Segurança de Esquemas de Hash de Senhas (*Password Hashing Schemes* – PHSs)

Ewerton R. Andrade

ewerton.andrade@unir.br

DACC - Departamento Acadêmico de Ciência da Computação
UNIR - Fundação Universidade Federal de Rondônia

19 de outubro de 2020

Agenda

1. Introdução
   - Entropia
   - Ataque de força bruta
   - Tabelas de consulta
2. Plataformas de Ataque
   - *Graphics Processing Units (GPUs)*
   - *Field Programmable Gate Arrays (FPGAs)*
3. Complexidade de Alguns Ataques
   - PBKDF2
   - BCRYPT
   - SCRYPT
   - Lyra
4. Considerações Finais
   - PHS ao longo dos anos
   - Funções utilizadas internamente
   - Conclusões

Agenda

1. Introdução
   - Entropia
   - Ataque de força bruta
   - Tabelas de consulta
2. Plataformas de Ataque
   - *Graphics Processing Units (GPUs)*
   - *Field Programmable Gate Arrays (FPGAs)*
3. Complexidade de Alguns Ataques
   - PBKDF2
   - BCRYPT
   - SCRYPT
   - Lyra
4. Considerações Finais
   - PHS ao longo dos anos
   - Funções utilizadas internamente
   - Conclusões

Entropia

- A entropia de uma senha é uma **medida da incerteza** do valor daquela senha, sendo expressa em bits;
- Quanto maior a entropia de uma senha, mais difícil se torna para um atacante descobri-la, pois existe um número maior de combinações possíveis [BDP06];
- Considerando uma senha de l caracteres, escolhidos aleatoriamente de um alfabeto b, então [BDP06]:

$$\text{Entropia} = \log_2(b^l)$$

Entropia de senhas escolhidas por usuários

- O cálculo da entropia de senhas escolhidas por usuários, entretanto, se dá de maneira diferente, pois os caracteres em questão **não possuem uma distribuição aleatória**;
- Para realizar o cálculo desta entropia, pode-e usar o conjunto de regras definido pelo NIST [BDP06]:
  - A entropia do primeiro caractere é definida com sendo 4 bits;
  - A entropia dos próximos sete caracteres é definida como sendo 2 bits por caractere;
  - Do nono ao vigésimo carácter, a entropia é de 1,5 bits por caractere;
  - A partir do vigésimo primeiro caractere, a entropia é de 1 bit por caractere;
  - Se são utilizados caracteres em caixa alta e símbolos não alfabéticos, são adicionados 6 bits à entropia;
  - Se for realizada uma verificação da senha contra um dicionário de senhas comuns, são adicionados mais 6 bits ao valor da entropia.

Entropia em algumas páginas da Internet [FH07]

- Média de entropia de áginas na Internet é de apenas **40,54 bits**;
- Páginas mais sensíveis como *paypal*, possuem uma entropia média de **42,04 bits**;
- A página *Microsoft Outlook Web App* (OWA) possui regras que obrigam a escolha de uma senha segura, e com isto obteve a média de **51,36 bits** de entropia.

Nível básico de segurança costuma ser de **128 bits**.

Ataque de força bruta

- Neste tipo de ataque **não há uma limitação da quantidade de tentativas**, diferentemente de uma autenticação online, em que se poderia **bloquear a senha após algumas tentativas incorretas** [CDW04];
- Nas primeiras versões do Unix por exemplo, era armazenado o *hash* das senhas em arquivo, utilizando uma função não reversível chamada *crypt*.
- Com o aumento do poder computacional, se tornou "relativamente barato" descobrir as senhas dos usuários de Unix, **simplesmente** calculando o *hash* de todas as senhas possíveis e comparando-o com os valores armazenados [PM99].

Tabelas de consulta

- **Tabelas de consulta** visam trocar tempo de processamento por memória, nos permitindo recuperar o texto original (*plaintext*) de uma senha através da busca do valor de seu *hash*. *Hash* este, gerado a partir de uma função criptográfica (função de *hash*, por exemplo: MD5, SHA-1, SHA-2, etc).

Tabelas de consulta – Exemplo

Valor do *Hash*	Texto Original
d577273ff885c3f84dadb8578bb41399	12345
165969f0f64edac96aa92b15463e9ec8	minhasenha
e8eaffd121b603c9c62029ffe6a306d3	123senha
ffca61e8725743f630ec39d5e0d2b655	12/10/1965
fc83f799d71a3eb160d7d34444871dbd	maria

Tabela: Tabela de consulta (MD5).

Valor do *Hash*	Texto Original
f33ae3bc9a22cd7564990a794789954409977013966fb1a8f43c35776b833a95	12345
1625d17edbd9f93dad0e5c4a562d2aeedad5d44a0dbb6a1f8457768ba8f2785f	minhasenha
a72ffd5bea8c370cf6396e7bc040f8977f9082be9d3b0cbd73d8a835dde87b3f	123senha
5231e56b40dfb4863437d98c157533bfaf8412b2b1775892477fc26a41d4a2e6	12/10/1965
6738ef1b0509836ea7a0fcc2f31887930454c96bb9c7bf2f6b04adbe2bb0d290	maria

Tabela: Tabela de consulta (SHA2-256).

Tabelas de consulta – Exemplo

Valor do *Hash*	Texto Original
d577273ff885c3f84dadb8578bb41399	12345
165969f0f64edac96aa92b15463e9ec8	minhasenha
e8eaffd121b603c9c62029ffe6a306d3	123senha
ffca61e8725743f630ec39d5e0d2b655	12/10/1965
fc83f799d71a3eb160d7d34444871dbd	maria

Tabela: Tabela de consulta (MD5).

Usuário	Valor do *Hash*	e-mail
Joao1970	3cd7a0db76ff9dca48979e24c39b408c	joao@uol.com.br
JoseDaSilva	fc83f799d71a3eb160d7d34444871dbd	josedasilva@gmail.com
Antonio	05c3f4a5ece94ee4788ba349504618a0	antonio1970@globo.com

Tabela: Tabela de Usuários em um Banco de Dados.

Tabelas de consulta – Exemplo

Valor do *Hash*	Texto Original
d577273ff885c3f84dadb8578bb41399	12345
165969f0f64edac96aa92b15463e9ec8	minhasenha
e8eaffd121b603c9c62029ffe6a306d3	123senha
ffca61e8725743f630ec39d5e0d2b655	12/10/1965
fc83f799d71a3eb160d7d34444871dbd	maria

Tabela: Tabela de consulta (MD5).

Usuário	Valor do *Hash*	e-mail
Joao1970	3cd7a0db76ff9dca48979e24c39b408c	joao@uol.com.br
JoseDaSilva	fc83f799d71a3eb160d7d34444871dbd	josedasilva@gmail.com
Antonio	05c3f4a5ece94ee4788ba349504618a0	antonio1970@globo.com

Tabela: Tabela de Usuários em um Banco de Dados.

Tabelas de consulta – *salt*

- A fim de **impedir o funcionamento das "Rainbow Tables"**, adiciona-se ao cálculo da chave derivada um valor aleatório, chamado de *salt*;
- Por exemplo, o cálculo de *hash* usando o *salt* poderia ser feito como:

$$H(senha||salt)$$

onde || representa a operação de concatenação e H representa a função de *hash* escolhida.

Agenda

1. Introdução
   - Entropia
   - Ataque de força bruta
   - Tabelas de consulta
2. Plataformas de Ataque
   - *Graphics Processing Units (GPUs)*
   - *Field Programmable Gate Arrays (FPGAs)*
3. Complexidade de Alguns Ataques
   - PBKDF2
   - BCRYPT
   - SCRYPT
   - Lyra
4. Considerações Finais
   - PHS ao longo dos anos
   - Funções utilizadas internamente
   - Conclusões

Plataformas de Ataque

- As principais ameaças dos PHSs são as plataformas que se beneficiam de **hardwares massivamente paralelo** e altamente escalável e sejam relativamente baratos;
- Nesta linha, os exemplos mais proeminentes são Unidades de Processamento Gráfico (**GPUs**) e hardware personalizado a partir de **FPGAs** [DGK12].

GPUs – Marcos evolutivos

- Com o aumento da procura de **renderização de alta definição** em tempo real, tradicionalmente realizado um grande número de núcleos de processamento, fizeram com que as GPUs aumentassem sua capacidade de paralelização;
- Mais recentemente, GPUs evoluíram de apenas **plataformas específicas para dispositivos de computação universal**. Começando a dar suporte a linguagens como CUDA[Nvi12a] e OpenCL [Khr12], que ajudam a aproveitar o seu poder computacional;
- Com isto, as GPUs começaram a ser utilizadas para fins mais genéricos, incluindo a **quebra de senha** [Spr11, DGK12].

GPUs – Alguns exemplos

NVidia Tesla K20X [Nvi12b]:

- 2.688 núcleos de 732 MHz;
- 6 GB de DRAM compartilhada, com 250 GB/s de taxa de transferência.

NVidia GT540M (do meu notebook):

- 96 núcleos de 900 MHz;
- 2 GB de DRAM compartilhada, com 28,8 GB/s de taxa de transferência.

GPUs – Possível cenário

Suponha um cenário em que o adversário disponha de uma NVidia Tesla K20X.

- Caso os senhas sejam armazenadas utilizando apenas a função de *hash* MD5 aplicado no texto legível, onde a função MD5 leve apenas **2 ms para ser executada**, consumindo somente **0.5 MB de memória**.

GPUs – Possível cenário

Suponha um cenário em que o adversário disponha de uma NVidia Tesla K20X.

- Caso os senhas sejam armazenadas utilizando apenas a função de *hash* MD5 aplicado no texto legível, onde a função MD5 leve apenas **2 ms para ser executada**, consumindo somente **0.5 MB de memória**.

Neste cenário é fácil perceber que o adversário testará 2.688 senhas a cada dois ms.

Resultando em 1.344.000 senhas testada por segundo, ou seja, $4.838.400.000 \approx 2^{32,17}$ senhas testadas por hora.

GPUs – Possível cenário

Suponha um cenário em que o adversário disponha de uma NVidia Tesla K20X.

- Caso os senhas sejam armazenadas utilizando apenas a função de *hash* MD5 aplicado no texto legível, onde a função MD5 leve apenas **2 ms para ser executada**, consumindo somente **0.5 MB de memória**.

Neste cenário é fácil perceber que o adversário testará 2.688 senhas a cada dois ms.

Resultando em 1.344.000 senhas testada por segundo, ou seja, $4.838.400.000 \approx 2^{32,17}$ senhas testadas por hora.

Todavia, **se algum PHS que utilize 20 MB de RAM** for utilizado no lugar da função de *hash* MD5, o número máximo de núcleos que podem ser utilizados simultaneamente, torna-se 300, apenas 11 % do total de núcleos disponível.

FPGA – Conceito Geral

- Um FPGA é um circuito integrado programável de alta performance;
- Como esses dispositivos são configurados para executar uma tarefa específica, eles **podem ser otimizados** para um determinado fim (por exemplo, usando pipelining [Dan08, KMM+06]);
- Além disto, quando comparado com as GPUs, FPGAs pode ser vantajosas devido seu **consumo de energia consideravelmente menor** [CMHM10, FBCS12].

FPGA – Recente exemplo de quebra de senhas [DGK12]

- Devido ao **pequeno consumo de memória** do algoritmo PBKDF2, sendo que a maioria do funcionamento básico envolve a função SHA-2, que é realizada usando o cache de memória do dispositivo (muito mais rápido do que o DRAM) [DGK12, Sec. 4.2];
- Dürmuth *et al*, usando um cluster RIVYERA S3-5000 [Sci] com 128 FPGAs, demostraram ser possível testar 356.352 senhas por segundo em uma arquitetura onde 5.376 senhas são processadas em paralelo.

FPGA – Recente exemplo de quebra de senhas [DGK12]

- Devido ao **pequeno consumo de memória** do algoritmo PBKDF2, sendo que a maioria do funcionamento básico envolve a função SHA-2, que é realizada usando o cache de memória do dispositivo (muito mais rápido do que o DRAM) [DGK12, Sec. 4.2];
- Dürmuth *et al*, usando um cluster RIVYERA S3-5000 [Sci] com 128 FPGAs, demostraram ser possível testar 356.352 senhas por segundo em uma arquitetura onde 5.376 senhas são processadas em paralelo.

Todavia – assim como no exemplo das GPUs – **se algum PHS que utilize 20 MB de RAM** for utilizado no lugar da função de PBKDF2, o número máximo de senhas será drasticamente reduzido.

No exemplo em questão, somente 3 senhas poderiam ser processados em paralelo, já que os FPGAs envolvidos dispõem de 64 MB de RAM.

Agenda

1. Introdução
   - Entropia
   - Ataque de força bruta
   - Tabelas de consulta
2. Plataformas de Ataque
   - *Graphics Processing Units (GPUs)*
   - *Field Programmable Gate Arrays (FPGAs)*
3. Complexidade de Alguns Ataques
   - PBKDF2
   - BCRYPT
   - SCRYPT
   - Lyra
4. Considerações Finais
   - PHS ao longo dos anos
   - Funções utilizadas internamente
   - Conclusões

PBKDF2

Algorithm PBKDF2.

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

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

Onde:
k corresponde ao tamanho desejado para a chave gerada pelo PBKDF2; e

h corresponde ao tamanho da saída da função utilizada internamente.

PBKDF2

Algorithm PBKDF2.

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

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

Onde:
k corresponde ao tamanho desejado para a chave gerada pelo PBKDF2; e

h corresponde ao tamanho da saída da função utilizada internamente.

PBKDF2 – Resumo

Seja,

- τ a quantidade de memória utilizada pelas variáveis do sistema.

Ataques					
PBKDF2	Sequencial (Padrão)		Estágios Intermediários		Sem memória*
	Memória	Tempo	Memória	Tempo	Tempo
	$\mathcal{O}(\tau)$	$\mathcal{O}(l.T)$	-	-	-

Tabela: Complexidade dos ataques aplicáveis ao PBKDF2.

BCRYPT

Algorithm Bcrypt.

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

1: $s \leftarrow InitState()$ ▷ Copies the digits of π into the sub-keys and S-boxes S_i
2: $s \leftarrow ExpandKey(s, salt, pwd)$
3: **for** $i \leftarrow 1$ **to** 2^T **do**
4: $\quad s \leftarrow ExpandKey(s, 0, salt)$; $s \leftarrow ExpandKey(s, 0, pwd)$
5: **end for**
6: $ctext \leftarrow$ "$OrpheanBeholderScryDoubt$"
7: **for** $i \leftarrow 1$ **to** 64 **do** { $ctext \leftarrow BlowfishEncrypt(s, ctext)$ } **end for**
8: **return** $T \,||\, salt \,||\, ctext$
9: **function** EXPANDKEY($s, salt, pwd$)
10: $\quad$ **for** $i \leftarrow 1$ **to** 32 **do** { $P_i \leftarrow P_i \oplus pwd[32 * (i-1) \ldots 32 * i - 1]$ } **end for**
11: $\quad$ **for** $i \leftarrow 1$ **to** 9 **do**
12: $\quad\quad temp \leftarrow BlowfishEncrypt(s, salt[64 * (i-1) \ldots 64 * i - 1])$
13: $\quad\quad P_{0+(i-1)*2} \leftarrow temp[0 \ldots 31]$; $P_{1+(i-1)*2} \leftarrow temp[32 \ldots 64]$
14: $\quad$ **end for**
15: $\quad$ **for** $i \leftarrow 1$ **to** 4 **do**
16: $\quad\quad$ **for** $j \leftarrow 1$ **to** 128 **do**
17: $\quad\quad\quad temp \leftarrow BlowfishEncrypt(s, salt[64 * (j-1) \ldots 64 * j - 1])$
18: $\quad\quad\quad S_i[(j-1) * 2] \leftarrow temp[0 \ldots 31]$; $S_i[1 + (j-1) * 2] \leftarrow temp[32 \ldots 63]$
19: $\quad\quad$ **end for**
20: $\quad$ **end for**
21: $\quad$ **return** s
22: **end function**

BCRYPT

Algorithm Bcrypt.

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

1: $s \leftarrow InitState()$ ▷ Copies the digits of π into the sub-keys and S-boxes S_i
2: $s \leftarrow ExpandKey(s, salt, pwd)$
3: **for** $i \leftarrow 1$ **to** 2^T **do** — 2^T
4: $s \leftarrow ExpandKey(s, 0, salt)$; $s \leftarrow ExpandKey(s, 0, pwd)$ — $\approx 2^9.2^T + 2^6$
5: **end for**
6: $ctext \leftarrow$ "$OrpheanBeholderScryDoubt$"
7: **for** $i \leftarrow 1$ **to** 64 **do** { $ctext \leftarrow BlowfishEncrypt(s, ctext)$ } **end for** — 2^6
8: **return** $T \,||\, salt \,||\, ctext$
9: **function** EXPANDKEY($s, salt, pwd$)
10: **for** $i \leftarrow 1$ **to** 32 **do** { $P_i \leftarrow P_i \oplus pwd[32 * (i-1) \ldots 32 * i - 1]$ } **end for** — 2^5
11: **for** $i \leftarrow 1$ **to** 9 **do**
12: $temp \leftarrow BlowfishEncrypt(s, salt[64 * (i-1) \ldots 64 * i - 1])$
13: $P_{0+(i-1)*2} \leftarrow temp[0 \ldots 31]$; $P_{1+(i-1)*2} \leftarrow temp[32 \ldots 64]$ — 9
14: **end for**
15: **for** $i \leftarrow 1$ **to** 4 **do** — 2^2
16: **for** $j \leftarrow 1$ **to** 128 **do**
17: $temp \leftarrow BlowfishEncrypt(s, salt[64 * (j-1) \ldots 64 * j - 1])$
18: $S_i[(j-1) * 2] \leftarrow temp[0 \ldots 31]$; $S_i[1 + (j-1) * 2] \leftarrow temp[32 \ldots 63]$ — $2^2.2^7$
19: **end for**
20: **end for**
21: **return** s
22: **end function**

BCRYPT – Resumo

Sejam,

- τ a quantidade de memória utilizada pelas variáveis do sistema;
- β os 4 KBytes de memória utilizados pelas S-Boxes e sub-chaves do algoritmo Blowfish [PM99].

Ataques					
	Sequencial (Padrão)		Estágios Intermediários		Sem memória*
BCRYPT	Memória $\mathcal{O}(\tau+\beta)$	Tempo $\mathcal{O}(2^{9+T})$	Memória -	Tempo -	Tempo -

Tabela: Complexidade dos ataques aplicáveis ao BCRYPT.

SCRYPT

Algorithm Scrypt.

PARAM: h ▷ The output length of *BlockMix*'s internal hash function
INPUT: pwd ▷ The password
INPUT: $salt$ ▷ A random salt
INPUT: k ▷ The key length
INPUT: b ▷ The block size, satisfying $b = 2r \cdot h$
INPUT: R ▷ Cost parameter (memory usage and processing time)
INPUT: p ▷ Parallelism parameter
OUTPUT: K ▷ The password-derived key

$(B_0 ... B_{p-1}) \leftarrow PBKDF2_{HMAC-SHA-256}(pwd, salt, 1, p \cdot b)$
for $i \leftarrow 0$ **to** $p - 1$ **do** { $B_i \leftarrow ROMix(B_i, R)$ } **end for**
$K \leftarrow PBKDF2_{HMAC-SHA-256}(pwd, B_0 || B_1 || ... || B_{p-1}, 1, k)$
return K ▷ Outputs the k-long key

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

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

SCRYPT – Sequencial (Padrão)

Algorithm Scrypt.

PARAM: h ▷ The output length of *BlockMix*'s internal hash function
INPUT: pwd ▷ The password
INPUT: $salt$ ▷ A random salt
INPUT: k ▷ The key length
INPUT: b ▷ The block size, satisfying $b = 2r \cdot h$
INPUT: R ▷ Cost parameter (memory usage and processing time)
INPUT: p ▷ Parallelism parameter
OUTPUT: K ▷ The password-derived key

Memória $\approx p.R.2r$
Processamento $\approx p.R.2r$

1: $(B_0 ... B_{p-1}) \leftarrow PBKDF2_{HMAC-SHA-256}(pwd, salt, 1, p \cdot b)$
2: **for** $i \leftarrow 0$ **to** $p-1$ **do** { $B_i \leftarrow ROMix(B_i, R)$ } **end for** — p
3: $K \leftarrow PBKDF2_{HMAC-SHA-256}(pwd, B_0 || B_1 || ... || B_{p-1}, 1, k)$
4: **return** K ▷ Outputs the k-long key

5: **function** ROMIX(B, R) ▷ Sequential memory-hard function
6: $X \leftarrow B$
7: **for** $i \leftarrow 0$ **to** $R-1$ **do** ▷ Initializes memory array V
8: $V_i \leftarrow X$; $X \leftarrow BlockMix(X)$
9: **end for** — R
10: **for** $i \leftarrow 0$ **to** $R-1$ **do** ▷ Reads random positions of V
11: $j \leftarrow Integerify(X) \bmod R$; $X \leftarrow BlockMix(X \oplus V_j)$
12: **end for** — R
13: **return** X
14: **end function**

15: **function** BLOCKMIX(B) ▷ Hash function with (b-long) inputs/outputs
16: $Z \leftarrow B_{2r-1}$ ▷ $r = b/2h$, where $h = 512$ for Salsa20/8
17: **for** $i \leftarrow 0$ **to** $2r-1$ **do** { $Z \leftarrow Hash(Z \oplus B_i)$; $Y_i \leftarrow Z$ } **end for** — $2r$
18: **return** $(Y_0, Y_2, ..., Y_{2r-2}, Y_1, Y_3, Y_{2r-1})$
19: **end function**

SCRYPT – Sem memória*

Algorithm Scrypt.

PARAM: h ▷ The output length of *BlockMix*'s internal hash function
INPUT: pwd ▷ The password
INPUT: $salt$ ▷ A random salt
INPUT: k ▷ The key length
INPUT: b ▷ The block size, satisfying $b = 2r \cdot h$
INPUT: R ▷ Cost parameter (memory usage and processing time)
INPUT: p ▷ Parallelism parameter
OUTPUT: K ▷ The password-derived key

Processamento ≈ *p.R.R.2r*

1: $(B_0 ... B_{p-1}) \leftarrow PBKDF2_{HMAC-SHA-256}(pwd, salt, 1, p \cdot b)$
2: **for** $i \leftarrow 0$ **to** $p - 1$ **do** { $B_i \leftarrow ROMix(B_i, R)$ } **end for** — *p*
3: $K \leftarrow PBKDF2_{HMAC-SHA-256}(pwd, B_0 || B_1 || ... || B_{p-1}, 1, k)$
4: **return** K ▷ Outputs the k-long key

5: **function** ROMIX(B, R) ▷ Sequential memory-hard function
6: $X \leftarrow B$
7: **for** $i \leftarrow 0$ **to** $R - 1$ **do** ▷ Initializes memory array V
8: $V_i \leftarrow X$; $X \leftarrow BlockMix(X)$
9: **end for** — *R*
10: **for** $i \leftarrow 0$ **to** $R - 1$ **do** ▷ Reads random positions of V
11: $j \leftarrow Integerify(X) \bmod R$; $X \leftarrow BlockMix(X \oplus V_j)$
12: **end for** — *R*
13: **return** X
14: **end function**

15: **function** BLOCKMIX(B) ▷ Hash function with (b-long) inputs/outputs
16: $Z \leftarrow B_{2r-1}$ ▷ $r = b/2h$, where $h = 512$ for Salsa20/8
17: **for** $i \leftarrow 0$ **to** $2r - 1$ **do** { $Z \leftarrow Hash(Z \oplus B_i)$; $Y_i \leftarrow Z$ } **end for** — *2r*
18: **return** $(Y_0, Y_2, ..., Y_{2r-2}, Y_1, Y_3, Y_{2r-1})$
19: **end function**

SCRYPT – Resumo

Ataques					
	Sequencial (Padrão)		Estágios Intermediários		Sem memória*
SCRYPT	Memória	Tempo	Memória	Tempo	Tempo
	$O(R)$	$O(R)$	-	-	$O(R^2)$

Tabela: Complexidade dos ataques aplicáveis ao SCRYPT.

Lyra

Algorithm The Lyra Algorithm.

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

▷ **Setup phase**: Initializes a $(R \times C)$ memory matrix whose cells have b bits each
$Hash.absorb(\text{pad}(pwd \,\|\, salt))$ ▷ Padding rule: 10^*1
for $row \leftarrow 1$ **to** $R - 1$ **do**
    $M[row] \leftarrow Hash.squeeze_\rho(C \cdot b)$
end for
▷ **Wandering phase**: Iteratively overwrites blocks of the memory matrix
$row \leftarrow 0$
for $i \leftarrow 0$ **to** $T - 1$ **do** ▷ **Time Loop**
    for $j \leftarrow 0$ **to** $R - 1$ **do** ▷ **Rows Loop**: randomly visits R rows
        for $col \leftarrow 0$ **to** $C - 1$ **do** ▷ **Columns Loop**: visits blocks in row
            $M[row][col] \leftarrow M[row][col] \oplus Hash.duplexing_\rho(M[row][col], b)$
        end for
        $row \leftarrow Hash.duplexing((M[row][C-1] \bmod C), |row|) \bmod R$
    end for
end for
▷ **Wrap-up phase**: key computation
$Hash.absorb(\text{pad}(pwd \,\|\, salt))$ ▷ Uses the sponge's current state
$K \leftarrow Hash.squeeze(k)$
return K ▷ Outputs the k-long key

Lyra – Sequencial (Padrão)

Algorithm The Lyra Algorithm.

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

▷ **Setup phase**: Initializes a $(R \times C)$ memory matrix whose cells have b bits each
$Hash.absorb(\text{pad}(pwd \,\|\, salt))$ ▷ Padding rule: 10^*1
for $row \leftarrow 1$ **to** $R-1$ **do**
  $M[row] \leftarrow Hash.squeeze_\rho(C \cdot b)$
end for

$R.C$

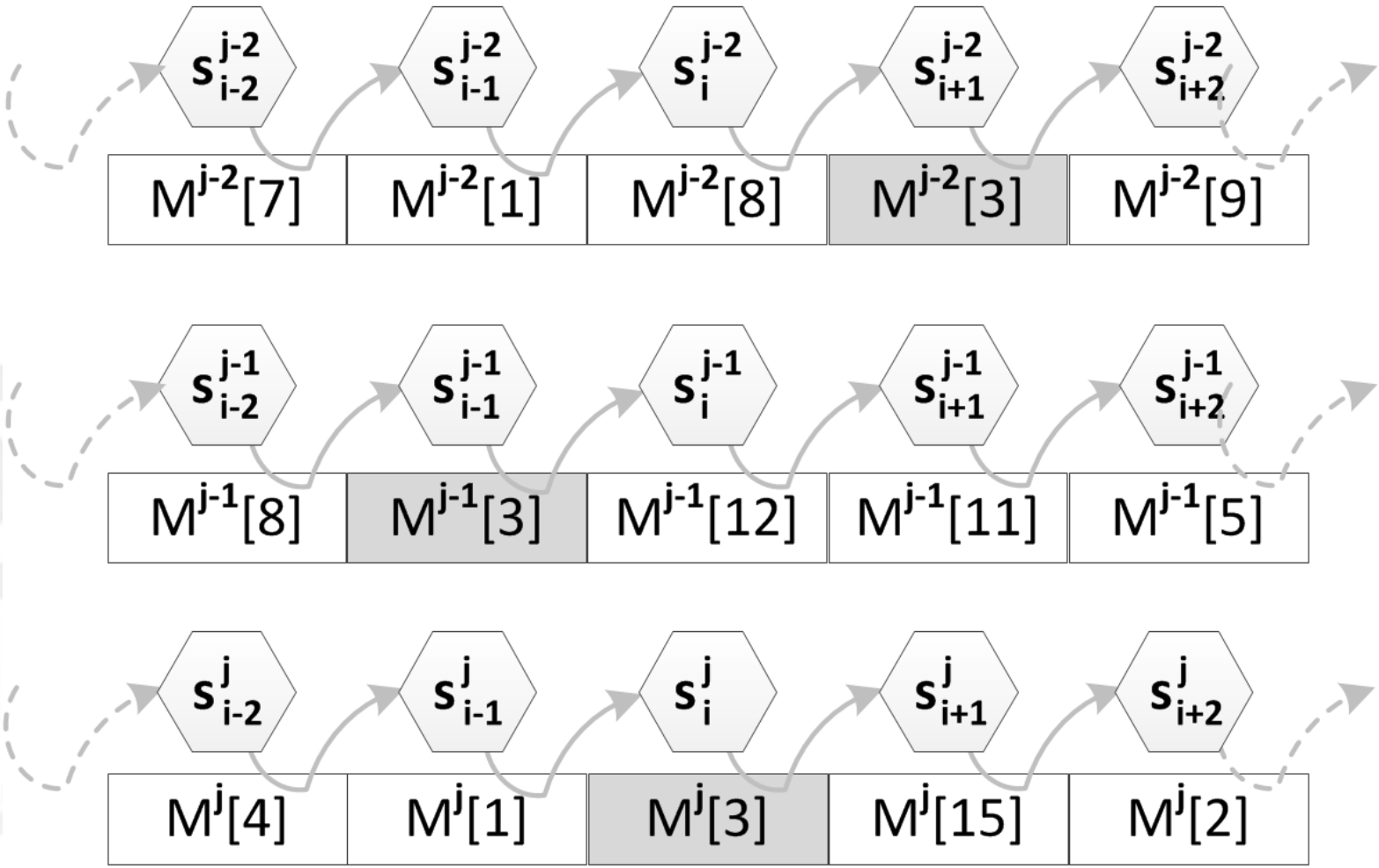
▷ **Wandering phase**: Iteratively overwrites blocks of the memory matrix
$row \leftarrow 0$
for $i \leftarrow 0$ **to** $T-1$ **do** ▷ **Time Loop**
  for $j \leftarrow 0$ **to** $R-1$ **do** ▷ **Rows Loop**: randomly visits R rows
    for $col \leftarrow 0$ **to** $C-1$ **do** ▷ **Columns Loop**: visits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

T

R

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

Lyra – Estágios Intermediários



Lyra – Estágios Intermediários

Algorithm The Lyra Algorithm.

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

Processamento ≈ (R+T).R.T/2
Memória ≈ R.(T-1)

1: ▷ **Setup phase**: Initializes a $(R \times C)$ memory matrix whose cells have b bits each
2: $Hash.absorb(\text{pad}(pwd \,\|\, salt))$ ▷ Padding rule: 10*1
3: **for** $row \leftarrow 1$ **to** $R-1$ **do**
4: $\quad M[row] \leftarrow Hash.squeeze_\rho(C \cdot b)$
5: **end for**
6: ▷ **Wandering phase**: Iteratively overwrites blocks of the memory matrix
7: $row \leftarrow 0$
8: **for** $i \leftarrow 0$ **to** $T-1$ **do** ▷ **Time Loop**
9: $\quad$ **for** $j \leftarrow 0$ **to** $R-1$ **do** ▷ **Rows Loop**: randomly visits R rows
10: $\quad\quad$ **for** $col \leftarrow 0$ **to** $C-1$ **do** ▷ **Columns Loop**: visits blocks in row
11: $\quad\quad\quad M[row][col] \leftarrow M[row][col] \oplus Hash.duplexing_\rho(M[row][col], b)$
12: $\quad\quad$ **end for**
13: $\quad\quad row \leftarrow Hash.duplexing((M[row][C-1] \bmod C), |row|) \bmod R$
14: $\quad$ **end for**
15: **end for**
16: ▷ **Wrap-up phase**: key computation
17: $Hash.absorb(\text{pad}(pwd \,\|\, salt))$ ▷ Uses the sponge's current state
18: $K \leftarrow Hash.squeeze(k)$
19: **return** K ▷ Outputs the k-long key

T

R

Lyra – Sem memória*

Algorithm The Lyra Algorithm.

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

$Processamento \approx R.(R/2)^T$

1: ▷ **Setup phase**: Initializes a $(R \times C)$ memory matrix whose cells have b bits each
2: $Hash.absorb(\text{pad}(pwd \,\|\, salt))$ ▷ Padding rule: 10^*1
3: **for** $row \leftarrow 1$ **to** $R-1$ **do**
4: $\quad M[row] \leftarrow Hash.squeeze_\rho(C \cdot b)$
5: **end for**
6: ▷ **Wandering phase**: Iteratively overwrites blocks of the memory matrix
7: $row \leftarrow 0$
8: **for** $i \leftarrow 0$ **to** $T-1$ **do** ▷ **Time Loop** — T
9: $\quad$ **for** $j \leftarrow 0$ **to** $R-1$ **do** ▷ **Rows Loop**: randomly visits R rows
10: $\quad\quad$ **for** $col \leftarrow 0$ **to** $C-1$ **do** ▷ **Columns Loop**: visits blocks in row
11: $\quad\quad\quad M[row][col] \leftarrow M[row][col] \oplus Hash.duplexing_\rho(M[row][col], b)$ — R
12: $\quad\quad$ **end for**
13: $\quad\quad row \leftarrow Hash.duplexing((M[row][C-1] \bmod C), |row|) \bmod R$
14: $\quad$ **end for**
15: **end for**
16: ▷ **Wrap-up phase**: key computation
17: $Hash.absorb(\text{pad}(pwd \,\|\, salt))$ ▷ Uses the sponge's current state
18: $K \leftarrow Hash.squeeze(k)$
19: **return** K ▷ Outputs the k-long key

Lyra – Resumo

Ataques					
	Sequencial (Padrão)		Estágios Intermediários		Sem memória*
Lyra	Memória $\mathcal{O}(R.C)$	Tempo $\mathcal{O}(R.T)$	Memória $\mathcal{O}(R.T)$	Tempo $\mathcal{O}(R^2.T + R.T^2)$	Tempo $\mathcal{O}(R^{T+1})$

Tabela: Complexidade dos ataques aplicáveis ao Lyra.

Resumo

Sejam,

- τ a quantidade de memória utilizada pelas variáveis do sistema;
- β os 4 KBytes de memória utilizados pelas S-Boxes e sub-chaves do algoritmo Blowfish [PM99].

Ataques					
	Sequencial (Padrão)		Estágios Intermediários		Sem memória*
PBKDF2	Memória $\mathcal{O}(\tau)$	Tempo $\mathcal{O}(l.T)$	Memória -	Tempo -	Tempo -
BCRYPT	$\mathcal{O}(\tau+\beta)$	$\mathcal{O}(2^{9+T})$	-	-	-
SCRYPT	$\mathcal{O}(R)$	$\mathcal{O}(R)$	-	-	$\mathcal{O}(R^2)$
Lyra	$\mathcal{O}(R.C)$	$\mathcal{O}(R.T)$	$\mathcal{O}(R.T)$	$\mathcal{O}(R^2.T+R.T^2)$	$\mathcal{O}(R^{T+1})$

Tabela: Complexidade dos ataques aplicáveis aos principais PHSs.

Agenda

1. Introdução
   - Entropia
   - Ataque de força bruta
   - Tabelas de consulta
2. Plataformas de Ataque
   - *Graphics Processing Units (GPUs)*
   - *Field Programmable Gate Arrays (FPGAs)*
3. Complexidade de Alguns Ataques
   - PBKDF2
   - BCRYPT
   - SCRYPT
   - Lyra
4. Considerações Finais
   - PHS ao longo dos anos
   - Funções utilizadas internamente
   - Conclusões

PHS ao longo dos anos

- 60's

armazenar: *''password''*

- 70's

armazenar: *hash(''password'')*

- final dos 70's, 80's e meados dos 90's

armazenar: *hash(''password'', salt)*

- 2000's ...

armazenar: *hash(''password'', salt, cost)*

Funções utilizadas internamente

- A segurança da função de derivação de chave está diretamente ligada à segurança da função utilizada internamente;
- A função de *hash* SHA-1 adotado pelo algoritmo PBKDF2 e a função de *hash* Salsa20/8 adotada pelo algoritmo SCRYPT **possuem vulnerabilidades** conhecidas [WYY05, AFK+08];
- Enquanto a função esponja BLAKE2 adotada pelo Lyra **permanece segura** [MQZ10].

Conclusões

- Esquemas de Hash de Senhas (PHSs) modernos permitem aos usuários legítimos **ajustar**, de acordo com sua necessidade, os **custos de memória e processamento** conforme o nível de segurança e os recursos disponíveis na plataforma de destino;
- Existem soluções modernas extremamente interessantes e viáveis:
  - **Argon2**
  - Catena
  - Lyra2
  - Makwa
  - yescrypt

Dúvidas?

Referências I

J-P. Aumasson, S. Fischer, S. Khazaei, W. Meier e C. Rechberger.

New features of latin dances: Analysis of Salsa, ChaCha, and Rumba.

Em *Fast Software Encryption*, volume 5084, páginas 470–488, Berlin, Heidelberg, 2008. Springer-Verlag.

W. Burr, D. Dodson e W. Polk.

Information security.

Recommendations of the National Institute of Standards and Technology, 2006.
`http://csrc.nist.gov/publications/nistpubs/800-63/SP800-63V1_0_2.pdf`.

A. Conklin, G. Dietrich e D. Walz.

Password-based authentication: A system perspective.

Em *Proc. of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04)*, volume 7 of *HICSS'04*, páginas 170–179, Washington, DC, USA, 2004. IEEE Computer Society.

Eric S. Chung, Peter A. Milder, James C. Hoe e Ken Mai.

Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs?

Em *Proc. of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO'43, páginas 225–236, Washington, DC, USA, 2010. IEEE Computer Society.

Yoginder S. Dandass.

Using FPGAs to parallelize dictionary attacks for password cracking.

Em *Proc. of the 41st Annual Hawaii International Conference on System Sciences (HICSS 2008)*, páginas 485–485. IEEE, 2008.

Markus Dürmuth, Tim Güneysu e Markus Kasper.

Evaluation of standardized password-based key derivation against parallel processing platforms.

Em *Computer Security–ESORICS 2012*, volume 7459 of *Lecture Notes in Computer Science*, páginas 716–733. Springer Berlin Heidelberg, 2012.

Referências II

Jeremy Fowers, Greg Brown, Patrick Cooke e Greg Stitt.

A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications.
Em *Proc. of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, FPGA'12, páginas 47–56, New York, NY, USA, 2012. ACM.

D. Florencio e C. Herley.

A large scale study of web password habits.
Em *Proc. of the 16th international conference on World Wide Web*, páginas 657–666, Alberta, Canada, 2007.

Khronos Group.

The OpenCL Specification – Version 1.2, 2012.

Athanasios P. Kakarountas, Haralambos Michail, Athanasios Milidonis, Costas E. Goutis e George Theodoridis.

High-speed FPGA implementation of secure hash algorithm for IPSec and VPN applications.
The Journal of Supercomputing, 37(2):179–195, 2006.

Mao Ming, He Qiang e Shaokun Zeng.

Security analysis of BLAKE-32 based on differential properties.
Em *Computational and Information Sciences (ICCIS), 2010 International Conference on*, páginas 783–786. IEEE, 2010.

Nvidia.

CUDA C programming guide.
http://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2012.

Nvidia.

Tesla Kepler family product overview.
http://www.nvidia.com/content/tesla/pdf/Tesla-KSeries-Overview-LR.pdf, 2012.

Referências III

N. Provos e D. Mazières.

A future-adaptable password scheme.

Em *Proc. of the FREENIX track: 1999 USENIX annual technical conference*, 1999.

SciEngines.

Rivyera s3-5000.

http://sciengines.com/products/computers-and-clusters/rivyera-s3-5000.html.

Martijn Sprengers.

GPU-based password cracking: On the security of password hashing schemes regarding advances in graphics processing units.

Dissertação de Mestrado, Radboud University Nijmegen, 2011.

Xiaoyun Wang, Yiqun Lisa Yin e Hongbo Yu.

Finding collisions in the full SHA-1.

Em *Advances in Cryptology - CRYPTO 2005: 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings*, volume 3621 of *Lecture Notes in Computer Science*, páginas 17–36. Springer, 2005.

Créditos

- A imagem utilizada como plano de fundo em todos os slides segue a licença de uso que consta em `http://www.unir.br` – © Fundação Universidade Federal de Rondônia;
- A imagem utilizada no slide de Dúvidas segue a licença de uso que consta em `http://pixabay.com` – © Creative Commons.