Security of Key Derivation Functions

Ewerton Rodrigues Andrade

ewerton@usp.br

Advisor: Prof. Dr. Marcos Antonio Simplicio Junior

Poli - Escola Politécnica
USP - Universidade de São Paulo

PCS5734 - Segurança da Informação: Algoritmos e Protocolos

Responsibles for course:

Prof. Dr. Paulo S. L. M. Barreto

Prof. Dr. Marcos Antonio Simplicio Junior

October 7th, 2013

Agenda

1. ~~Introduction~~
   - ~~Entropy~~
   - ~~Brute-force attacks~~
   - ~~Rainbow Tables~~
2. Attack platforms
   - Graphics Processing Units (GPUs)
   - Field Programmable Gate Arrays
3. Complexity of some attacks
   - PBKDF2
   - BCRYPT
   - SCRYPT
   - Lyra
4. Conclusions
   - Internal functions
   - Conclusions

Agenda

1. Introduction
   - Entropy
   - Brute-force attacks
   - Rainbow Tables
2. Attack platforms
   - Graphics Processing Units (GPUs)
   - Field Programmable Gate Arrays
3. Complexity of some attacks
   - PBKDF2
   - BCRYPT
   - SCRYPT
   - Lyra
4. Conclusions
   - Internal functions
   - Conclusions

Agenda

Attack platforms

- The most dangerous threats faced by KDFs comes from platforms that benefit from **economies of scale**, especially when cheap, **massively parallel** hardware is available;
- The most prominent examples of such platforms are Graphics Processing Units (**GPUs**) and custom hardware synthesized from **FPGAs** [DGK12].

GPUs – Evolution

- Following the increasing demand for **high-definition real-time rendering**, Graphics Processing Units (GPUs) have traditionally carried a large number of processing cores, boosting its parallelization capability;
- Only more recently, however, GPUs evolved from **specific platforms into devices for universal computation** and started to give support to standardized languages that help harnessing their computational power, such as CUDA [Nvi12a] and OpenCL [Khr12];
- As a result, they became more intensively employed for more general purposes, including **password cracking** [Spr11, DGK12].

GPUs – Examples

NVidia Tesla K20X [Nvi12b]:

- 2.688 cores operating at 732 MHz;
- 6 GB of shared DRAM, with a bandwidth of 250 GB/s.

NVidia GT540M (the vga card of my notebook):

- 96 cores operating at 900 MHz;
- 2 GB of shared DRAM, with a bandwidth of 28,8 GB/s.

GPUs – Possible scenario

Assume a scenario where the adversary have of a NVidia Tesla K20X.

- In case the passwords are stored using some KDF applied to the plaintext, and the KDF take only **2 ms to run**, consuming only **0.5 MB of memory**.

GPUs – Possible scenario

Assume a scenario where the adversary have of a NVidia Tesla K20X.

- In case the passwords are stored using some KDF applied to the plaintext, and the KDF take only **2 ms to run**, consuming only **0.5 MB of memory**.

In this scenario it is easy to conceive that the adversary will test 2.688 passwords every two ms. Resulting in 1.344.000 passwords tested per second, or 4.838.400.000 $\approx 2^{32,17}$ passwords tested per hour.

GPUs – Possible scenario

Assume a scenario where the adversary have of a NVidia Tesla K20X.

- In case the passwords are stored using some KDF applied to the plaintext, and the KDF take only **2 ms to run**, consuming only **0.5 MB of memory**.

In this scenario it is easy to conceive that the adversary will test 2.688 passwords every two ms. Resulting in 1.344.000 passwords tested per second, or 4.838.400.000 $\approx 2^{32,17}$ passwords tested per hour.

However, **if a sequential KDF requires 20 MB of DRAM**, the maximum number of cores that could be used simultaneously becomes 300, only 11 % of the total available.

FPGA

- An FPGA is a collection of configurable logic blocks wired together and with memory elements, forming a programmable and high-performance integrated circuit;
- As such devices are configured to perform a specific task, **they can be highly optimized** for its purpose (e.g., using pipelining [Dan08, KMM^{+}06]);
- Furthermore, When compared to GPUs, FPGAs may also be advantageous due to the latter's **considerably lower energy consumption** [CMHM10, FBCS12].

FPGA – A recent example of password-cracking [DGK12]

- The **small memory usage** of the PBKDF2 algorithm, as most of the underlying SHA-2 processing is performed using the device's memory cache (much faster than DRAM) [DGK12, Sec. 4.2];
- Dürmuth *et al*, using a RIVYERA S3-5000 cluster [Sci] with 128 FPGAs, reported a throughput of 356.352 passwords tested per second in an architecture having 5.376 password processed in parallel.

FPGA – A recent example of password-cracking [DGK12]

- The **small memory usage** of the PBKDF2 algorithm, as most of the underlying SHA-2 processing is performed using the device's memory cache (much faster than DRAM) [DGK12, Sec. 4.2];
- Dürmuth *et al*, using a RIVYERA S3-5000 cluster [Sci] with 128 FPGAs, reported a throughput of 356.352 passwords tested per second in an architecture having 5.376 password processed in parallel.

However – as in the GPU's example – **if a sequential KDF requires 20 MB of DRAM** in place of PBKDF2, the resulting throughput would presumably be much lower, especially considering that the FPGAs employed can have up to 64 MB of DRAM [Sci] and, thus, up to 3 passwords can be processed in parallel rather than 5.376.

Agenda

PBKDF2

Algorithm PBKDF2.

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

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

Where:
k represents the desired size for the key generated by PBKDF2; and

h represents the size of the output of the function used internally.

PBKDF2

Algorithm PBKDF2.

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

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

Where:
k represents the desired size for the key generated by PBKDF2; and
h represents the size of the output of the function used internally.

PBKDF2 – Summary

Let,

- τ be the amount of memory used by the system variables.

Attacks					
	Sequential (Default)		Intermediate states		Memory-free*
PBKDF2	Memory	Time	Memory	Time	Time
	$\mathcal{O}(\tau)$	$\mathcal{O}(l.T)$	-	-	-

Table: Complexity of attacks applicable to 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

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

```
s ← InitState()   ▷ Copies the digits of π into the sub-keys and S-boxes S_i
s ← ExpandKey(s, salt, pwd)
for i ← 1 to 2^T do
    s ← ExpandKey(s, 0, salt)  ;  s ← ExpandKey(s, 0, pwd)
end for
ctext ← "OrpheanBeholderScryDoubt"
for i ← 1 to 64 do { ctext ← BlowfishEncrypt(s, ctext) } end for
return T || salt || ctext
function ExpandKey(s, salt, pwd)
    for i ← 1 to 32 do { P_i ← P_i ⊕ pwd[32 * (i − 1) ... 32 * i − 1] } end for
    for i ← 1 to 9 do
        temp ← BlowfishEncrypt(s, salt[64 * (i − 1) ... 64 * i − 1])
        P_{0+(i−1)*2} ← temp[0 ... 31]  ;  P_{1+(i−1)*2} ← temp[32 ... 64]
    end for
    for i ← 1 to 4 do
        for j ← 1 to 128 do
            temp ← BlowfishEncrypt(s, salt[64 * (j − 1) ... 64 * j − 1])
            S_i[(j − 1) * 2] ← temp[0 ... 31]  ;  S_i[1 + (j − 1) * 2] ← temp[32 ... 63]
        end for
    end for
    return s
end function
```

2^T

$\approx 2^9.2^T + 2^6$

2^6

2^5

9

2^2

$2^2.2^7$

BCRYPT – Summary

Let,

- τ be the amount of memory used by the system variables;
- β be the 4 KBytes of memory used by the S-Boxes and sub-keys of Blowfish algorithm [PM99].

Attacks					
	Sequential (Default)		Intermediate states		Memory-free*
BCRYPT	Memory	Time	Memory	Time	Time
	$\mathcal{O}(\tau + \beta)$	$\mathcal{O}(2^{9+T})$	-	-	-

Table: Complexity of attacks applicable to 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 – Sequential (Default)

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

Memory cost ≈ p.R.2r
Processing cost ≈ p.R.2r

$(B_0 ... B_{p-1}) \leftarrow PBKDF2_{HMAC-SHA-256}(pwd, salt, 1, p \cdot b)$
for $i \leftarrow 0$ **to** $p-1$ **do** { $B_i \leftarrow ROMix(B_i, R)$ } **end for** — *p*
$K \leftarrow PBKDF2_{HMAC-SHA-256}(pwd, B_0 || B_1 || ... || B_{p-1}, 1, k)$
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 — *R*
    $V_i \leftarrow X$; $X \leftarrow BlockMix(X)$
  end for
  for $i \leftarrow 0$ **to** $R-1$ **do** ▷ Reads random positions of V — *R*
    $j \leftarrow Integerify(X) \bmod R$; $X \leftarrow BlockMix(X \oplus V_j)$
  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** — *2r*
  return $(Y_0, Y_2, ..., Y_{2r-2}, Y_1, Y_3, Y_{2r-1})$
end function

SCRYPT – Memory-free*

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 · 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}) ← PBKDF2_{HMAC-SHA-256}(pwd, salt, 1, p · b)
for i ← 0 to p − 1 do { B_i ← ROMix(B_i, R) } end for                    p
K ← 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 ← B
    for i ← 0 to R − 1 do    ▷ Initializes memory array V
        V_i ← X ; X ← BlockMix(X)                                          R
    end for
    for i ← 0 to R − 1 do    ▷ Reads random positions of V
        j ← Integerify(X) mod R ; X ← BlockMix(X ⊕ V_j)                    R
    end for
    return X
end function
function BlockMix(B)    ▷ Hash function with (b-long) inputs/outputs
    Z ← B_{2r-1}    ▷ r = b/2h, where h = 512 for Salsa20/8
    for i ← 0 to 2r − 1 do { Z ← Hash(Z ⊕ B_i) ; Y_i ← Z } end for        2r
    return (Y_0, Y_2, ..., Y_{2r-2}, Y_1, Y_3, Y_{2r-1})
end function
```

Processing cost ≈ p.R.R.2r

SCRYPT – Summary

	Attacks				
	Sequential (Default)		Intermediate states		Memory-free*
SCRYPT	Memory	Time	Memory	Time	Time
	$\mathcal{O}(R)$	$\mathcal{O}(R)$	-	-	$\mathcal{O}(R^2)$

Table: Complexity of attacks applicable to 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 in the Setup and Wandering phases
INPUT: pwd ▷ The password
INPUT: $salt$ ▷ A random salt
INPUT: T ▷ Time cost, in number of iterations
INPUT: R ▷ Number of rows in the memory matrix
INPUT: C ▷ Number of columns in the memory matrix
INPUT: k ▷ The desired key length, in bits
OUTPUT: K ▷ The password-derived k-long key

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

Lyra – Sequential (Default)

Algorithm The Lyra Algorithm.

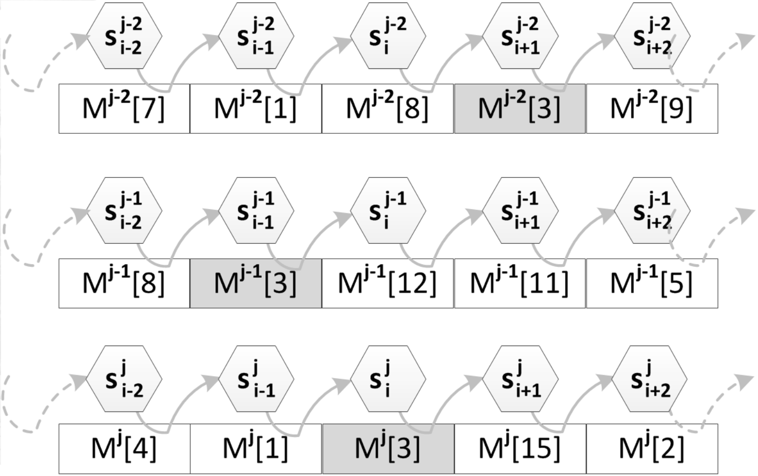
```
PARAM: Hash    ▷ Sponge with block size b (in bits) and underlying permutation f
PARAM: ρ    ▷ Number of rounds of f in the Setup and Wandering phases
INPUT: pwd    ▷ The password
INPUT: salt    ▷ A random salt
INPUT: T    ▷ Time cost, in number of iterations
INPUT: R    ▷ Number of rows in the memory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

▷ Setup: Initializes a (R × C) memory matrix whose cells have b bits each
Hash.absorb(pad(salt || pwd))    ▷ Padding rule: 10*1
M[0] ← Hash.squeeze_ρ(C · b)
for row ← 1 to R − 1 do
    for col ← 0 to C − 1 do
        M[row][col] ← Hash.duplexing_ρ(M[row − 1][col], b)
    end for
end for
▷ Wandering: Iteratively overwrites blocks of the memory matrix
row ← 0
for i ← 0 to T − 1 do    ▷ Time Loop
    for j ← 0 to R − 1 do    ▷ Rows Loop: randomly visits R rows
        for col ← 0 to C − 1 do    ▷ Columns Loop: visits blocks in row
            M[row][col] ← M[row][col] ⊕ Hash.duplexing_ρ(M[row][col], b)
        end for
        col ← M[row][C − 1] mod C
        row ← Hash.duplexing(M[row][col], |R|) mod R
    end for
end for
▷ Wrap-up: key computation
Hash.absorb(pad(salt))    ▷ Uses the sponge's current state
K ← Hash.squeeze(k)
return K    ▷ Outputs the k-long key
```

R.C (setup loop, lines 4–8)

T (time loop, lines 11–19)

R (rows loop, lines 12–18)

Lyra – Intermediate states



Lyra – Intermediate states

Algorithm The Lyra Algorithm.

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

Processing cost ≈ (R+T).R.T/2
Memory cost ≈ R.(T-1)

```
▷ Setup: Initializes a (R × C) memory matrix whose cells have b bits each
Hash.absorb(pad(salt || pwd))   ▷ Padding rule: 10*1
M[0] ← Hash.squeeze_ρ(C · b)
for row ← 1 to R − 1 do
    for col ← 0 to C − 1 do
        M[row][col] ← Hash.duplexing_ρ(M[row − 1][col], b)
    end for
end for
▷ Wandering: Iteratively overwrites blocks of the memory matrix
row ← 0
for i ← 0 to T − 1 do   ▷ Time Loop                                  T
    for j ← 0 to R − 1 do   ▷ Rows Loop: randomly visits R rows
        for col ← 0 to C − 1 do   ▷ Columns Loop: visits blocks in row
            M[row][col] ← M[row][col] ⊕ Hash.duplexing_ρ(M[row][col], b)
        end for                                                     R
        col ← M[row][C − 1] mod C
        row ← Hash.duplexing(M[row][col], |R|) mod R
    end for
end for
▷ Wrap-up: key computation
Hash.absorb(pad(salt))   ▷ Uses the sponge's current state
K ← Hash.squeeze(k)
return K   ▷ Outputs the k-long key
```

Lyra – Memory-free*

Algorithm The Lyra Algorithm.

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

Processing cost $\approx R.(R/2)^T$

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

T

R

Lyra – Summary

Attacks					
	Sequential (Default)		Intermediate states		Memory-free*
Lyra	Memory	Time	Memory	Time	Time
	$\mathcal{O}(R.C)$	$\mathcal{O}(R.T)$	$\mathcal{O}(R.T)$	$\mathcal{O}(R^2.T + R.T^2)$	$\mathcal{O}(R^{T+1})$

Table: Complexity of attacks applicable to Lyra.

Summary

Let,

- τ be the amount of memory used by the system variables;
- β be the 4 KBytes of memory used by the S-Boxes and sub-keys of Blowfish algorithm [PM99].

Attacks					
	Sequential (Default)		Intermediate states		Memory-free*
PBKDF2	Memory $\mathcal{O}(\tau)$	Time $\mathcal{O}(l.T)$	Memory -	Time -	Time -
BCRYPT	$\mathcal{O}(\tau+\beta)$	$\mathcal{O}(2^{9+T})$	-	-	-
SCRYPT	$\mathcal{O}(R)$	$\mathcal{O}(R)$	-	-	$\mathcal{O}(R^2)$
Lyra	$\mathcal{O}(R.C)$	$\mathcal{O}(R.T)$	$\mathcal{O}(R.T)$	$\mathcal{O}(R^2.T+R.T^2)$	$\mathcal{O}(R^{T+1})$

Table: Complexity of attacks applicable to the main KDFs.

Agenda

Internal functions

- The security of the key derivation function is directly linked to the security of the function used internally;
- The hash function SHA-1 adopted by the PBKDF2 algorithm and the hash function Salsa20/8 adopted by the Scrypt algorithm **have known vulnerabilities** [WYY05, AFK^{+}08];
- While the sponge function BLAKE2 adopted by Lyra **remains safe** [MQZ10].

Conclusions

- Lyra is Lyra, a password-based key derivation scheme that allows legitimate users to **fine tune memory and processing costs** according to the desired level of security and resources available in the target platform;
- Moreover, the combination of a strictly sequential design, the high costs of exploring memory-processing trade-offs, and the ability to raise the memory usage beyond what is attainable with similar-purpose solutions (e.g., scrypt) for a similar security level and processing time, make **Lyra an appealing KDF alternative**.

Questions?

References 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.

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.

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.

Khronos Group.
The OpenCL Specification – Version 1.2, 2012.

References II

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.

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.

References III

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.

Copyright