← 1 to 64 do
 9:     ctext ← BlowfishEncrypt(s, ctext)
10: end for
11: return T || sal || ctext
12: function ExpandKey(s, sal, senha)
13:     for i ← 1 to 32 do
14:         P_i ← P_i ⊕ senha[32(i − 1) . . . 32i − 1]
15:     end for
16:     for i ← 1 to 9 do
17:         temp ← BlowfishEncrypt(s, sal[64(i − 1) . . . 64i − 1])
18:         P_{0+2(i−1)} ← temp[0 . . . 31]
19:         P_{1+2(i−1)} ← temp[32 . . . 64]
20:     end for
21:     for i ← 1 to 4 do
22:         for j ← 1 to 128 do
23:             temp ← BlowfishEncrypt(s, sal[64(j − 1) . . . 64j − 1])
24:             S_i[2(j − 1)] ← temp[0 . . . 31]
25:             S_i[1 + 2(j − 1)] ← temp[32 . . . 63]
26:         end for
27:     end for
28:     return s
29: end function
```

utilizando a função chamada $EksBlowfishSetup$, e então utilizar o algoritmo EksBlowfish para cifrar iterativamente, por 64 vezes, uma frase constante.

Como pode ser observado no Algoritmo 2, a função $EksBlowfishSetup$ copia os primeiros dígitos do número π nas sub-chaves e *S-boxes* S_i. Então, a função $EksBlowfishSetup$ atualiza os valores das sub-chaves e *S-boxes* invocando a função $ExpandKey(sal, senha)$, com um valor de sal de 128 bits. Primeiramente, a função $ExpandKey$ realiza operações de XOR entre a senha e as sub-chaves, e então, iterativamente, cifra uma das metades do sal, e realiza uma operação de XOR entre o resultado cifrado e a outra metade do sal, substituindo as próximas sub-chaves e *S-boxes*. Depois que todas as sub--chaves e *S-boxes* forem atualizadas, o algoritmo invoca $ExpandKey(0, sal)$ e $ExpandKey(0, senha)$, alternadamente, por 2^T iterações. O parâmetro T, defi-

nido pelo usuário, determina então, o tempo gasto no processo de atualização das sub-chaves e das *S-boxes*, controlando efetivamente o tempo total de execução do bcrypt.

Como no PBKDF2, o bcrypt permite que o usuário parametrize apenas seu tempo total de execução. Além desta deficiência, algumas de suas características podem ser consideradas pequenas desvantagens quando comparadas ao PBKDF2. Primeiramente, o bcrypt utiliza uma estrutura dedicada, ao invés de uma função de *hash* convencional, levando à necessidade de implementar uma nova primitiva criptográfica, aumentando o tamanho do código do algoritmo. A quantidade de execuções do laço interno da função $EksBlowfishSetup$ cresce de forma exponencial, fazendo com que o ajuste fino do tempo de execução do algoritmo precise ser realizado de forma empírica, impactando ao usuário legítimo. Por fim, o bcrypt possui uma restrição incomum, não permitindo que usuários escolham senhas com comprimento maior que 56 bytes.

3.3 Scrypt

O projeto do scrypt é focado em acoplar os custos de memória e tempo de processamento (PERCIVAL, 2009). Para este fim, o scrypt emprega funções sequenciais que, assintoticamente, utilizam quase a mesma quantidade de memória quanto operações, para as quais uma implementação paralela não consegue obter um custo significativamente mais baixo. Como consequência, se o número de operações e a quantidade de memória utilizada forem ambos $O(R)$, a complexidade de um ataque com pouca memória (i.e., um ataque no qual a quantidade de memória seja reduzida para $O(1)$), se torna $\Omega(R^2)$, onde R é um parâmetro do sistema. Percival (2009) fornece uma definição mais formal deste conceito.

Algorithm 3 Scrypt.

Param: h ▹ Comprimento da saída da função de*hash* interna de *BlockMix*
Input: $senha$ ▹ A senha
Input: sal ▹ O sal
Input: k ▹ O tamanho da chave
Input: b ▹ O tamanho do bloco, satisfazendo $b = 2r \cdot h$
Input: R ▹ Parâmetro de custo (uso de memória e tempo de processamento)
Input: p ▹ Parâmetro de paralelismo
Output: K ▹ A chave derivada da senha

1: $(B_0...B_{p-1}) \leftarrow$PBKDF2$_{HMAC-SHA-256}(senha, sal, 1, p \cdot b)$
2: **for** $i \leftarrow 0$ **to** $p-1$ **do**
3: $B_i \leftarrow$ROMix(B_i, R)
4: **end for**
5: $K \leftarrow$PBKDF2$_{HMAC-SHA-256}(senha, B_0||B_1||...||B_{p-1}, 1, k)$
6: **return** K ▹ Retorna a chave, de comprimento k

7: **function** ROMix(B, R)
8: $X \leftarrow B$
9: **for** $i \leftarrow 0$ **to** $R-1$ **do** ▹ Inicializa o vetor de memória V
10: $V_i \leftarrow X$; $X \leftarrow$BlockMix(X)
11: **end for**
12: **for** $i \leftarrow 0$ **to** $R-1$ **do** ▹ Lê possições aleatórias de V
13: $j \leftarrow Integerify(X) \bmod R$
14: $X \leftarrow$BlockMix$(X \oplus V_j)$
15: **end for**
16: **return** X
17: **end function**

18: **function** BlockMix(B) ▹ função de *hash* com comprimento de saída b
19: $Z \leftarrow B_{2r-1}$ ▹ $r = b/2h$, onde $h = 512$ para Salsa20/8
20: **for** $i \leftarrow 0$ **to** $2r-1$ **do**
21: $Z \leftarrow Hash(Z \oplus B_i)$; $Y_i \leftarrow Z$
22: **end for**
23: **return** $(Y_0, Y_2, ..., Y_{2r-2}, Y_1, Y_3, Y_{2r-1})$
24: **end function**

Conforme mostrado no Algoritmo 3, primeiramente são inicializados p blocos de memória B_i, cada um com comprimento b. Esta operação é realizada utilizando o algoritmo PBKDF2, com HMAC-SHA-256 (NIST, 2002), como função de *hash* subjacente e uma única iteração. Então, cada bloco B_i é processado (incrementalmente ou em paralelo) pela função *ROMix*. Basicamente, a função *ROMix* inicializa um vetor V de R elementos, cada um com comprimento b, aplicando a função de *hash* sobre B_i. Então, R posições de V são visitadas aleatoriamente, atualizando a variável de estado interno X, durante este processo estritamente sequencial, verificando que estas posições estão, de fato, disponíveis na memória. A função de *hash* empregada pela *ROMix* é chamada de *BlockMix*, e se comporta como uma função que tenha tamanhos de entrada e saída arbitrários, utilizando, internamente, a cifra de fluxo

Salsa20/8 (BERNSTEIN, 2008), cujo tamanho de saída é $h = 512$. Após o final dos p processos de *ROMix*, os blocos B_i são utilizados como sal em uma iteração final do PBKDF2, resultando na chave K.

O scrypt apresenta um projeto muito interessante, sendo uma das poucas soluções existentes que permite a configuração dos custos de processamento e memória. Uma de suas principais deficiências é o forte acoplamento entre o custo de memória e tempo de processamento, especificamente, o projeto do scrypt impede que o usuário aumente o custo de processamento, mantendo a quantidade de memória fixa, a não ser que o usuário esteja disposto a elevar o valor do parâmetro p e permitir que o usuário explore o paralelismo. Uma inconveniência do scrypt se dá ao fato do algoritmo utilizar duas funções de *hash* subjacentes, HMAC-SHA-256 para o PBKDF2 e Salsa20/8 no núcleo da função *BlockMix*, elevando a complexidade de implementação. Por fim, apesar das vulnerabilidades conhecidas da função Salsa20/8 (AUMASSON *et al.*, 2008) não colocarem a segurança do scrypt em risco, é aconselhável utilizar uma alternativa mais forte, especialmente, considerando que a estrutura interna do algoritmo não impõe muitas restrições na função que será utilizada por *BlockMix*. Neste caso, uma função esponja poderia ser uma alternativa. Porém, as propriedades intrínsecas das funções esponjas tornam algumas das operações do scrypt desnecessárias, pois, como as funções esponja são compatíveis com entradas e saídas de qualquer tamanho, seria possível substituir toda a estrutura *BlockMix*. Além, disto, esponjas podem operar de maneira *stateful*, tornando a variável de estado X redundante.

Inspirado no projeto do scrypt, o Lyra foi construído sobre as propriedades das funções esponja, a fim de permitir uma solução mais simples e segura. De fato, o custo de processamento de ataques, envolvendo menos memória que o especificado pelo algoritmo, cresce muito mais que quadraticamente no

Lyra, sendo melhor que o obtido pelo scrypt e evitando que trocas de memória por tempo sejam úteis. Esta característica desencoraja o atacante a trocar uso de memória por tempo de processamento, que é exatamente o objetivo de funções de derivação de chaves, que permitem a configuração de ambos parâmetros. Adicionalmente, o Lyra permite uma utilização maior de memória para um tempo de processamento similar, quando comparado ao scrypt, aumentando o custo de possíveis ataques.

3.4 PHC

Esta seção traz um curto resumo dos principais concorrentes inscritos no PHC.

3.4.1 Catena

O Catena (FORLER; LUCKS; WENZEL, 2013; FORLER; LUCKS; WENZEL, 2014) apresenta um projeto simples, elegante, e de fácil compreensão. Sua segurança pode ser verificada de maneira simples, pois sua estrutura é baseada num tipo especial de grafo, chamado de reversão de bits (LENGAUER; TARJAN, 1982). Este tipo de grafo permite que a segurança do Catena seja demonstrada através de um "jogo de pedras" ("*pebbling game*"), possibilitando uma definição acurada da troca do custo de memória, por custo de tempo (COOK, 1973; DWORK; NAOR; WEE, 2005). Além disto, com este tipo de grafo, a ordem de visitação da memória é definida, evitando ataques de temporização de cache. Recentemente, foi descoberta uma falha na demonstração de segurança do Catena, que mostrou que a penalidade para a troca de memória por tempo é, de fato, menor do que o que foi demonstrado originalmente (BIRYUKOV; KHOVRATOVICH; GROZSCHAEDL, 2014).

Apesar de utilizar quantidades significativas de memória e processamento para gerar suas chaves, o Catena não permite um ajuste fino de seus parâmetros. Isso ocorre, porque o número de operações que serão realizadas é definido pela quantidade de memória utilizada, chamada de *garlic*. Para poder ajustar o tempo, usuários podem alterar a profundidade do grafo de reversão de bits, fazendo mais operações, com a mesma quantidade de memória. Porém, isso tem um grande impacto no tempo de execução em plataformas legítimas.

O Catena também utiliza duas ideias interessantes no seu projeto. A primeira delas é o "protocolo de alívio de servidor". Este permite que, parte do esforço necessário para computar as chaves seja transferido do servidor para o cliente, economizando recursos e tempo de processamento nos servidores. A outra ideia interessante é uma "atualização independente de cliente", que permite que o usuário legítimo aumente o principal parâmetro de segurança a qualquer momento, inclusive para contas inativas, aumentando seu nível de segurança.

3.4.2 Yescrypt

De forma semelhante ao scrypt, o yescrypt (PESLYAK, 2015) é um PHS focado no ajuste de tempo e memória pelo usuário. Por isso, além dos parâmetros comuns (quantidade de memória e tempo de processamento), permite que o usuário configure algumas opções adicionais, por exemplo: indicar se o usuário irá realizar operações em modo somente leitura, somente escrita ou ambos (PESLYAK, 2015). Além disto, também permite que o usuário utilize memória ROM no processo de geração de chaves.

Apesar de possuir um número elevado de parâmetros, o yescrypt possui algumas desvantagens no ajuste fino de sua execução, pois exige que a quan-

tidade de memória seja uma potência de dois, não permitindo que o usuário escolha uma quantidade arbitrária de memória. Além disso, o usuário não consegue realizar um ajuste fino do tempo de processamento, pois, conforme o parâmetro de tempo é elevado, o yescrypt apresenta uma grande perda de eficiência. Sendo assim, para ajustar o tempo de execução, o usuário legítimo deve ajustar os outros parâmetros de configuração e testar a execução, verificando se o tempo atingiu o valor desejado (PESLYAK, 2015).

3.4.3 Argon2

Bem como o Lyra, o Argon2 (BIRYUKOV; DINU; KHOVRATOVICH, 2015) foi criado visando segurança contra ataques que substituam o uso de memória pelo uso de processamento. Desta forma, o algoritmo penaliza principalmente os atacantes que desejam utilizar menos memória durante a computação da chave.

Este algoritmo utiliza internamente a função Blake2, da mesma forma que o Lyra. A primeira versão do Argon tinha um desempenho inferior, quando comparado aos demais candidatos, porém na segunda versão houve uma melhoria no desempenho, tornando-o mais competitivo.

O Argon2 foi eleito o vencedor do concurso *Password Hashing Competition*.

4 LYRA

Como qualquer PHS, o Lyra recebe como entrada um sal e uma senha, criando uma saída pseudoaleatória, que pode então ser utilizada como chave para algoritmos de criptografia (NIST, 2009). Internamente, a memória é organizada como uma matriz, que é acessada repetidamente, durante todo o processo de derivação da chave. Esta matriz é acessada iterativamente, de acordo com a parametrização, feita pelo usuário, permitindo que o tempo de execução do Lyra seja configurado, de acordo com os recursos disponíveis na plataforma alvo. A construção e visitação da matriz de memória são realizadas através da combinação das operações de *absorb*, *squeeze* e *duplexing* da esponja subjacente, mantendo-se o estado da esponja entre as operações. Ou seja, o estado da esponja não é zerado durante a operação *absorb*.

4.1 Estrutura

O algoritmo 4 detalha a execução do Lyra, que é divida em três fases: *Setup*, *Wandering* e *Wrap-up*.

4.1.1 *Setup*

A primeira parte do algoritmo é a fase de *Setup* (linhas 1 – 8), em que a matriz de memória, com tamanho $R \times C$ é construída, com cada célula tendo

Algorithm 4 O algoritmo Lyra.

```
Param: Hash   ▷ Esponja com tamanho de bloco b (em bits) e permutação f
Param: ρ   ▷ Número de rodadas de f nas fases de Setup e Wandering
Param: W   ▷ O tamanho da palavra na máquina alvo (geralemte 32 ou 64)
Input: senha   ▷ A senha
Input: sal   ▷ Um sal aleatório
Input: T   ▷ Custo de tempo, em número de iterações
Input: R   ▷ Número de linhas na matriz de memória
Input: C   ▷ Número de colunas na matriz de memória
Input: k   ▷ O comprimento da chave, em bits
Output: K   ▷ A chave derivada da senha, com k bits de comprimento

 1: ▷ Setup: Inicializa a matriz de memória, com (R × C), com cada célula tendo b bits
 2: Hash.absorb(pad(senha || sal || basil))   ▷ Regra de preenchimento: 10*1
 3: M[0] ← Hash.squeeze_ρ(C · b)
 4: for row ← 1 to R − 1 do
 5:     for col ← 0 to C − 1 do
 6:         M[row][col] ← Hash.duplexing_ρ(M[row − 1][col], b)
 7:     end for
 8: end for
 9: ▷ Wandering: Sobrescreve blocos da matriz de memória iterativamente
10: row ← 0
11: for i ← 0 to T − 1 do   ▷ Laço de Tempo
12:     for j ← 0 to R − 1 do   ▷ Laço de linhas: visita R linhas, aleatoriamente
13:         for col ← 0 to C − 1 do   ▷ Laço de colunas: visita os blocos nas linhas
14:             M[row][col] ← M[row][col] ⊕ Hash.duplexing_ρ(M[row][col], b)
15:         end for
16:         col ← trunc(M[row][C − 1], W) mod C
17:         row ← Hash.duplexing(M[row][col], W) mod R
18:     end for
19: end for
20: ▷ Wrap-up: computação da chave
21: Hash.absorb(pad(sal))   ▷ Utiliza o estado atual da esponja
22: K ← Hash.squeeze(k)
23: return K   ▷ Retorna a chave com k bits
```

b blocos. Os parâmetros R e C são definidos pelo usuário e b é o taxa de bits da esponja subjacente.

Esta fase inicia-se quando a esponja absorve a senha, o sal e o *basil* devidamente preenchidos, gerando um estado interno dependente destes (linha 2). A regra de preenchimento adotada pelo Lyra é o `pad10*1`, descrito em (BERTONI *et al.*, 2011a), sendo denotado simplesmente por pad. Esta regra consiste em concatenar à entrada um único bit 1, seguido por quantos bits 0 forem necessários, seguido por um único bit 1 final, de forma que pelo menos dois bits 1 sejam concatenados.

Na primeira operação de *absorb*, o objetivo do *basil* é, basicamente, evitar colisões, para combinações triviais de senhas e sais. Por exemplo, para cada

$(u, v \mid u + v = \alpha)$, haveria uma colisão se $senha = \mathbb{0}^u$, $sal = \mathbb{0}^v$ e o *basil* fosse uma cadeia de bits vazia. Porém, isto não ocorreria se o *basil* incluísse explicitamente u e v. Portanto, o *basil* pode ser visto como uma extensão do sal, podendo incluir uma variedade de informações, como por exemplo: a lista de parâmetros de entrada, uma identificação do usuário, um nome de domínio, entre outras. Cabe notar, entretanto, que em sistemas em que o sal tenha comprimento fixo, o uso de um *basil* pode ser considerado opcional.

Durante o preenchimento da matriz é utilizado o conceito de número de rodadas reduzidas, que foi apresentado na família de algoritmos de autenticação de mensagens Alred (DAEMEN; RIJMEN, 2005; DAEMEN; RIJMEN, 2010; SIMPLICIO JR *et al.*, 2009; SIMPLICIO JR; BARRETO, 2012). No Lyra, as operações *duplexing* e *squeeze* são executadas com uma versão de f com número de rodadas reduzido, denotada por f_ρ, indicando que são executadas ρ rodadas, ao invés do número de rodadas normal ρ_{max}. Por exemplo, na implementação de referência do Lyra, são adotados os valores $\rho = 1$ e $\rho_{max} = 12$, indicando que, durante a fase *Setup*, as operações de *duplexing* e *squeeze* serão executadas com apenas uma rodada interna da função f, ao invés das 12 rodadas que caracterizam a execução normal de f. Esta abordagem acelera as operações *duplexing* e *squeeze*, permitindo que, em um mesmo período de tempo, um maior número de posições de memória seja percorrido em comparação à execução de f com rodadas completas. A primeira linha da matriz de memória é preenchida através de uma única operação *squeeze* reduzida $Hash.squeeze_\rho$ (linha 3). Sem que o estado da esponja seja reiniciado, a operação *duplexing* reduzida $Hash.duplexing_\rho$ é chamada, repetidamente, até que todas as linhas da matriz de memória sejam preenchidas (linha 6). Após o preenchimento completo da matriz de memória, o estado interno da esponja subjacente não é reinicializado para zero, sendo mantido para utilização nas

próximas fases.

4.1.2 Wandering

A fase que mais consome recursos, *Wandering* (linhas 10 – 19), é inicializada após o término da fase *Setup*. Um total de $(T \cdot R)$ linhas da matriz de memória são visitadas iterativamente, R linhas por iteração do laço iniciado na linha 11, chamado de *Laço de Tempo*. As linhas visitadas tem todas as suas células lidas e combinadas com a saída da função $Hash.duplexing_{\rho}$ (linha 14).

A ordem de visitação da fase *Wandering* é determinada pela variável interna row, que é inicializada com zero na linha 10. Sendo assim, a primeira linha $M[0]$ é sempre visitada primeiro. O restante das linhas é visitada de uma maneira pseudoaleatória, visto que o valor de row é atualizado após cada visita (linha 17). Esta atualização é feita através de uma operação *duplexing* completa, de uma das células da linha visitada mais recentemente, resultando em um valor pseudoaleatório de row, tornando a ordem de visitação dependente de todo o processamento realizado até o momento. O índice da célula é escolhido conforme $trunc(M[row][C-1], W) \bmod C$, de forma que só seja possível determiná-lo após o processamento da última célula da linha correspondente. Este processo tem por objetivo exigir que a matriz de memória esteja disponível durante toda a derivação da chave.

Idealmente, o valor do parâmetro Cdeve ser escolhido de forma que o tamanho de uma linha da matriz seja o mesmo do *cache* da máquina, diminuindo os custos das operações de leitura e escrita em cada célula. Com o valor de Cdefinido, ajusta-se o valor de Raté que a matriz consuma a quantidade de memória desejada pelo usuário.

4.1.3 Wrap-up

Durante a fase *Wrap-up* (linhas 21 – 22), a chave é computada, aplicando-se a operação de *absorb* no sal, seguida da operação de *squeeze* com número de rodadas completo, utilizando o estado atual da esponja. Assim, o número de bits gerados é tão arbitrário quanto permitido pela esponja subjacente. A operação de *squeeze stateful* e com número de rodadas completo, aplicada neste último estágio, garante que o processo todo seja não inversível e de natureza sequencial.

4.2 Projeto estritamente sequencial

Bem como o PBKDF2 e outros PHSs existentes, o projeto do Lyra é estritamente sequencial, visto que o estado interno da esponja é atualizado iterativamente, sempre que uma célula da matriz de memória é processada. Assumindo que o estado da esponja, após processar a célula $c_i = M[row][col + i]$, seja s_i. Após o processamento de c_i, que resulta em c'_i, o estado atualizado é s_{i+1}. Supondo que o atacante queira paralelizar o laço de colunas (linhas 13 – 15), processando $\{c_0, c_1, c_2\}$ de forma mais rápida que computar $c'_0 = s_0$, $c'_1 = s_0 \oplus c_0$, $c'_2 = c'_1 \oplus c_1$, $s_3 = c'_2 \oplus c_2$ sequencialmente. Se a transformação f fosse afim, esta tarefa seria facilmente realizada. Por exemplo, se f fosse a função identidade, o atacante poderia utilizar quatro núcleos de processamento de uma GPU para fazer $x = s_0 \oplus c_0$, $y = c_0 \oplus c_1$, $z = c_1 \oplus c_2$ em paralelo e então, calcular $c'_0 = s_0$, $c'_1 = x$, $c'_2 = x \oplus c_1$, $s_3 = x \oplus z$. Utilizando uma FPGA com ligações adequadas, esta tarefa poderia ser realizada ainda mais rapidamente, em um único passo.

Contudo, para uma transformação altamente não linear f_ρ, é difícil decompor duas operações *duplexing* iterativas $f_\rho(f_\rho(s_0 \oplus c_0) \oplus c_1)$ em uma forma

paralelizável eficiente. Em analogia, o custo de paralelizar diversas iterações de f_ρ cresce com o número de iterações.

É interessante notar que, se f_ρ tiver um comportamento cíclico óbvio, ou seja, a esponja é reiniciada com um estado conhecido s_0, depois de v visitações, o atacante pode paralelizar a visitação de c_i e c_{i+v}. Entretanto, qualquer f_ρ razoavelmente segura previne tal comportamento cíclico por projeto, uma vez que esta propriedade poderia ser explorada, a fim encontrar colisões internas contra a própria f. Resumindo, mesmo que um atacante seja capaz de paralelizar algumas partes internas de f_ρ, a natureza *stateful* da fase *Wandering* cria diversos gargalos compostos de operações seriais, prevenindo que células sejam visitadas em paralelo.

Considerando que os ataques estruturais, mencionados acima sejam impraticáveis, ainda é possível conseguir o paralelismo através de força bruta. Ou seja, o atacante pode criar duas instâncias de esponjas, I_0 e I_1, e inicializar seus estados internos para s_0 e s_1, respectivamente. Se s_0 é conhecido, tudo o que o atacante precisa fazer é computar s_1 mais rapidamente do que processar c_0 com I_0 efetivamente. Por exemplo, o atacante poderia utilizar uma grande tabela de mapeamento de estados e blocos de entrada para estados resultantes, e então, utilizar a entrada da tabela $(s_0, c_0) \mapsto s_1$. No entanto, para qualquer esponja criptográfica razoável, é esperado que o tamanho do estado e do bloco sejam consideravelmente grandes, como 512 ou 1.024 bits, por exemplo. Isto significa que a quantidade de memória necessária, para construir uma tabela de mapeamento completa, torna esta abordagem impraticável.

De forma alternativa, o atacante pode simplesmente inicializar várias instâncias de I_1, com os valores supostos de s_1, e aplicá-los a c_1, em paralelo. Quando I_0 termina sua execução e o valor de s_1 é, inevitavelmente, deter-

minado, o atacante pode compará-lo ao valor suposto, mantendo apenas os resultados obtidos com a instância correta.

A princípio, pode parecer que uma f com número de rodadas reduzido pode facilitar esta tarefa, visto que os estados consecutivos s_0 e s_1 podem compartilhar alguns bits ou relações entre bits, reduzindo o número de possibilidades que precisam ser incluídas entre os estados supostos. Aceitando-se a hipótese anterior como verdadeira, e considerando-se que qualquer transformação f tenha uma relação complexa entre entrada e saída, para acelerar a visitação de uma célula, o atacante deve conseguir explorar esta relação de forma mais rápida que processar ρ rodadas de f. Caso contrário, o processo de determinar o espaço de suposição desejado será mais lento que, simplesmente, processar as células sequencialmente. Além disto, para adivinhar o estado que será alcançado após visitar v células, o atacante teria que explorar as relações entre aproximadamente $v \cdot \rho$ rodadas de f, mais rápido que, simplesmente, executar $v \cdot \rho$ rodadas de f. Por isso, mesmo que seja possível adivinhar dois estados consecutivos, de forma mais rápida do que executar ρ de f, esta estratégia não é escalável, visto que a relação entre os bits é diluída conforme $v \cdot \rho$ se aproxima de ρ_{max}.

Uma análise semelhante pode ser aplicada aos laços de tempo e linhas. A diferença em relação ao que já foi explicado e o laço de linhas é que, para determinar qual linha será visitada a seguir e iniciar seu processamento em paralelo, o atacante precisa encontrar o estado interno que resulta da visitação da linha atual, sem ter, de fato, visitado-a, o que envolve $C \cdot \rho$ rodadas de f. Para o laço de tempo, determinar o estado depende do resultado da visitação de várias linhas de forma aleatória, o que envolve $C \cdot R \cdot \rho$ rodadas de f.

Portanto, mesmo que o atacante tenha a seu dispor, equipamentos altamente paralelizáveis, é improvável que ele consiga utilizar todo este potencial

de paralelismo para acelerar a operação de uma instância do Lyra.

4.3 Configurando a quantidade de memória e tempo de processamento

A quantidade de memória ocupada pela matriz do Lyra é dada por $m = b \cdot R \cdot C$. O valor de b corresponde à taxa de bits da função esponja subjacente. Sendo assim, não há a necessidade de preencher os blocos de entrada, quando eles forem processados pela construção duplex, o que leva a uma implementação mais simples. Os parâmetros R e C, por sua vez, são definidos pelo usuário, permitindo a configuração da quantidade de memória necessária durante a operação do algoritmo. O parâmetro C deve ser escolhido de forma que cada linha da matriz tenha o tamanho do *cache* do processador, acelerando as operações de leitura e escrita da matriz.

Ignorando as operações auxiliares, o custo de processamento do Lyra é determinado, basicamente, pelo número de chamados à função f da esponja subjacente. Por isso, considerando todas as fases do algoritmo e supondo que ambos $(|senha| + |sal| + |basil|)$ e k sejam menores do que b, o custo total de processamento do Lyra é aproximadamente: $1 + (R - 1) \cdot C \cdot \rho/\rho_{max}$ para a fase *Setup*, mais $T \cdot R \cdot (1 + C \cdot \rho/\rho_{max})$ para a fase *Wandering*, mais 2 para a fase *Wrap-up*, resultando em, aproximadamente, $(T + 1) \cdot R \cdot C \cdot \rho/\rho_{max}$ chamadas à f. O custo de memória impõe o limite inferior do custo de processamento, porém, é possível ajustar este tempo linearmente, sem afetar a quantidade de memória, através da escolha de um valor adequado para o parâmetro T. Portanto, os usuários podem utilizar o recurso mais abundante a seu dispor, mesmo que a quantidades de memória e poder de processamento estejam desbalanceados. Isto permite que o Lyra utilize mais memória que o scrypt para um tempo de processamento semelhante, pois, enquanto o scrypt utiliza

hashes com rodadas completas, o Lyra utiliza *hashes* com rodadas reduzidas.

4.4 Função esponja subjacente

O Lyra é compatível com qualquer função da família esponja, porém, o Keccak (BERTONI *et al.*, 2011b), não é a melhor alternativa para este fim, pois este se destaca pelo desempenho em *hardware* e não em *software*. Portanto, para a aplicação específica de derivação de chave baseada em senha, o Keccak fornece maiores vantagens para atacantes munidos de *hardwares* personalizados, quando comparado aos usuários legítimos, que utilizam implementações em *software*.

A recomendação é utilizar um algoritmo seguro, voltado à implementação em *software*, com baixo paralelismo como a transformação f da esponja. Um exemplo é a função de compressão do Blake2b (AUMASSON *et al.*, 2013), que é considerada uma boa permutação (AUMASSON *et al.*, 2010; MING; QIANG; ZENG, 2010), fornecendo um nível de segurança similar ao encontrado no Keccak (CHANG *et al.*, 2012).

4.5 Considerações Práticas

O Lyra apresenta uma estrutura simples, sendo construído com base nas propriedades intrínsecas das funções esponja, operando em um modo completamente *stateful*, o estado da esponja é inicializado na fase *Setup* e mantido por toda a execução do algoritmo, até a geração da chave ao final da fase *Wandering*. De fato, todo o algoritmo é composto, basicamente, por controles de laços e inicialização de variáveis, enquanto todo o processamento de dados é realizado pela função de *hash* subjacente. Portanto, o Lyra é facilmente implementado em *software*, especialmente se uma função esponja já estiver

disponível.

A matriz de memória do Lyra foi projetada de forma a permitir que o usuário legítimo aproveite as funcionalidades de hierarquia de memória, como *caching* e *prefetching*. Tais mecanismos, geralmente, fazem com que acessos aos locais de memória próximos sejam muito mais rápidos quando comparados aos acessos à posições aleatórias, mesmo para chips de memória classificados como "acesso aleatório" (PERCIVAL, 2009). Como resultado, uma matriz de memória com $R = 1$ deve ser visitada mais rapidamente que uma matriz com $C = 1$, para valores idênticos de $R \cdot C$.

Portanto, ao escolher os valores adequados de R e C, o Lyra pode ser otimizado para ser executado mais rapidamente em plataformas legítimas, ainda impondo penalidades a atacantes que utilizem diferentes condições de acesso à memória. Por exemplo, ao configurar $b \cdot C$ para ser, aproximadamente, do tamanho do cache da plataforma alvo, a latência de memória pode ser reduzida significativamente, permitindo que o valor de T seja aumentado, sem impactar no desempenho nesta plataforma específica.

Outra consideração prática abordada no Lyra, diz respeito ao tempo em que senha original, fornecida pelo usuário, deve permanecer na memória. Isto é tratado através da sobrescrita da posição de memória que armazena a senha, logo na primeira operação *absorb* (linha 2 do algoritmo 4). Desta forma, evita-se que uma implementação descuidada mantenha a senha na memória volátil ou permita que a senha seja armazenada em memória não volátil, devido a *swap* de memória durante a operação do algoritmo. Com isto, o Lyra está de acordo com as regras de expurgar informações privadas da memória o mais rápido possível, assim que esta informação não for mais necessária, impedindo que esta seja recuperada, caso o dispositivo seja roubado (HALDERMAN *et al.*, 2009; YUILL; DENNING; FEER, 2006).

O Lyra possui melhor desempenho quando comparado com o scrypt, sendo, mais amigável para o usuário legítimo, pois este deseja que a operação de armazenar sua senha seja o mais rápida possível. Além disto, também é possível utilizar o Lyra para aliviar a carga do servidor, fazendo com que o *hash* seja computado no computador do cliente. O servidor fornece o valor do sal para o cliente, que calcula o valor do *hash* e retorna para o servidor, que compara com o valor armazenado. Desta forma, a maior parte do processamento é feito localmente, no computador do cliente, diminuindo a sobrecarga no servidor.

4.6 Paralelismo em plataformas legítimas

Apesar de um PHS estritamente sequencial ser interessante para prevenir ataques, esta pode não ser a melhor escolha, caso o usuário legítimo possua uma plataforma equipada com múltiplos núcleos de processamento, como uma CPU ou GPU. Neste caso, o usuário legítimo pode querer aproveitar sua capacidade de paralelismo para aumentar o uso de memória do PHS, mantendo o tempo de processamento dentro de limites humanamente aceitáveis.

Na defesa contra um atacante, executando diversas instâncias em paralelo, esta estratégia aumenta, instantaneamente, o custo de memória, proporcionalmente ao número de núcleos utilizados pelo usuário legítimo. Por exemplo, se a chave for computada por um PHS estritamente serial, que utilize 10 MB de memória e leve 1 segundo para ser executado em um único núcleo, um atacante que tiver acesso a 1.000 núcleos de processamento e 10 GB de memória poderia realizar 1.000 testes de senha por segundo, sendo um por núcleo. Entretanto, se a chave for computada por um PHS sendo executado em dois núcleos, uma tentativa de descobrir a senha exigiria 20 MB e 1 segundo, obrigando o atacante a possuir 20 GB de memória para obter os

mesmos 1.000 testes por segundo.

Visando permitir ao usuário legítimo explorar suas capacidades de paralelismo, foi proposta uma versão alterada do Lyra, chamada $Lyra_p$, onde o parâmetro $p \geqslant 1$ é o grau de paralelismo desejado. A operação do $Lyra_p$ é descrita a seguir.

Na fase de *Setup*, são geradas p cópias da esponja, cada um responsável por inicializar e processar uma matriz $R \times C$. Esta operação é realizada de forma semelhante ao Lyra. A diferença é que cada esponja i ($0 \leqslant i \leqslant p-1$), após ser inicializada na linha 2 do algoritmo 4, deve realizar $p-1$ operações de *absorb* de forma *stateful* e com rodadas completas, em um bloco adicional de valor $\mathrm{pad}(i)$, sendo que i é representado por um valor de $|p-1|$ bits.

Por exemplo, para $p = 2$, a primeira esponja irá realizar uma operação de *absorb* sobre o bit 0, propriamente preenchido, enquanto a segunda esponja irá fazer a mesma operação para o bit 1. Para $p = 4$, a operação de *absorb* será aplicada três vezes em cada esponja, com os valores 0, 1, 2 e 3, representados com dois bits. Esta abordagem garante que cada uma das p esponjas seja inicializada com um estado interno distinto, mesmo realizando *absorb* com valores idênticos de sal e senha. Além disso, como o número de operações *absorb* realizadas depende de p, a execução de $Lyra_{p'}$, com $p' \neq p$ não pode ser reutilizada para ataques a $Lyra_p$, dificultando o ataque, caso o valor de p não seja conhecido.

O restante da fase de *Setup* (linhas 4 – 8 do algoritmo 4), prossegue, da mesma forma para cada uma das p esponjas, bem como as fases *Wandering* e *Wrap-up*. A operação destas esponjas podem ser totalmente paralelizadas, com cada *thread* utilizando aproximadamente a mesma quantidade de memória para o tempo de processamento desejado. Após o término do processamento de todas as esponjas, as p sub-chaves geradas são unidas, através de

operações XOR, resultando na chave K.

É importante ressaltar que, para $p = 1$, o $Lyra_p$ se comporta exatamente da mesma forma que o Lyra, fazendo com que ambos algoritmos sejam completamente compatíveis. Portanto, o $Lyra$ pode ser visto como uma abreviação de $Lyra_1$.

4.7 Análise de segurança

O projeto do Lyra foi desenvolvido de forma que: a chave derivada seja não inversível, devido às operações de *hash* inicial e final, sobre a senha e o sal; atacantes não são capazes de paralelizar o algoritmo 4, usando múltiplas instâncias da esponja criptográfica $Hash$, sendo incapazes de acelerar o processo de testar uma senha, através de múltiplos núcleos de processamento; uma vez iniciada, a matriz de memória deve permanecer disponível durante a maior parte do processo de derivação da chave, o que significa que a operação ideal do Lyra requer memória (rápida) suficiente para armazenar seu conteúdo.

Para obter melhor desempenho, um usuário legítimo, provavelmente, armazene a matriz inteira em memória volátil, facilitando o acesso a ela em cada uma das muitas iterações das fases *Wandering* e *Wrap-up*. Um atacante que esteja executando diversas instâncias do Lyra, por sua vez, pode decidir armazenar apenas uma parte menor da matriz na memória rápida, visando reduzir o custo de memória para cada tentativa de descobrir a senha. Mesmo que esta alternativa diminua a quantidade de senhas testadas por segundo, para cada instância do Lyra, o objetivo desta estratégia é permitir que um número maior de senhas seja testado em paralelo, aumentando a quantidade de senhas testadas por segundo do processo como um todo.

Existem, basicamente, dois métodos para se conseguir isto. O primeiro é trocar o custo de memória pelo custo de tempo, armazenando apenas o estado interno da esponja, após o processamento de cada linha e recalculando a próxima linha a ser visitada do início, quando necessário. Este ataque é chamado *Ataque com pouca memória*. O segundo consiste em utilizar memória de baixo custo, porém mais lenta, como discos magnéticos, chamado de *Ataque com memória lenta*.

A seguir, ambos ataques serão apresentados e avaliados, mostrando as desvantagens de cada uma destas abordagens alternativas. O objetivo desta discussão é demonstrar que o projeto do Lyra desencoraja os atacantes a realizar estas trocas de memória por tempo, enquanto testam muitas senhas em paralelo. Com isto, os atacantes provavelmente devem pagar o custo de memória, como parametrizado pelo usuário legítimo, limitando sua habilidade de tirar proveito de plataformas altamente paralelizáveis, como GPUs e FPGAs, para descobrir a senha.

4.7.1 Ataque com pouca memória

Antes de discutir ataques com pouca memória contra o Lyra, deve-se analisar como estes ataques podem ser realizados contra a estrutura *ROMix* (algoritmo 3), visto que seu projeto foi concebido, principalmente, para prover proteção contra este tipo de ataque. De acordo com o projeto do scrypt, é possível formular o Teorema 1 abaixo:

Teorema 1. *Enquanto o custo de memória e processamento do scrypt são ambos $O(R)$ para o parâmetro de sistema R, pode-se obter um custo de memória de $O(1)$ (um ataque com pouca memória), aumentando o custo de processamento para $O(R^2)$.*

Demonstração. O atacante executa o laço para inicializar o vetor de memória V (linhas 9 – 11), que é chamada *ROMix*$_{ini}$. Entretanto, ao invés de armazenar os valores de V_i, o atacante mantém apenas o valor da variável interna X. Então, sempre que um elemento V_j de V precisar ser lido (linha 14 do algoritmo 3), o atacante executa *ROMix*$_{ini}$ por j iterações, determinando o valor de V_j e atualizando X. Ignorando as operações auxilares, o custo médio deste ataque é $R + (R \cdot R)/2$ execuções iterativas de *BlockMix* e o armazenamento de uma única variável, de comprimento b, sendo R o parâmetro de custo do scrypt. □

Em comparação, um atacante tentando utilizar um ataque similar contra o Lyra deve prosseguir da seguinte forma. Primeiramente, o atacante executa a fase *Setup* e armazena apenas o estado interno da esponja resultante. Como row é configurado para 0, no início da fase *Wandering*, a primeira linha é sempre visitada primeiro. Portanto, apenas os valores do sal e da senha são necessários na primeira execução do laço de colunas.

Quando a linha 17 é alcançada pela primeira vez e a variável row é atualizada para um valor aleatório r, o atacante pode executar a fase de *Setup* por r iterações, mantendo na memória apenas $M[r]$, e não a matriz completa. Depois que $M[r]$ for processada, seu valor pode ser removido da memória, liberando espaço para a próxima linha a ser visitada. Entretanto, estas visitas subsequentes possuem mais um fator complicante: se a linha a ser visitada, $M[r']$, já foi visitada previamente e, portanto, atualizada, executar parte da fase de *Setup* irá fornecer uma versão desatualizada de $M[r']$ ao atacante. Para obter o valor correto que será fornecido à esponja, o atacante precisa executar a fase *Setup* completa, além de todos os passos da fase *Wandering* anteriores a última visitação de $M[r']$. Não obstante, como estes passos da fase *Wandering* podem, por sua vez, depender de linhas que já foram modificadas após a fase *Setup*, estas linhas também precisam ser recalculadas,

levando à chamadas recursivas, que crescem em tamanho, conforme mais linhas são modificadas.

Fornecer um limite acurado da complexidade de tal ataque contra o Lyra é, portanto, uma tarefa complicada. De fato, o ataque pode ser acelerado, se alguns estados intermediários forem mantidos na memória, adicionando mais variáveis à análise. Não obstante, visando compreender como o atacante poderia utilizar tais trocas de memória por processamento, serão discutidos cenários de testes mais simplificados. Em cada cenário, as complexidades totais de memória e processamento, mostradas no Teorema 1 serão igualadas, o que permite uma comparação direta entre a segurança do Lyra e do scrypt, quando o atacante tenta realizar esta troca de memória por processamento.

4.7.1.1 Notações

Seguindo a notação mostrada no algoritmo 4, s^i_j denota o estado da esponja quando as variáveis de controle dos laços de tempo e linhas são i e j, respectivamente, antes da linha correspondente ser visitada. $M^j[r]$ representa a linha r da matriz de memória durante a iteração j do laço de tempo, considerando que esta linha ainda não foi visitada durante esta iteração. Por último, $M^j[r_{(i,j)}]$ representa a linha que é visitada quando o estado interno da esponja é s^i_j. A figura 3 ilustra esta notação.

A fim de simplificar a análise, foi considerado que cada linha é visitada apenas uma vez em cada laço de tempo, ou seja, todas as linhas são visitadas em cada iteração j, sendo visitadas um total de T vezes. Esta simplificação é razoável para uma análise do comportamento médio, visto que a visitação de linhas deve seguir uma distribuição uniforme, com, aproximadamente, o mesmo número de visitas para cada linha.

A fim de tornar a análise mais concisa, ignora-se a pequena diferença

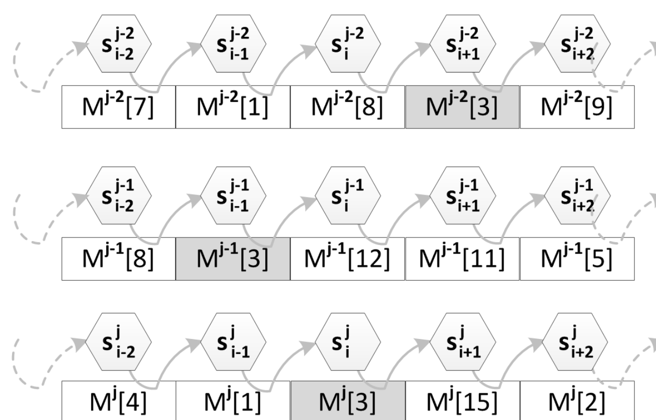

Figura 3: Exemplo simplificado da operação do Lyra, mostrando parte da fase *Wandering*. As células destacadas representam as linhas que serão computadas em sequência, considerando que cada uma delas é visitada, uma vez durante cada iteração do laço de tempo.

entre criar uma linha na fase *Setup* e visitar esta linha, na fase *Wandering*, sendo utilizado σ para representar ambos processos. Assim, o custo de σ é, aproximadamente, $C \cdot \rho/\rho_{max}$ chamadas à f.

4.7.1.2 Cenário 1: Armazenando todos os estados internos

No primeiro ataque considerado, o atacante nunca armazena uma linha $M[r]$, e sim o estado da esponja, exatamente antes desta linha ser processada. Então, sempre que for necessário acessar esta linha durante uma iteração j, o atacante deve calcular $M^0[r]$ a partir da senha e do sal, realizando r chamadas a σ. Depois, deve utilizar os j estados, armazenados previamente, para calcular $M^j[r]$, o que representa mais j chamadas a σ, e finalmente, prosseguir com a fase *Wandering*, chamando σ uma última vez para visitar $M^j[r]$. A ideia desta abordagem é que, como o estado da esponja geralmente é menor que uma linha inteira da matriz, caso o atacante decida por armazenar apenas o estado da esponja, ele deve ser capaz de reduzir o custo de memória do algoritmo, sofrendo a penalidade em termos de custo de processamento.

Por exemplo, no cenário descrito na Figura 3, o atacante pode decidir não armazenar $M[3]$, mas, apenas os estados anteriores a esta visitação. Neste caso, entre os estados mostrados, s_{i+1}^{j-2} e s_{i-1}^{j-1} devem estar armazenados quando a fase *Wandering* alcançar sua iteração j. Neste ponto, a linha i, a ser visitada é $M^j[r_{(i,j)}] = M^j[3]$, que deve ser calculada do início. Para isto, é necessário executar a fase *Setup* até $M^0[3]$ ser obtida e, então, executando σ iterativamente com os estados de esponja armazenados. $M^j[3]$ é obtido através da execução de σ com o estado s_{i-1}^{j-1} sobre $M^{j-1}[3]$, e depois executando σ com o estado s_{i+1}^{j-2} sobre $M^{j-2}[3]$.

O custo total de repetir esta estratégia para todo $0 \leqslant j < T$ e qualquer linha é, assim, $(R/2) \cdot T + T(T-1)/2$ chamadas a σ na média, armazenando $(T-1)$ estados intermediários. Estendendo esta estratégia, armazenando apenas estados da esponja, sem armazenar nenhuma linha, o custo é multiplicado por R, se tornando aproximadamente $(R+T) \cdot R \cdot T/2$ chamadas a σ na média e $R \cdot (T-1)$ estados intermediários.

Armazenar apenas algumas linhas da matriz, além dos estados da esponja, acelera o processo. Por exemplo, armazenando as linhas $M^j[R/2]$, estrategicamente posicionadas, para cada j, permite que qualquer linha seja calculada com apenas $(R/4)$ chamadas à σ, na média. Isto se deve ao fato de que, qualquer linha desejada, que tenha sido visitada antes de $M^j[R/2]$, durante a iteração j, possa ser calculada a partir desta, com o dobro de velocidade de se calcular esta mesma linha do início. Neste caso, o custo total de processar uma linha se torna $(R/4) \cdot T + T(T-1)/2)$ para todas as iterações $0 \leqslant j < T$. Generalizando esta abordagem, armazenar n linhas resulta em um custo total de $R((R/2n) \cdot T + T(T-1)/2)$ chamadas à σ, na média, para todo o processo. No entanto, como as linhas são C vezes maiores que o estado, o custo de memória total se torna o equivalente a $R \cdot (T-1) + n \cdot T \cdot C$ estados

intermediários.

Estas observações permitem a elaboração do Teorema 2. Este teorema mostra que, utilizando a estratégia descrita acima, de forma que nenhuma linha seja armazenada, o atacante tem seu custo de processamento elevado quadraticamente sobre o parâmetro R, que é o mesma penalidade obtida com o scrypt. Porém, de forma contrária ao scrypt, esta troca não é suficiente para reduzir o custo de memória para $O(1)$, sendo que este custo continua elevado.

Teorema 2. *O Lyra opera com os parâmetros T, R e C. Durante a operação regular do algoritmo, os custos de memória e processamento são, respectivamente, $O(R \cdot C)$ bits e $O(T \cdot R)$ chamadas à σ. Pode se obter um custo de memória de $O(R \cdot T)$ bits, ao elevar o custo de processamento a $O((R+T) \cdot R \cdot T)$ chamadas à σ.*

Demonstração. Os custos envolvidos na operação regular do Lyra são discutidos na seção 4.3, enquanto as trocas de custo de memória por processamento podem ser alcançadas utilizando-se do ataque descrito acima. □

4.7.1.3 Cenário 2: Armazenando poucos ou nenhum estado interno.

O atacante é capaz de reduzir ainda mais o custo de memória, se optar por armazenar um número menor de estados intermediários, calculando-os sob demanda, como é feito com as linhas da matriz de memória. O atacante pode determinar o estado s^j_{i+1}, se possuir o estado intermediário imediatamente anterior, s^j_i e a linha visitada pela esponja neste último estado, $M^j r_{(i,j)}$. Fazendo isto recursivamente, todas as iterações do algoritmo podem ser calculadas com um armazenamento mínimo, podendo chegar a $O(1)$. No entanto, como mostrado no Teorema 3, o custo computacional deste processo é muito mais elevado que o obtido com o scrypt.

Teorema 3. *O Lyra opera com os parâmetros T, R e C. Durante a operação regular do algoritmo, os custos de memória e processamento são, respectivamente, $O(R \cdot C)$ bits e $O(T \cdot R)$ chamadas à σ. Pode se obter um custo de memória de $O(1)$ bits, ao elevar o custo de processamento a $O(R^{T+1})$ chamadas à σ.*

Demonstração. Supondo que o atacante armazene apenas o último estado que foi calculado, bem como os estados intermediários obtidos no início de cada fase *Wandering*, s_0^j, descartando os demais. Quando o algoritmo entra na fase *Wandering* (i.e., para $j = 0$), o atacante pode obter o estado s_{i+1}^0 a partir de s_i^0, calculando a linha correspondente $M^0[r_{(i,0)}]$ do início. Assim, a fase de *Setup* é executada por $r_{(i,0)}$ iterações, resultando em um custo médio de $R/2$ chamadas à σ para cada valor de i. No entanto, para $j = 1$, o custo médio desta etapa é elevado para $R^2/4$, pois, computar s_{i+1}^1 a partir de s_i^1, requer o cálculo de $M^1[r_{(i,1)}]$, a partir de s_0^1 (o estado conhecido mais próximo), fazendo com que a fase de *Setup* seja executada $i + 1$ vezes, uma para cada $M^1[r_{(\alpha,1)}]$, onde $0 \leqslant \alpha \leqslant i$.

Seguindo o mesmo princípio, o custo médio de computar uma única atualização de estado s_{i+1}^j a partir de s_i^j, é $(R/2)^{j+1}$ chamadas à σ, o que leva à $R \cdot (R/2)^{j+1}$ chamadas durante toda a iteração j. Para a última iteração da fase *Wandering* apenas, o custo total é de $R \cdot (R/2)^T$, dominando o tempo de execução do Lyra neste cenário. □

4.7.1.4 Resumo

Assim, nota-se que o Lyra fornece um nível maior de segurança quando comparado ao scrypt, mesmo para valores de T pequenos. Supondo que o Lyra seja tão rápido quanto o scrypt para algum $T \geqslant 2$, e que ambos algoritmos operem sobre a mesma quantidade de elementos de memória R, resultando

em utilizações idênticas de memória. Neste caso, um ataque com pouca memória (Cenário 2) contra o Lyra tem complexidade $O(R^{T+1}) \geqslant O(R^3)$, enquanto no scrypt, esta complexidade será $O(R^2)$. Porém, se o atacante pode apenas elevar o custo de processamento quadraticamente, ele será capaz de atacar o scrypt com um custo de memória $O(1)$, enquanto o mesmo ataque ao Lyra resultará em um custo de memória de $O(R \cdot T)$.

4.7.2 Ataques com memória lenta

Fornecer segurança contra ataques com memória lenta é uma tarefa mais complexa, visto que o atacante se comporta como o usuário, durante a operação do algoritmo, mantendo toda a informação necessária armazenada em memória. A principal diferença entre o usuário legítimo e o atacante é a diferença de largura de banda, fornecida pelo dispositivo de memória utilizado, o que impacta no tempo necessário para gerar cada senha.

De maneira semelhante ao scrypt, o Lyra explora as propriedades de dispositivos de memória de baixo custo, através da ordem de visitação pseudo-aleatória. Esta estratégia eleva a latência de dispositivos de memória intrinsecamente sequenciais, como discos rígidos, especialmente se o atacante utiliza múltiplas instâncias simultaneamente, acessando diferentes seções de memória. Além disto, conforme discutido na seção 4.5, o padrão de visitação da memória, combinado com um valor baixo do parâmetro C pode diminuir os ganhos obtidos com mecanismos como *cahing* e *prefetching*, mesmo que o atacante utilize chips de memória de acesso aleatório.

Quando comparado ao scrypt, o Lyra introduz uma melhoria contra tais ataques, pois as posições de memória são lidas e escritas repetidamente, enquanto no scrypt, estas posições são apenas lidas. Com isto, o Lyra exige que os dados sejam, repetidamente, movidos na hierarquia de memória. O

impacto geral desta funcionalidade, no desempenho de ataques com memórias lentas, depende, no entanto, da arquitetura do sistema. Por exemplo, o tráfego no barramento de memória será elevado, enquanto os mecanismos de *cache* requerem um circuito mais complexo para lidar com o fluxo contínuo de informação entre eles e os níveis mais baixos de memória.

Outro aspecto positivo do projeto do Lyra é a realização de operações de XOR entre a linha e a saída da esponja, na fase *Wandering*, prevenindo que as posições de memória desta linha sejam subsistidas rapidamente. Assim, esta propriedade previne que o atacante utilize esta posição de memória em outra *thread* que está sendo executada em paralelo. Estas propriedades podem impactar à operação do usuário legítimo também, aumentando a necessidade da correta configuração dos parâmetros R, C e T, de acordo com a plataforma alvo.

4.8 Desempenho com os parâmetros recomendados

Para avaliar o desempenho do Lyra em *software*, foi utilizada a implementação de referência da função de compressão do Blake2b (AUMASSON *et al.*, 2013) como função f da esponja subjacente.

É importante notar que, apesar de geralmente os estados das esponjas serem inicializados com zeros, nesta implementação, os 512 bits menos significativos do estado são inicializados com zero e os 512 bits mais significativos recebem o valor do vetor de inicialização do Blake2b. Isto é necessário, pois o Blake2b não utiliza as constantes, originalmente, empregadas na função G do Blake (AUMASSON *et al.*, 2013), dependendo do vetor de inicialização para evitar pontos fixos. De fato, se o estado interno fosse totalmente preenchido

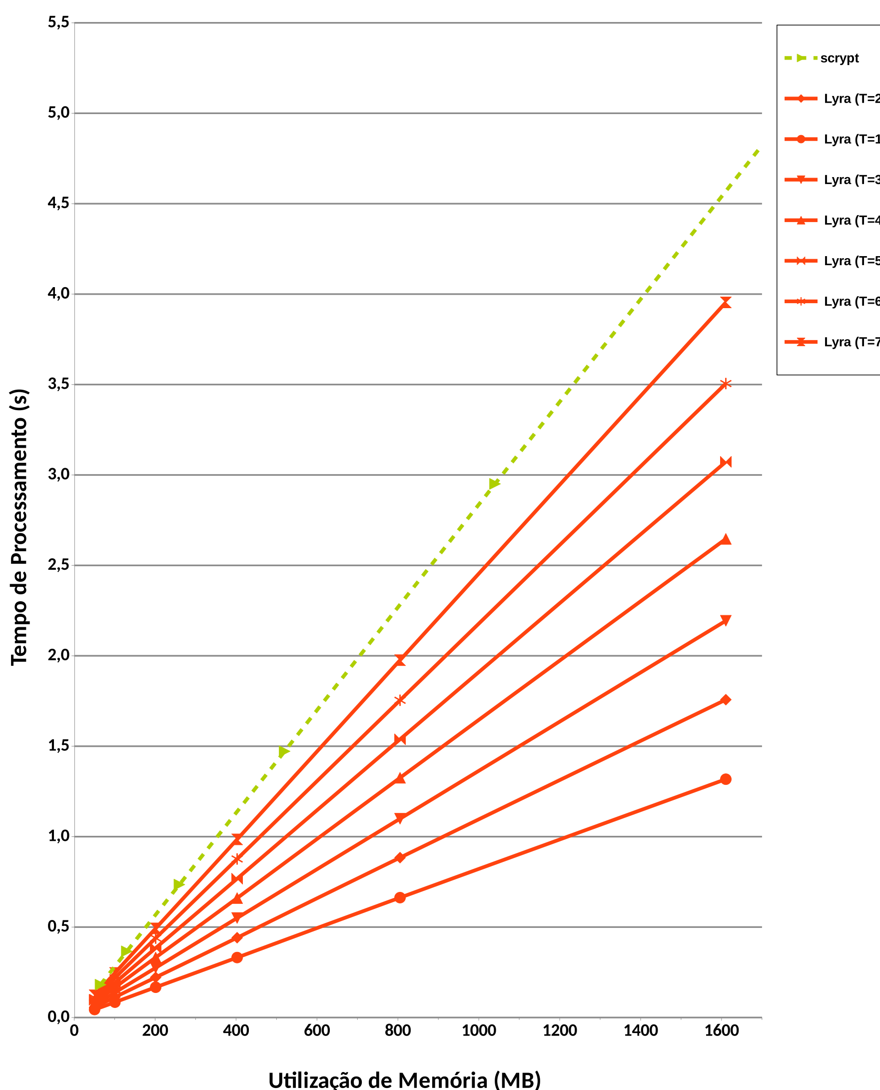

Figura 4: Desempenho do Lyra para $C = 256$, $\rho = 1$, e diferentes valores de T e R, comparado com scrypt.

com zeros, a operação de *absorb* com qualquer bloco de valor zero não iria alterar o valor do estado da esponja. Este não é um problema crítico para o Lyra, a não ser que a senha, o sal e o *basil* sejam vetores de zeros, grandes o suficiente para preencher blocos completos, pois caso contrário, a regra de preenchimento pad10*1 irá prevenir pontos fixos, mesmo que a entrada seja um vetor de zeros. Entretanto, a abordagem adotada é mais cautelosa e está de acordo com a especificação do Blake2b, que não é uma esponja.

Os resultados dos testes de desempenho são mostrados nas Figuras 4,

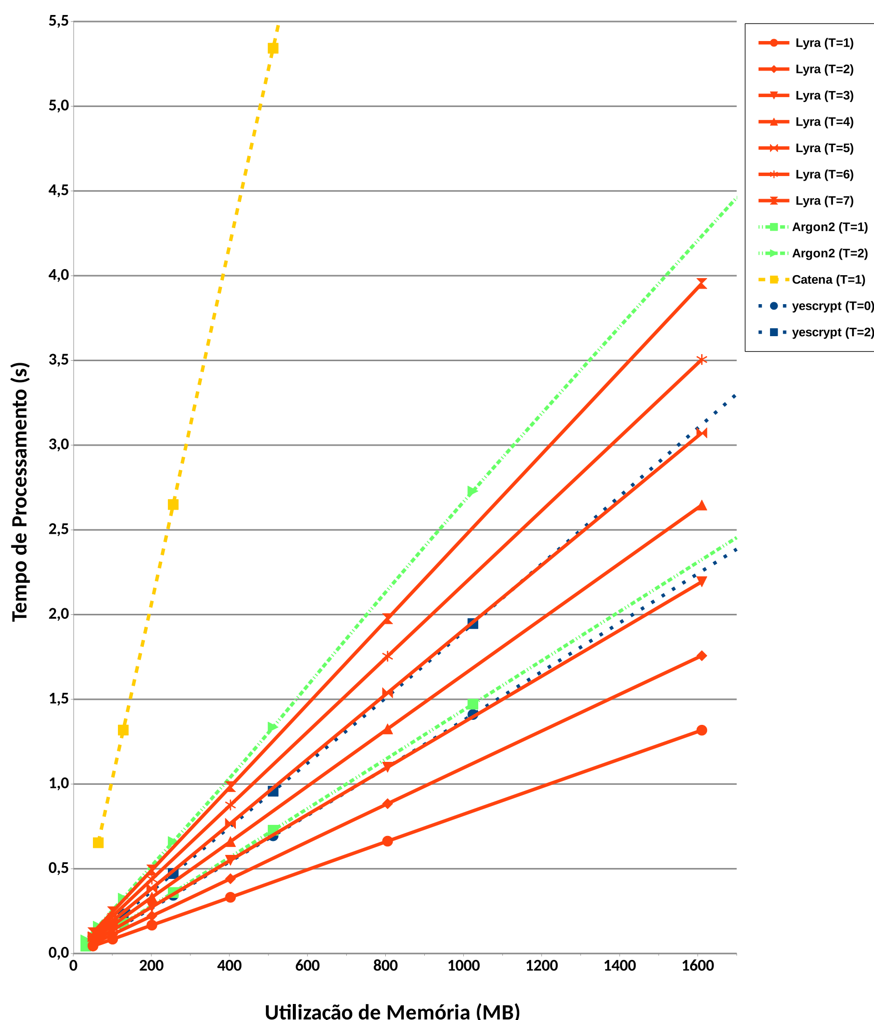

Figura 5: Desempenho do Lyra para $C = 256$, $\rho = 1$, e diferentes valores de T e R, comparado com os finalistas do PHC

5 e 6. Nestes testes, o Lyra foi parametrizado com $C = 256$, $\rho = 1$, $b = 736$ bits, e diferentes valores de T e R, fornecendo uma visão geral das possíveis combinações de parâmetros e seus resultados de desempenho. Todos os testes foram executados em uma plataforma equipada com um processador Intel Core i5-5200U (2.20 GHz Dual Core, 64 bits), 8 GB de DRAM, com sistema operacional Fedora 23 64 bits, utilizando o compilador *gcc* com otimização `-O3`.

A Figura 4 compara o Lyra com o scrypt, a Figura 6 compara o Lyra com o Lyra2, enquanto a Figura 5 compara o Lyra com os demais finalistas da com-

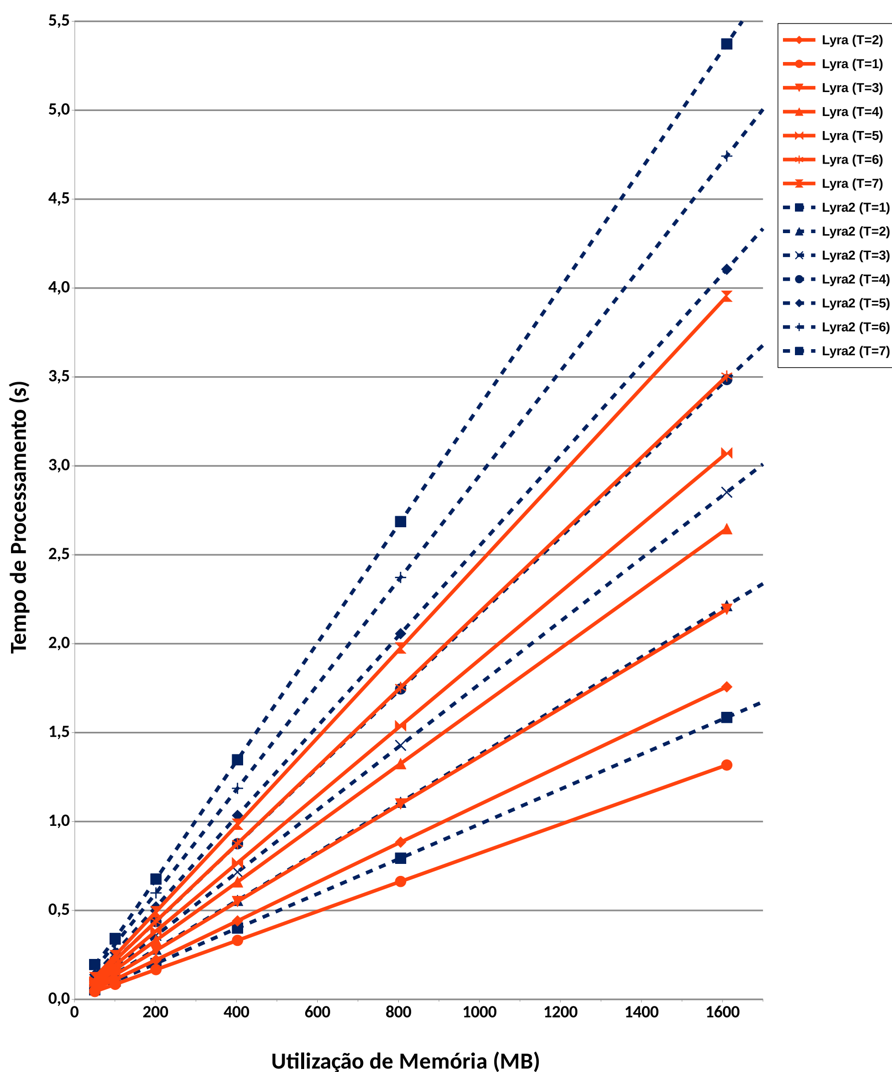

Figura 6: Desempenho do Lyra para $C = 256$, $\rho = 1$, e diferentes valores de T e R, comparado com o Lyra2

petição PHC. Conforme pode ser visto nas figuras, o algoritmo Lyra possui o melhor desempenho entre todos os que foram comparados, inclusive o Lyra2.

4.8.1 LyraP

Também foram realizados testes com a versão paralelizável do Lyra, conhecida como $Lyra_p$. Os resultados dos testes podem ser verificados na figura 7.

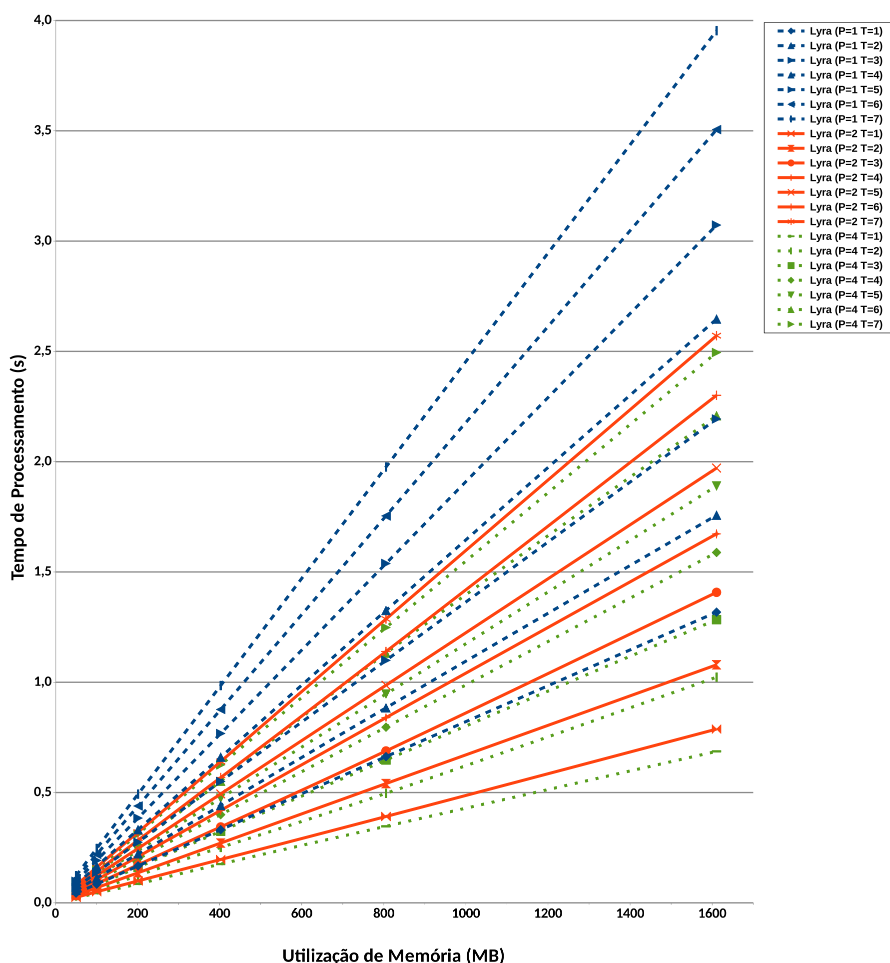

Figura 7: Desempenho do $Lyra_p$para $C = 256$, $\rho = 1$, e diferentes valores de T, R e p.

Esta figura compara o desempenho e consumo de memória do $Lyra_p$ para diferentes valores de p. É possível verificar que aumentando o valor de p, é possível aumentar o uso de memória, sem um impacto significativo no tempo de processamento.

Com $p = 1$, $R = 32.768$ e $T = 2$ o $Lyra_p$consome 800 MB de memória e executa em 0,82 segundos. Se alterarmos os parâmetros para $p = 2$, $R = 16.384$ e $T = 2$, o $Lyra_p$ consome os mesmos 800 MB de memória, porém executa em 0,54 segundos. Ao aumentarmos o valor de p para quatro, com os valores $p = 4$, $R = 8.192$ e $T = 2$, o $Lyra_p$ irá executar em 0,49 segundos, utilizando os mesmos 800 MB de memória durante sua execução.

Com isto, foi possível diminuir o tempo de execução de 0,82 segundos para 0,49 segundos, utilizando a capacidade de paralelismo da plataforma do usuário, sem perda de segurança. Ao diminuir o tempo de execução, mantendo a quantidade de memória utilizada, o algoritmo passa a ter maior resistência a ataques que trocam memória por processamento.

Se o usuário legítimo aceita um tempo de execução de um segundo, é possível aumentar o consumo de memória de $800MB$ para $1.400MB$, um ganho de 75%.

Ou seja, ao usar o paralelismo na plataforma legítima, aumenta-se mais ainda a resistência a ataques de força bruta e ataques que troquem memória por processamento.

4.8.2 Expectativa de custo de ataques

Considerando que o custo de um chip de memória DDR3 SO-DIMM seja aproximadamente U$5.00/GB (TRENDFORCE, 2014), a Tabela 1 mostra os custos de ataque ao Lyra com $T = 5$, quando um atacante tenta descobrir uma senha em um ano, utilizando este *hardware* citado acima.

Estes custos são obtidos considerando o número total de instâncias que devem ser executadas em paralelo, para ser possível testar todo o espaço de senhas em 365 dias, ignorando os custos relacionados à confecção e consumo de energia, e supondo que cada geração de chave demore o mesmo tempo obtido na nossa plataforma de testes.

Caso o atacante utilize uma plataforma mais veloz (e.g., uma FPGA ou computador mais poderoso), estes custos devem ser reduzidos proporcionalmente, visto que um número menor de instâncias seria necessário, reduzindo a quantidade de chips de memória. Similarmente, se o atacante utilizar memó-

rias mais rápidas (e.g., SRAM ou registradores), o tempo de processamento também será reduzido, diminuindo o número de instâncias que devem ser executadas em paralelo. Entretanto, neste caso, o custo de memória resultante pode ser significativamente maior, devido ao maior custo por GB destes dispositivos de memória. Contudo, os números fornecidos pela Tabela 1 não são valores absolutos, e sim uma referência, a fim de determinar o ganho de proteção obtido por utilizar o Lyra, visto que este custo adicional de memória é a principal vantagem de funções de derivação de chaves que exploram uso de memória.

Por fim, ao ser comparado ao scrypt, o Lyra possui a vantagem de impor custos de processamento elevados, caso o atacante tente evitar custos relacionados à memória, desencorajando tais abordagens. Considerando a fase *Wandering* final, apenas, e $T = 5$, o processamento adicional para se realizar um ataque sem memória contra o Lyra é de aproximadamente $(6.4 \cdot 10^4)^6 = 6.9 \cdot 10^{28}$ chamadas à σ, caso o algoritmo opere com 200 MB, ou $(3.2 \cdot 10^5)^6 = 1.1 \cdot 10^{33}$ chamadas à σ, para 1 GB de memória. Com esta mesma utilização de memória, o custo de processamento total de aplicar um ataque sem memória contra o scrypt é de $(2 \cdot 10^5)^2 = 4 \cdot 10^{10}$ chamadas à função *BlockMix* para 200 MB e $(1 \cdot 10^6)^2 = 1 \cdot 10^{12}$ chamadas para 1 GB. O tempo de processamento da função *BlockMix* é semelhante ao de σ para os

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

Tabela 1: Custo de memória (em U$) obtidos com o Lyra with $T = 6$, para atacantes tentando descobrir senhas em um período de um ano, utilizando um Intel Core i5-2400.

parâmetros utilizados em nosso experimento. Como esperado, tais custos de processamento elevados devem desencorajar ataques que evitem os custos de memória do Lyra, através de processamento adicional.

5 CONCLUSÕES E PRÓXIMOS PASSOS

Foi apresentado um novo PHS, Lyra, que permite que o usuário legítimo configure o tempo de processamento e utilização de memória, de acordo com a plataforma a seu dispor e o nível de segurança desejado. Para atingir este objetivo, o Lyra foi construído com base nas propriedades das funções esponja, operando de forma *stateful* e sequencial. Assim, a matriz de memória pode ser vista como um estado interno do Lyra, em conjunto com o estado da esponja.

Estas características do Lyra permitem a proteção ao usuário contra ataques de força bruta. E, mesmo que o atacante deseje trocar o custo de memória por custo de processamento, o Lyra impões sérias restrições à esta abordagem, fazendo com que o custo de processamento cresça muito rapidamente. Este crescimento exponencial é melhor que o encontrado no scrypt, que possui crescimento quadrático. Quando comparado ao scrypt, o Lyra também permite que, para um mesmo tempo de execução, o usuário legítimo possa utilizar uma quantidade maior de memória, aumentando a proteção contra ataques de força bruta.

Ao utilizar a capacidade de paralelismo da plataforma do usuário legítimo, é possível aumentar ainda mais a proteção contra ataques de força bruta, através do $Lyra_p$. É possível alcançar um ganho de 75% aumentando o número de *threads* de um para quatro.

O desempenho do Lyra se mostrou uma de suas maiores vantagens. Na comparação de desempenho, o Lyra se saiu melhor quando comparado a todos os finalistas do PHC.

Este trabalho produziu um artigo, publicado no *Journal of Cryptographic Engineering* (ALMEIDA *et al.*, 2014), além de servir de base para outro PHS, chamado Lyra2, que, por sua vez, está participando da competição de funções de derivação de chaves PHC. O Lyra2 mantém a mesma estrutura do Lyra, utilizando funções esponjas de forma *stateful* ao longo do processamento, número de rodadas reduzido e parametrização de custo de memória e tempo desacoplados. Porém, o Lyra2 explora uma maior utilização de banda de memória, visando dificultar ainda mais os ataques de força bruta, além de paralelismo em plataformas legítimas.

Os próximos passos são análises dos ataques possíveis contra o Lyra2, visando prover resistência a estes ataques.

REFERÊNCIAS

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>.

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.

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.; 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.

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. *Argon and Argon2*. v2. Luxembourg, 2015. <https://password-hashing.net/submissions/specs/Argon-v2.pdf>. Accessed: 2015-05-18.

BIRYUKOV, A.; KHOVRATOVICH, D.; GROZSCHAEDL, J. *Tradeoff cryptanalysis of password hashing schemes*. 2014. <https://www.cryptolux.org/images/5/57/Tradeoffs.pdf>.

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.

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.

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.

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.

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.; 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.

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.

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.

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.

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.

NIST. *Federal Information Processing Standard (FIPS PUB 198) – The Keyed-Hash Message Authentication Code*. National Institute of Standards and Technology, U.S. Department of Commerce. 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. *CUDA C Programming Guide*. 2012. <http://docs.nvidia.com/cuda/cuda-c-programming-guide/>.

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

PERCIVAL, C. 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.

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.

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.

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.

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.

TRENDFORCE. *DRAM Contract Price (Jan.22 2014)*. 2014. Http://www.trendforce.com/price (visited on Mar.29, 2014).

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

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.