Implementação eficiente em *software* da função Lyra2 em arquiteturas modernas

Guilherme P. Gonçalves[1], **Diego F. Aranha**[1]

[1]Laboratório de Segurança e Criptografia (LASCA)
Instituto de Computação (IC) – Universidade Estadual de Campinas (Unicamp)
Av. Albert Einsten, 1251 – Campinas – SP – Brasil

guilherme.p.gonc@gmail.com,dfaranha@ic.unicamp.br

Abstract. *Password authentication has become even more challenging with the significant increase in available computing power by means of dedicated hardware and GPUs. This work presents an efficient software implementation of the password hashing function Lyra2 in modern Intel platforms, according to version 2.5 of its specification. The resulting implementation employs AVX2 vector instructions for a performance improvement of 30% over the reference implementation. In practice, it is important to know the precise performance of such primitives to inform parameter choice and corresponding security guarantees.*

Resumo. *O problema de autenticação por senha tem se tornado cada vez mais desafiador, face ao crescimento significativo de poder computacional na forma de* hardware *dedicado e placas gráficas. Este trabalho apresenta uma implementação eficiente em* software *da função da derivação Lyra2 em arquiteturas Intel modernas, conforme a versão 2.5 de sua especificação. A implementação resultante emprega instruções vetoriais AVX2 para ganhos de desempenho em torno de 30% sobre a implementação de referência. Na prática, é importante determinar precisamente o desempenho de primitivas dessa natureza para informar a escolha de parâmetros e garantias de segurança correspondentes.*

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, o armazenamento da senha de alguma forma no sistema.

Como medida de segurança, recomenda-se não armazenar a senha conforme fornecida pelo usuário, mas sim um *hash* produzido por uma função de *hash* de senhas. O *hash* produz uma sequência pseudoaleatória de *bits* tal que, dado o valor de *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 calcula 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.

Muitas técnicas foram introduzidas para dificultar ataques de busca exaustiva – 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 *hardware* dedicado ou GPUs, caracterizados pelo paralelismo e escassez de memória.

Neste trabalho, foi produzida uma implementação da função de *hash* de senhas Lyra2, proposta por pesquisadores brasileiros e finalista do *Password Hashing Competition*[1], aproveitando conjuntos de instruções presentes em arquiteturas modernas. Acredita-se que a Lyra2 possua as propriedades de segurança exigidas de uma função de *hash* de senhas, e sua especificação inclui parâmetros de espaço e tempo independentes a serem respeitados por uma implementação correta. A implementação resultante[2] é compatível com a de referência e competitiva em termos de desempenho. Houve colaboração ainda com o desenvolvimento da especificação da Lyra2 [Simplício et al. 2015], na medida em que este trabalho trouxe à tona diversas inconsistências no documento de especificação e na implementação de referência.

2. A função Lyra2

Para o restante deste documento, os símbolos da primeira coluna da Tabela 1 a seguir serão usados com o significado dado pela segunda coluna.

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
$\|\|$	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

Tabela 1: Notação utilizada neste trabalho.

A função 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.

2.1. A construção de esponja

A definição canônica de uma esponja [Bertoni et al. 2011] descreve sua operação em termos de duas etapas – *absorbing* e *squeezing* –, sendo que na primeira a esponja

[1] `https://password-hashing.net/`

[2] Disponível em `https://github.com/guilherme-pg/lyra2`

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 (taxa) e c (capacidade) *bits*.

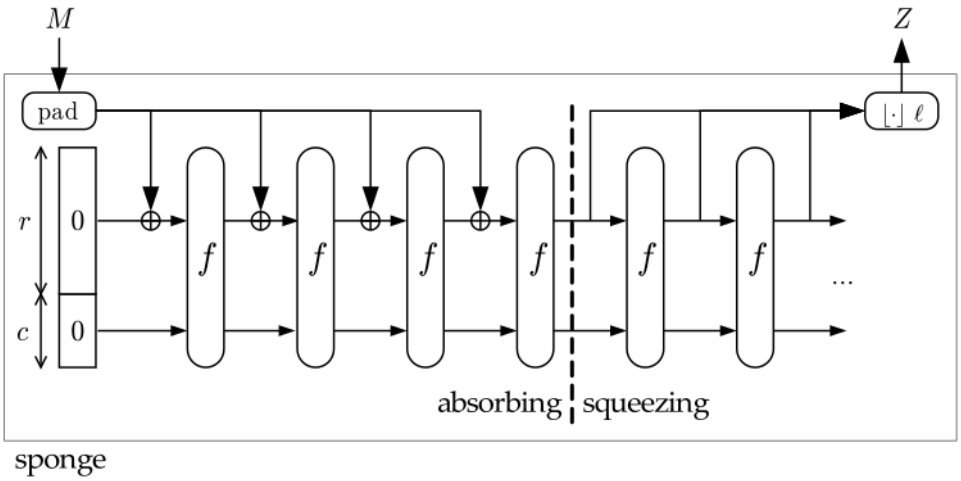

Figura 1. A construção de esponja, adaptado de [Bertoni et al. 2011].

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*.

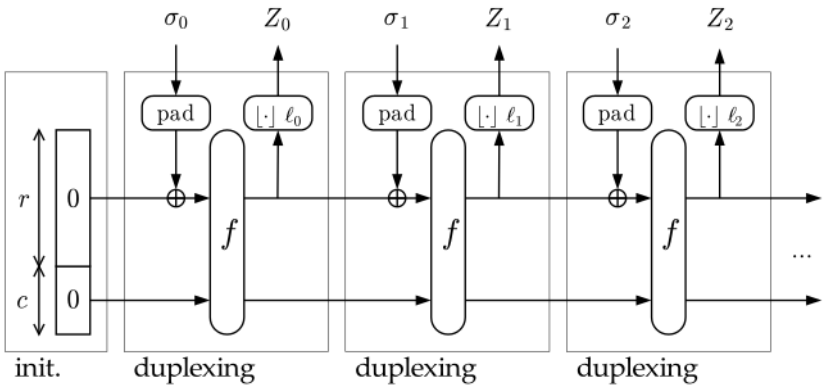

Figura 2. A construção de duplex, adaptado de [Bertoni et al. 2011].

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.

2.2. 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 [Aumasson et al. 2013] . 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.

2.3. 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 no código desenvolvido quanto no 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.

Algorithm 2 The Lyra2 Algorithm.

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

▷ BOOTSTRAPPING PHASE: Initializes the sponge's state and local variables
$params \leftarrow len(k) \,\|\, len(pwd) \,\|\, len(salt) \,\|\, T \,\|\, R \,\|\, C$ ▷ Byte representation of input parameters (others can be added)
$H.absorb(\texttt{pad}(pwd \,\|\, salt \,\|\, params))$ ▷ Padding rule: 10^*1. Password can be overwritten after this point
$gap \leftarrow 1$; $stp \leftarrow 1$; $wnd \leftarrow 2$ ▷ Initializes visitation step and window
$prev^0 \leftarrow 2$; $row^1 \leftarrow 1$; $prev^1 \leftarrow 0$

▷ SETUP PHASE: Initializes a $(R \times C)$ memory matrix, it's cells having b bits each
for $(col \leftarrow 0$ **to** $C-1)$ **do** $\{M[0][C-1-col] \leftarrow H_\rho.squeeze(b)\}$ **end for** ▷ Initializes $M[0]$
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** ▷ Initializes $M[1]$
for $(col \leftarrow 0$ **to** $C-1)$ **do** ▷ Initializes $M[2]$ and updates $M[0]$
  $rand \leftarrow H_\rho.duplex(M[0][col] \boxplus M[1][col])$
  $M[2][C-1-col] \leftarrow M[1][col] \oplus rand$
  $M[0][col] \leftarrow M[0][col] \oplus rotRt(rand)$ ▷ $rotRt()$: right rotation by L bits (e.g., 1 or more words)
end for
for $(row^0 \leftarrow 3$ **to** $R-1)$ **do** ▷ **Filling Loop**: initializes remainder rows
  for $(col \leftarrow 0$ **to** $C-1)$ **do** ▷ **Columns Loop**: $M[row^0]$ is initialized, while $M[row^1]$ is updated
    $rand \leftarrow H_\rho.duplex(M[row^1][col] \boxplus M[prev^0][col] \boxplus M[prev^1][col], b)$
    $M[row^0][C-1-col] \leftarrow M[prev^0][col] \oplus rand$
    $M[row^1][col] \leftarrow M[row^1][col] \oplus rotRt(rand)$
  end for
  $prev^0 \leftarrow row^0$; $prev^1 \leftarrow row^1$; $row^1 \leftarrow (row^1 + stp) \bmod wnd$ ▷ Picks rows to be revisited in next loop
  if $(row^1 = 0)$ **then** ▷ Window fully revisited
    $stp \leftarrow wnd + gap$; $wnd \leftarrow 2 \cdot wnd$; $gap \leftarrow -gap$ ▷ Doubles window size and roughly doubles step
  end if
end for

▷ WANDERING PHASE: Iteratively overwrites pseudorandom cells of the memory matrix
for $(\tau \leftarrow 1$ **to** $T)$ **do** ▷ **Time Loop**
  for $(i \leftarrow 0$ **to** $R-1)$ **do** ▷ **Visitation Loop**: $2R$ rows revisited in pseudorandom fashion
    for $(d \leftarrow 0$ **to** $1)$ **do** $\{row^d \leftarrow (\text{LSW}(rotRt^d(rand))) \bmod R\}$ **end for** ▷ Picks pseudorandom rows
    for $(col \leftarrow 0$ **to** $C-1)$ **do** ▷ **Columns Loop**: updates each $M[row^d]$
      for $(d \leftarrow 2$ **to** $3)$ **do** $\{col^d \leftarrow (\text{LSW}(rotRt^d(rand))) \bmod C\}$ **end for** ▷ Picks pseudorandom columns
      $rand \leftarrow H_\rho.duplex(M[row^0][col] \boxplus M[row^1][col] \boxplus M[prev^0][col^0] \boxplus M[prev^1][col^1], b)$
      for $(d \leftarrow 0$ **to** $1)$ **do**
        $M[row^d][col] \leftarrow M[row^d][col] \oplus rotRt^d(rand)$ ▷ Updates the d pseudorandom rows
      end for
    end for ▷ End of Columns Loop
    for $(d \leftarrow 0$ **to** $1)$ **do** $\{prev^d \leftarrow row^d\}$ **end for** ▷ Next iteration revisits most recently updated rows
  end for ▷ End of Visitation Loop
end for ▷ End of the Time Loop

▷ WRAP-UP PHASE: key computation
$H.absorb(M[row^0][col^0])$ ▷ Absorbs a final column with the full-round sponge
$K \leftarrow H.squeeze(k)$ ▷ Squeezes k bits with the full-round sponge
return K ▷ Provides k-long bitstring as output

Figura 3. O algoritmo Lyra2 implementado neste trabalho, conforme versão 2.5 da especificação [Simplício et al. 2015].

3. Descrição da implementação

A implementação proposta 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 (sem instruções vetoriais) e SSE2, cabe clarificar que, para o restante deste documento, e em particular na Seção 4, quaisquer menções à implementação proposta ou à de referência se referem às respectivas versões SSE2 (a não ser, é claro, que a versão AVX2 deste trabalho esteja sendo discutida).

A versão SSE2 da implementação, embora funcionalmente equivalente à de referência, é baseada em decisões de projeto levemente diferentes. Além de prezar pelo desempenho, o 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, a 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 4, não trouxe perda de desempenho significativa.

O código utilizado para a função de compressão é o da implementação de referência da BLAKE2b [3], 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 [Intel 2014] para utilizadar esse conjunto de instruções.

O algoritmo Lyra2 apresentado na Subseção 2.3 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, o trabalho usa $Rt = 128$ quando compilado com SSE2, e $Rt = 256$ em sua versão AVX2.

3.1. 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 2.2, 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:

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

[3] Disponível em `https://github.com/BLAKE2/BLAKE2`

uma rodada corresponde a:

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

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 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 [Aumasson et al. 2013]. 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 `vpshufd` para as rotações, e `vpermq` para as rotações de linhas da matriz.

Em termos de código, a 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 a partir do *intrinsic* de permutação de palavras `_mm256_permute4x64_epi64`, definido como abaixo:

```
#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]);          \
  DIAGONALIZE(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 a implementação para habilitar nela o uso de AVX2 dizem respeito aos requerimentos de alinhamento de memória nos operandos das novas instruções.

4. 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 uma máquina 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 prioridade normal, o padrão em ambos os sistemas.

Para cada um dos códigos, foram testados os binários gerados pelos compiladores LLVM 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 LLVM foram feitas nessa plataforma. Essas medições visam a avaliar o impacto de delegar a maior parte das escolhas de instruções para o compilador no 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 LLVM. 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 [Intel 2014], novamente usando os dois compiladores analisados. Os tempos de execução foram normalizados pelo da medição *ref-gcc*, para cada conjunto de parâmetros.

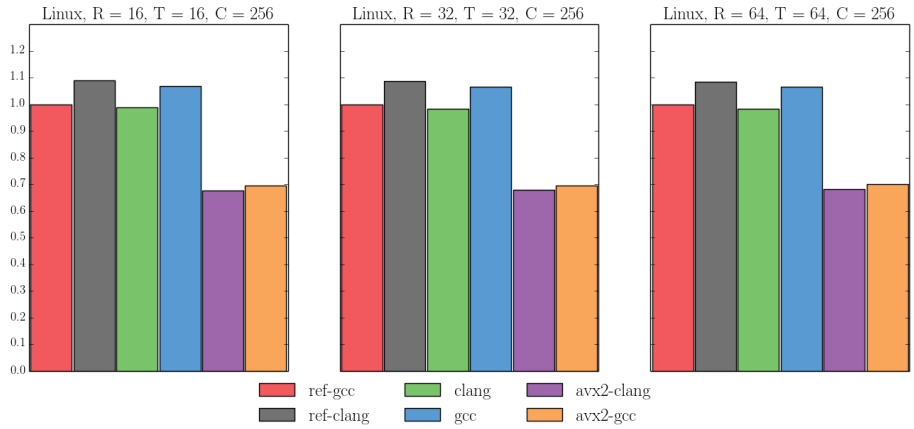

Figura 4. Tempos de execução normalizados para as três configurações de parâmetros em Linux.

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.

Para as demais versões, a escolha do compilador influenciou sensivelmente o desempenho observado, mas não apenas para esta 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. 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.

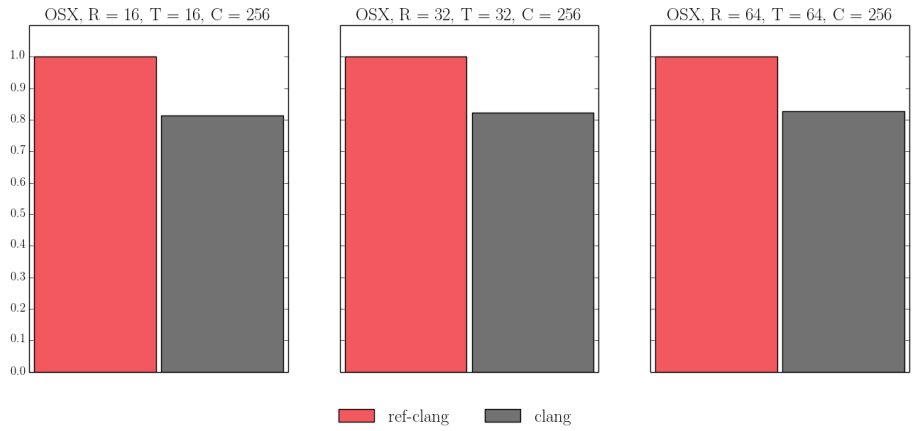

Figura 5. Tempos de execução normalizados para as três configurações de parâmetros em OSX.

Dado que a implementação proposta foi desenvolvida no ambiente com OSX e LLVM, 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.

5. Conclusão

Este trabalho produziu uma implementação de uma nova função de *hash* de senhas tomando proveito de arquiteturas e compiladores modernos. A 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. É importante determinar precisamente o desempenho de primitivas dessa natureza para tanto informar a escolha de parâmetros quanto estimar as garantias de segurança correspondentes.

Referências

Aumasson, J., Neves, S., Wilcox-O'Hearn, Z., and Winnerlein, C. (2013). BLAKE2: simpler, smaller, fast as MD5. In Jr., M. J. J., Locasto, M. E., Mohassel, P., and Safavi-Naini, R., editors, *Proceedings of the 11th International Conference on Applied Cryptography and Network Security (ACNS)*, volume 7954 of *Lecture Notes in Computer Science*, pages 119–135. Springer.

Bertoni, G., Daemen, J., Peeters, M., and Assche, G. (2011). Cryptographic sponge functions. `http://sponge.noekeon.org/`.

Intel (2014). Intel Architecture Instruction Set Extensions Programming Reference. Technical Report. `https://software.intel.com/en-us/isa-extensions/`.

Simplício, M. A., Almeida, L. C., Andrade, E. R., dos Santos, P. C. F., and Barreto, P. S. L. M. (2015). Lyra2: Password hashing scheme with improved security against time-memory trade-offs. `http://www.lyra-kdf.net`.