

Implementação da função Lyra2 para arquiteturas modernas

Aluno: Guilherme P. Gonçalves

Orientador: Diego de Freitas Aranha

Novembro de 2014

1 Introdução

A forma mais comum de autenticação de usuários em sistemas computacionais atualmente é o uso de senhas. Nesse paradigma, o usuário é responsável por escolher uma senha ao se registrar, que deve permanecer secreta, e o sistema verifica, a cada acesso, se o usuário conhece a senha correta. Isso implica, é claro, que a senha ficará armazenada no sistema de alguma forma.

Como medida de segurança, recomenda-se não armazenar a senha conforme fornecida pelo usuário, mas sim um *hash* dela, produzido por uma função de *hash* de senhas. O *hash* é uma sequência pseudoaleatória de bits tal que, dado o *hash* h de uma senha s , deve ser computacionalmente difícil descobrir qualquer senha (incluindo s) cujo *hash* também seja h – uma propriedade conhecida como resistência ao cálculo de pré-imagem.

Assim, em um procedimento de autenticação moderno, o sistema computa o *hash* da senha provida pelo usuário e o compara com o *hash* que havia sido armazenado para aquele usuário ao registrá-lo. Caso os *hashes* sejam iguais, o acesso é autorizado.

A partir disso, muitas técnicas foram introduzidas para dificultar ataques de força-bruta – aqueles em que um atacante faz tentativas sucessivas de adivinhar uma senha – sem onerar usuários exigindo que criem e memorizem senhas longas e suficientemente entrópicas.

No campo das funções de *hash* de senhas, isso significa a inclusão de parâmetros de tempo e espaço de memória mínimos a serem utilizados pela operação de *hash*. O parâmetro de tempo impõe um limite à velocidade com que um atacante pode fazer tentativas sequenciais, enquanto o de espaço visa a proteger contra ataques que empregam chips dedicados ou GPUs, caracterizados pelo paralelismo mas escassez de memória.

Neste trabalho final de graduação, produzimos uma implementação da função de *hash* de senhas Lyra2, proposta por pesquisadores brasileiros e candidata ao Password Hashing Competition, aproveitando conjuntos de instruções presentes em arquiteturas modernas. Acredita-se que a Lyra2 possua as propriedades de segurança exigidas de uma função de *hash*, e sua especificação inclui parâmetros de espaço e tempo independentes a serem respeitados por uma implementação correta.

A implementação resultante¹ é compatível com a de referência e competitiva com ela em termos de desempenho. Colaboramos ainda com o desenvolvimento da especificação da Lyra2 (Simplicio Jr. et al.), na medida em que este trabalho trouxe à tona diversas pequenas falhas e inconsistências no documento e na implementação de referência.

2 Notação

Para o restante deste documento, os símbolos da primeira coluna da tabela a seguir serão usados com o significado dado pela segunda coluna.

¹Disponível em <https://github.com/guilherme-pg/lyra2>

Símbolo	Significado
\oplus	XOR bit-a-bit
$\lfloor \cdot \rfloor_l$	Truncagem para l bits
$\ggg n$	Rotação n bits à direita
\boxplus	Adição sem carry entre palavras
\parallel	Concatenação
$len(n)$	Tamanho em bytes de n
$LSW(n)$	Palavra menos significativa de n
$rotRt(n)$	Rotação Rt bits à direita de n
$rotRt^m(n)$	Rotação $m \cdot Rt$ bits à direita de n

3 A função Lyra2

A Lyra2 se baseia na construção de esponja, uma forma geral para a geração de funções de *hash* seguras com entradas e saídas de tamanhos arbitrários a partir de uma função de compressão e um esquema de *padding*.

De fato, o cálculo da Lyra2 para uma determinada entrada pode ser descrito em alto nível como a aplicação iterada de operações de esponja, a serem descritas a seguir, a dados mantidos em uma matriz de estado do algoritmo. Consequentemente, conjectura-se que as propriedades de segurança da Lyra2 decorrem tanto da segurança da esponja subjacente, quanto da escolha criteriosa de operações de esponja empregadas, que dificulta a paralelização do algoritmo.

3.1 A construção de esponja

A definição canônica de uma esponja (Bertoni et al.) descreve sua operação em termos de duas etapas – *absorbing* e *squeezing* –, sendo que na primeira a esponja incorpora a entrada a seu estado interno, e na segunda produz uma saída do tamanho desejado baseada nesse estado. Assim, ao final de uma etapa de *squeezing*, a esponja terá produzido um *hash* pseudoaleatório de sua entrada, e pode ser restaurada a seu estado original para uma aplicação posterior.

A Figura 1 ilustra a construção de esponja. Nela, a entrada M é dividida em blocos após o *padding* e absorvida para gerar a saída Z . f é uma função de compressão. A linha tracejada separa as etapas de *absorbing* (esquerda) e *squeezing* (direita), e o estado interno da esponja é dividido em duas partes com tamanhos r (*rate*) e c (*capacity*) bits.²

Pode-se definir, ainda, uma construção similar à da esponja, denominada *duplex*, em que se mantém o estado interno entre aplicações. Nela, as operações de *absorb* e *squeeze* são substituídas por uma operação de *duplexing*, na qual um bloco de entrada é absorvido e um bloco de saída é imediatamente produzido. A Figura 2 ilustra a construção de *duplex*.³

A Lyra2 utiliza uma versão modificada da construção de *duplex*, em que as operações de *absorb* e *squeeze* também são suportadas, e podem ser efetuadas independentemente.

²©2007-2013 Guido Bertoni, Joan Daemen, Michaël Peeters and Gilles Van Assche / <http://sponge.noekeon.org/> / CC-BY-3.0

³©2007-2013 Guido Bertoni, Joan Daemen, Michaël Peeters and Gilles Van Assche / <http://sponge.noekeon.org/> / CC-BY-3.0

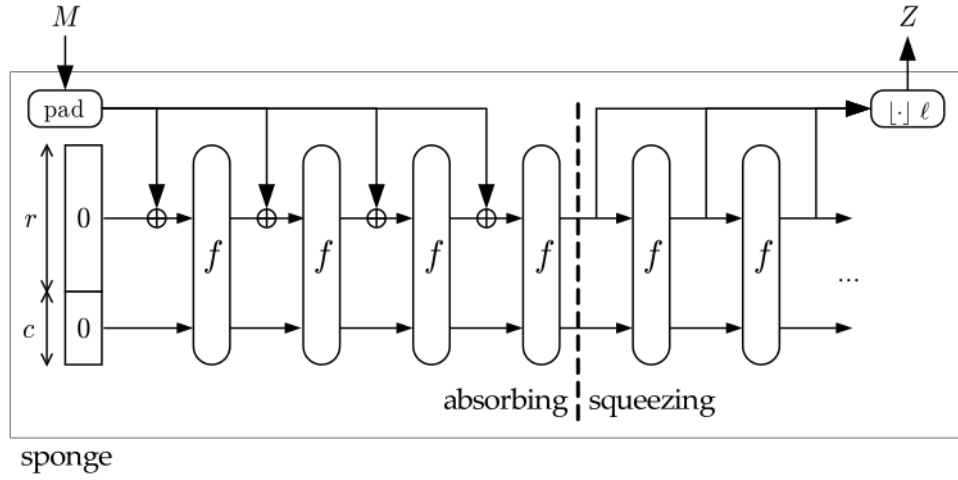


Figura 1: A construção de esponja.

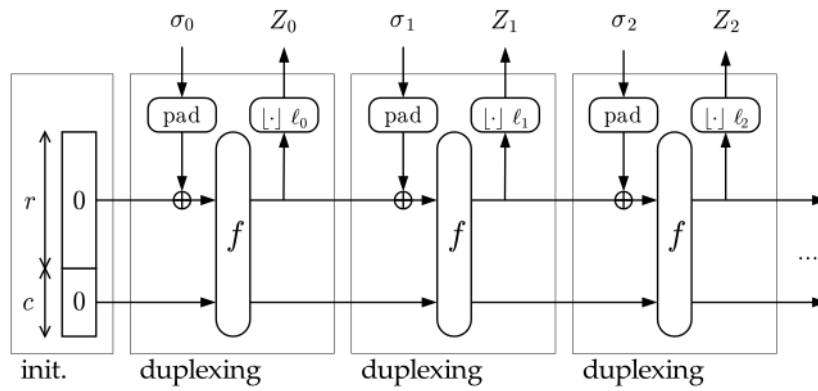


Figura 2: A construção de duplex.

3.2 A função de compressão

Conforme visto anteriormente, a construção de esponja depende de uma função f , denominada função de compressão. Para a esponja usada na Lyra2, a função de compressão é uma versão levemente adaptada da função G que integra a função de *hash* BLAKE2b (Neves et al.) . A função $G(a, b, c, d)$ utilizada é dada por:

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

No contexto do algoritmo BLAKE2b, essa função é aplicada repetidamente a uma matriz de estado 4×4 de inteiros de 64 bits, primeiramente aos elementos de cada uma das colunas, depois aos de cada uma das diagonais. Essas oito aplicações constituem uma *rodada*, e o algoritmo prevê transformações de seu estado em conjuntos de 12 rodadas por vez.

Na Lyra2, o estado da esponja contém 128 bytes e é visto como uma matriz de estado linearizada, e as rodadas são definidas de forma análoga. No entanto, a fim de melhorar o desempenho do algoritmo, a maior parte das compressões efetuadas pela esponja não utiliza $\rho_{max} = 12$ rodadas, mas sim $\rho < \rho_{max}$ (na prática, utiliza-se $\rho = 1$). Tais operações com apenas ρ rodadas são denominadas operações *reduzidas* da esponja.

3.3 O algoritmo Lyra2

A Figura 3 contém o pseudocódigo do algoritmo Lyra2 implementado para este trabalho.

Basicamente, o algoritmo consiste de três etapas: *bootstrapping*, *setup* e *wandering*. Todas elas trabalham sobre uma matriz de estado de $R \times C$ blocos de b bits, onde R e C são, portanto, parâmetros de espaço.

O tamanho dos blocos é definido de forma que possam ser absorvidos sem *padding* pela esponja utilizada. Embora um bloco de $b = r$ bits pareça natural tendo em vista as Figuras 1 e 2, a especificação sugere aumentar r após a primeira absorção (linha 3 da Figura 3). Na prática, tanto nosso código quanto o de referência utilizam um r inicial de 512 bits, e utilizam posteriormente um novo $r = b = 768$ bits.

A fase de *bootstrapping* inicializa a esponja com o vetor de inicialização da função de compressão, e então absorve os parâmetros de entrada. A fase de *setup* inicializa a matriz de estado, e a fase de *wandering* aplica operações reduzidas de esponja a células pseudoaleatórias dessa matriz. Nessa fase, o parâmetro T controla o número de iterações a serem feitas, e, portanto, o tempo utilizado.

O tempo total de execução do algoritmo é dado por $(T + 1) \cdot R \cdot C \cdot \frac{\rho}{\rho_{max}}$ vezes o tempo de execução da função de compressão da esponja, de forma que, ainda que o tempo de execução seja limitado inferiormente pelos parâmetros de memória R e C , ainda é possível aumentá-lo mantendo o consumo de memória constante, através do parâmetro T .

4 Descrição da implementação

Nossa implementação utiliza a linguagem C e possui duas variantes: uma utilizando as instruções vetoriais do conjunto SSE2, e outra utilizando as do conjunto AVX2. Como o código de referência possui versões genérica

Algorithm 2 The Lyra2 Algorithm.

PARAM: H \triangleright Sponge with block size b (in bits) and underlying permutation f
PARAM: H_ρ \triangleright Reduced-round sponge for use in the Setup and Wandering phases (e.g., f with ρ)
PARAM: Rt \triangleright Number of bits to be used in rotations (recommended: a multiple of the machine's word size, W)
INPUT: pwd \triangleright The password
INPUT: $salt$ \triangleright A salt
INPUT: T \triangleright Time cost, in number of iterations ($T \geq 1$)
INPUT: R \triangleright Number of rows in the memory matrix (recommended: a power of two)
INPUT: C \triangleright Number of columns in the memory matrix (recommended: $C \cdot \rho \geq \rho_{max}$)
INPUT: k \triangleright The desired key length, in bits
OUTPUT: K \triangleright The password-derived k -long key

1: \triangleright **BOOTSTRAPPING PHASE:** Initializes the sponge's state and local variables
2: $params \leftarrow len(k) \parallel len(pwd) \parallel len(salt) \parallel T \parallel R \parallel C$ \triangleright Byte representation of input parameters (others can be added)
3: $H.absorb(pad(pwd \parallel salt \parallel params))$ \triangleright Padding rule: 10^*1 . Password can be overwritten after this point
4: $gap \leftarrow 1$; $stp \leftarrow 1$; $wnd \leftarrow 2$ \triangleright Initializes visitation step and window
5: $prev^0 \leftarrow 2$; $row^1 \leftarrow 1$; $prev^1 \leftarrow 0$
6: \triangleright **SETUP PHASE:** Initializes a $(R \times C)$ memory matrix, it's cells having b bits each
7: **for** ($col \leftarrow 0$ **to** $C-1$) **do** $\{M[0][C-1-col] \leftarrow H_\rho.squeeze(b)\}$ **end for** \triangleright Initializes $M[0]$
8: **for** ($col \leftarrow 0$ **to** $C-1$) **do** $\{M[1][C-1-col] \leftarrow M[0][col] \oplus H_\rho.duplex(M[0][col], b)\}$ **end for** \triangleright Initializes $M[1]$
9: **for** ($col \leftarrow 0$ **to** $C-1$) **do** \triangleright Initializes $M[2]$ and updates $M[0]$
10: $rand \leftarrow H_\rho.duplex(M[0][col] \boxplus M[1][col])$
11: $M[2][C-1-col] \leftarrow M[1][col] \oplus rand$
12: $M[0][col] \leftarrow M[0][col] \oplus rotRt(rand)$ $\triangleright rotRt()$: right rotation by L bits (e.g., 1 or more words)
13: **end for**
14: **for** ($row^0 \leftarrow 3$ **to** $R-1$) **do** \triangleright **Filling Loop:** initializes remainder rows
15: **for** ($col \leftarrow 0$ **to** $C-1$) **do** \triangleright **Columns Loop:** $M[row^0]$ is initialized, while $M[row^1]$ is updated
16: $rand \leftarrow H_\rho.duplex(M[row^1][col] \boxplus M[prev^0][col] \boxplus M[prev^1][col], b)$
17: $M[row^0][C-1-col] \leftarrow M[prev^0][col] \oplus rand$
18: $M[row^1][col] \leftarrow M[row^1][col] \oplus rotRt(rand)$
19: **end for**
20: $prev^0 \leftarrow row^0$; $prev^1 \leftarrow row^1$; $row^1 \leftarrow (row^1 + stp) \bmod wnd$ \triangleright Picks rows to be revisited in next loop
21: **if** ($row^1 = 0$) **then** \triangleright Window fully revisited
22: $stp \leftarrow wnd + gap$; $wnd \leftarrow 2 \cdot wnd$; $gap \leftarrow -gap$ \triangleright Doubles window size and roughly doubles step
23: **end if**
24: **end for**
25: \triangleright **WANDERING PHASE:** Iteratively overwrites pseudorandom cells of the memory matrix
26: **for** ($\tau \leftarrow 1$ **to** T) **do** \triangleright **Time Loop**
27: **for** ($i \leftarrow 0$ **to** $R-1$) **do** \triangleright **Visitation Loop:** $2R$ rows revisited in pseudorandom fashion
28: **for** ($d \leftarrow 0$ **to** 1) **do** $\{row^d \leftarrow (LSW(rotRt^d(rand))) \bmod R\}$ **end for** \triangleright Picks pseudorandom rows
29: **for** ($col \leftarrow 0$ **to** $C-1$) **do** \triangleright **Columns Loop:** updates each $M[row^d]$
30: **for** ($d \leftarrow 2$ **to** 3) **do** $\{col^d \leftarrow (LSW(rotRt^d(rand))) \bmod C\}$ **end for** \triangleright Picks pseudorandom columns
31: $rand \leftarrow H_\rho.duplex(M[row^0][col] \boxplus M[row^1][col] \boxplus M[prev^0][col^0] \boxplus M[prev^1][col^1], b)$
32: **for** ($d \leftarrow 0$ **to** 1) **do**
33: $M[row^d][col] \leftarrow M[row^d][col] \oplus rotRt^d(rand)$ \triangleright Updates the d pseudorandom rows
34: **end for**
35: **end for** \triangleright End of Columns Loop
36: **for** ($d \leftarrow 0$ **to** 1) **do** $\{prev^d \leftarrow row^d\}$ **end for** \triangleright Next iteration revisits most recently updated rows
37: **end for** \triangleright End of Visitation Loop
38: **end for** \triangleright End of the Time Loop
39: \triangleright **WRAP-UP PHASE:** key computation
40: $H.absorb(M[row^0][col^0])$ \triangleright Absorbs a final column with the full-round sponge
41: $K \leftarrow H.squeeze(k)$ \triangleright Squeezes k bits with the full-round sponge
42: **return** K \triangleright Provides k -long bitstring as output

Figura 3: O algoritmo Lyra2 implementado neste trabalho.

(sem instruções vetoriais) e SSE2, cabe clarificar que, para o restante deste documento, e em particular na seção 5, quaisquer menções a nossa implementação ou à de referência se referem às respectivas versões SSE2 (a não ser, é claro, que nossa versão AVX2 esteja sendo discutida).

A versão SSE2 de nossa implementação, embora funcionalmente equivalente à de referência, é baseada em decisões de design levemente diferentes. Além de prezar pelo desempenho, nosso trabalho foi desenvolvido com particular cuidado pela legibilidade, facilitando eventuais auditorias do código, e portabilidade, evitando-se utilizar extensões específicas de determinados compiladores e aderindo-se de forma bastante estrita ao padrão C99.

Uma diferença particularmente perceptível entre as implementações é que, enquanto a de referência utiliza largamente *intrinsics* para obter controle fino sobre as instruções geradas, nossa implementação mantém os operadores de alto nível da linguagem C e delega ao compilador a tarefa de emitir as instruções vetoriais para a maior parte do código, à exceção da função de compressão. Além do benefício em legibilidade, essa decisão tornou prático emitir versões utilizando diferentes conjuntos de instruções usando o mesmo código, e, conforme a seção 5, não trouxe perda de desempenho significativa.

O código utilizado para a função de compressão é o da implementação de referência da BLAKE2b,⁴ com pequenas adaptações para remover as constantes utilizadas durante aplicações da função G. No entanto, como essa implementação da BLAKE2b não possui versão AVX2, foi necessário adaptá-la conforme descrito na subseção 4.1 para utilizar esse conjunto de instruções.

O algoritmo Lyra2 apresentado na subseção 3.3 não especifica o parâmetro Rt , que determina o tamanho em bits das rotações de blocos; a implementação de referência, em sua versão SSE2, utiliza $Rt = 128$ bits, dado que este é o tamanho de um vetor SSE2. Da mesma forma, nosso trabalho usa $Rt = 128$ quando compilado com SSE2, e $Rt = 256$ em sua versão AVX2.

4.1 A função de compressão em AVX2

Esta seção detalha as mudanças feitas na implementação de referência da função de compressão da Lyra2 para aproveitar as instruções vetoriais do conjunto AVX2.

Conforme exposto na subseção 3.2, a função de compressão da Lyra2 consiste de ρ rodadas da função G sobre uma matriz de estado 4×4 de 64 bits. Assim, para uma matriz da forma:

$$\begin{pmatrix} v_0 & v_1 & v_2 & v_3 \\ v_4 & v_5 & v_6 & v_7 \\ v_8 & v_9 & v_{10} & v_{11} \\ v_{12} & v_{13} & v_{14} & v_{15} \end{pmatrix}$$

uma rodada corresponde a:

$$\begin{matrix} G(v_0, v_4, v_8, v_{12}) & G(v_1, v_5, v_9, v_{13}) & G(v_2, v_6, v_{10}, v_{14}) & G(v_3, v_7, v_{11}, v_{15}) \\ G(v_0, v_5, v_{10}, v_{15}) & G(v_1, v_6, v_{11}, v_{12}) & G(v_2, v_7, v_8, v_{13}) & G(v_3, v_4, v_9, v_{14}) \end{matrix}$$

onde as aplicações em cada linha podem ocorrer em paralelo. No caso da Lyra2, o estado da esponja possui $16 \times 64 = 1024$ bits e é interpretado como uma matriz de estado linearizada em ordem de linhas para os propósitos da função de compressão.

Na implementação vetorizada de referência da BLAKE2b, cada registrador vetorial corresponde a mais de um inteiro de 64 bits da matriz de estado – por exemplo, em uma implementação SSE2, v_0 e v_1 compartilham um registrador de 128 bits. Uma rodada consiste então em executar a função G sobre as linhas da matriz em

⁴Disponível em <https://github.com/BLAKE2/BLAKE2>

paralelo, rotacionar a i -ésima linha da matriz de estado por i posições à esquerda, de forma que as antigas diagonais se tornem as colunas, aplicar a função G sobre as (novas) colunas, e desfazer as rotações (Neves, Aumasson). O procedimento de rotação das linhas é chamado de *diagonalização*.

Dessa forma, uma versão AVX2 da função de compressão pode utilizar as mesmas técnicas que uma versão SSE2, adaptadas para o fato de que os registradores de 256 bits comportam uma linha inteira da matriz de estado por vez. Especificamente, o código AVX2 produzido para este trabalho utiliza as novas instruções `vpaddq` para as adições de 64 bits, `vpxor` para as operações de XOR, `vpshufd` e `vpermq` para as rotações de linhas da matriz.

Em termos de código, nossa implementação AVX2 possui a mesma estrutura geral que a de referência, apesar do uso de *intrinsics* diferentes. A função G é dividida nas partes $G1$ e $G2$: $G1$ contém as instruções até a rotação de 24 bits à direita, e $G2$ contém o restante da função. $G1$ e $G2$ são implementadas conforme a listagem a seguir:

```
#define G1(row1,row2,row3,row4) \
    row1 = _mm256_add_epi64(row1, row2); \
    row4 = _mm256_xor_si256(row4, row1); \
    row4 = _mm256_roti_epi64(row4, -32); \
    row3 = _mm256_add_epi64(row3, row4); \
    row2 = _mm256_xor_si256(row2, row3); \
    row2 = _mm256_roti_epi64(row2, -24);

#define G2(row1,row2,row3,row4) \
    row1 = _mm256_add_epi64(row1, row2); \
    row4 = _mm256_xor_si256(row4, row1); \
    row4 = _mm256_roti_epi64(row4, -16); \
    row3 = _mm256_add_epi64(row3, row4); \
    row2 = _mm256_xor_si256(row2, row3); \
    row2 = _mm256_roti_epi64(row2, -63);
```

Cabe notar que, na listagem acima, `_mm256_roti_epi64`, responsável pelas rotações, é a única primitiva que não é um *intrinsic*. Essa macro é definida como:

```
#define r16_256 _mm256_setr_epi8( \
    2, 3, 4, 5, 6, 7, 0, 1, 10, 11, 12, 13, 14, 15, 8, 9, 18, \
    19, 20, 21, 22, 23, 16, 17, 26, 27, 28, 29, 30, 31, 24, 25)
#define r24_256 _mm256_setr_epi8( \
    3, 4, 5, 6, 7, 0, 1, 2, 11, 12, 13, 14, 15, 8, 9, 10, 19, \
    20, 21, 22, 23, 16, 17, 18, 27, 28, 29, 30, 31, 24, 25, 26)
#define _mm256_roti_epi64(x, c) \
    (-(c) == 32) ? _mm256_shuffle_epi32((x), _MM_SHUFFLE(2,3,0,1)) \
    : (-(c) == 24) ? _mm256_shuffle_epi8((x), r24_256) \
    : (-(c) == 16) ? _mm256_shuffle_epi8((x), r16_256) \
    : (-(c) == 63) ? _mm256_xor_si256(_mm256_srli_epi64((x), -(c)), \
    _mm256_add_epi64((x), (x))) \
    : _mm256_xor_si256(_mm256_srli_epi64((x), -(c)), \
    _mm256_slli_epi64((x), 64-(-(c))))
```

A diagonalização e sua inversa são implementadas com o *intrinsic* `_mm256_permute4x64_epi64`:

```
#define DIAGONALIZE(row1, row2, row3, row4) \
    row2 = _mm256_permute4x64_epi64(row2, _MM_SHUFFLE(0,3,2,1)); \
    row3 = _mm256_permute4x64_epi64(row3, _MM_SHUFFLE(1,0,3,2)); \
```

```

row4 = _mm256_permute4x64_epi64(row4, _MM_SHUFFLE(2,1,0,3));

#define UNDIAGONALIZE(row1, row2, row3, row4) \
row2 = _mm256_permute4x64_epi64(row2, _MM_SHUFFLE(2,1,0,3)); \
row3 = _mm256_permute4x64_epi64(row3, _MM_SHUFFLE(1,0,3,2)); \
row4 = _mm256_permute4x64_epi64(row4, _MM_SHUFFLE(0,3,2,1));

```

E, assim, uma rodada é definida como:

```

#define BLAKE2B_ROUND(v) \
G1(v[0], v[1], v[2], v[3]); \
G2(v[0], v[1], v[2], v[3]); \
UNDIAGONALIZE(v[0], v[1], v[2], v[3]); \
G1(v[0], v[1], v[2], v[3]); \
G2(v[0], v[1], v[2], v[3]); \
UNDIAGONALIZE(v[0], v[1], v[2], v[3]);

```

onde o parâmetro v é uma matriz de estado linearizada, de forma que $v[i]$ é um `__m256i` contendo a i -ésima linha.

Além dessas adaptações na função de compressão, as únicas mudanças não-triviais feitas no restante do código de nossa implementação para habilitar nela o uso de AVX2 dizem respeito aos requerimentos de alinhamento de memória nos operandos das novas instruções.

5 Resultados experimentais

Esta seção apresenta os resultados de uma comparação de desempenho entre diferentes implementações da função Lyra2. Os testes foram conduzidos em dois ambientes diferentes: um Macbook Pro Retina modelo Mid-2012 rodando o sistema operacional OSX na versão 10.10.1, com um processador Intel Core i7-3720QM (família Ivy Bridge) e 16 GiB de memória, e um desktop com processador Intel Core i7-4770 (família Haswell) e 8GiB de memória rodando a distribuição Linux Fedora 18. O recurso TurboBoost, que aumenta dinamicamente a frequência do processador, foi desativado em ambas as máquinas.

A metodologia utilizada consistiu em executar cada implementação 1000 vezes com as mesmas entradas e extraindo-se uma senha de 64 bytes, para três conjuntos de parâmetros R e T do algoritmo Lyra2. O parâmetro C foi deixado fixo em 256, o valor sugerido pela implementação de referência. A mediana dos tempos levados nessas 1000 execuções foi tomada como métrica final de desempenho, para cada conjunto de parâmetros. Os processos foram executados usando *niceness* zero, o padrão em ambos os sistemas.

Para cada um dos códigos, foram testados os binários gerados pelos compiladores clang e gcc, nas respectivas versões 3.4.2 e 4.9.0. No entanto, devido ao limitado suporte ao gcc no sistema OSX, apenas medições com clang foram feitas nessa plataforma. Essas medições visam a avaliar o impacto de delegar a maior parte das escolhas de instruções para o compilador em nosso código, indo de encontro à abordagem da implementação de referência, que se baseia fortemente no uso explícito de *intrinsics*.

A Figura 4 ilustra os resultados experimentais obtidos no ambiente Linux. Nela, os nomes *ref-gcc* e *ref-clang* correspondem à implementação de referência compilada, respectivamente, com gcc e clang. Os nomes *gcc* e *clang* são análogos, mas referem-se à implementação proposta neste trabalho. Finalmente, *avx2-clang* e *avx2-gcc* correspondem à variante que utiliza as instruções AVX2, descrita na subseção 4.1, novamente usando os dois compiladores analisados. Os tempos de execução foram normalizados pelo da medição *ref-gcc* para cada conjunto de parâmetros.

As versões AVX2, conforme esperado, tiveram desempenho melhor do que as demais. Nelas, o compilador utilizado não trouxe diferença significativa (*avx2-clang* foi apenas cerca de 3% mais rápida do que *avx2-gcc* em todos os testes), e ambas foram cerca de 30% mais rápidas do que *ref-gcc* em todos os testes.

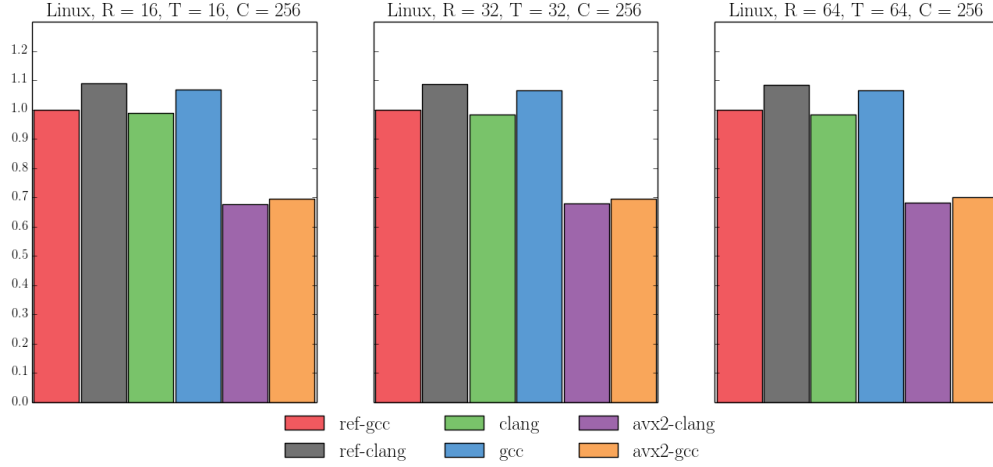


Figura 4: Tempos de execução normalizados para as três configurações de parâmetros em Linux.

Para as demais versões, a escolha do compilador influenciou sensivelmente no desempenho observado, mas não apenas para nossa implementação: considerando os três conjuntos de parâmetros, a maior variação observada foi entre *ref-clang* e *ref-gcc* para $R = T = 16$, em que a primeira foi 8.95% mais lenta ($1947\mu s$ contra $1787\mu s$). De fato, para todos os conjuntos de parâmetros, as versões *ref-clang* foram cerca de 8% mais lentas do que as *ref-gcc*, e as versões *gcc* foram 8% mais lentas do que as *clang*. Entre as versões sem AVX2, *clang* foi a mais rápida para todas as configurações, mas por apenas cerca de 1.5% do tempo de *ref-gcc*, uma diferença pouco expressiva.

A Figura 5 mostra os resultados dos mesmos testes executados na máquina rodando OSX. Como nela não há suporte ao conjunto de instruções AVX2, não foi possível avaliar *avx2-clang* nessa plataforma.

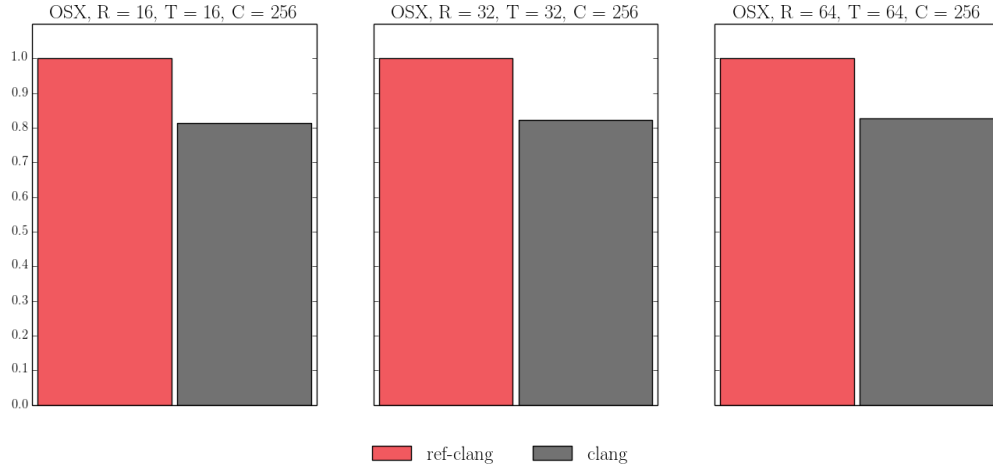


Figura 5: Tempos de execução normalizados para as três configurações de parâmetros em OSX.

Aqui registrou-se significativa diferença de desempenho entre os binários, com *clang* sendo cerca de 18% mais rápida do que a implementação de referência, em todos os testes.

Dado que nossa implementação foi desenvolvida no ambiente com OSX e clang, e a de referência foi produzida em ambiente Linux com gcc, pode-se dizer que as diferenças de desempenho entre compiladores refletem a maior exposição de cada implementação à plataforma em que foi escrita.

6 Conclusão

Este trabalho produziu uma implementação de uma nova função de *hash* de senhas tomando proveito de arquiteturas e compiladores modernos. Nossa implementação é compatível e competitiva com a de referência, tendo-se obtido significativa melhora de desempenho com o uso conjunto de instruções AVX2, e seu desenvolvimento contribuiu com a evolução da especificação.

7 Referências bibliográficas

Bertoni, Guido, Joan Daemen, Michaël Peeters, Gilles Van Assche“Cryptographic Sponge Functions.”

Neves, Samuel, Jean-Philippe Aumasson“Implementing BLAKE with AVX, AVX2, and XOP.”

Neves, Samuel, Jean-Philippe Aumasson, Wilcox-O’HearnZooko, Christian Winnerlein“BLAKE2: simpler, Smaller, Fast as MD5.” <https://blake2.net>.

Simplicio Jr., Marcos A., Leonardo C. Almeida, Ewerton R. Andrade, Paulo C. F. dos Santos, Paulo S. L. M. Barreto“The Lyra2 Reference Guide.” <http://www.lyra-kdf.net/Lyra2ReferenceGuide.pdf>.