Implementação compacta em software do algoritmo Ketje

Carlos Gregoreki[1], **Diego F. Aranha**[1]

[1]Instituto de Computação (IC) - Universidade Estadual de Campinas (Unicamp)

cgregoreki@gmail.com, dfaranha@ic.unicamp.br

Abstract. *The advances in mobile computing brought to light the information security problem, imposing new requirements for cryptosystems for increasingly smaller hardware, with reduced computation and storage capacity. This work presents a compact implementation of the authenticated encryption scheme Ketje performed on the programming language C, according to version 1.1 of its specification, for keys and nonces with length multiple of 8 bits. The present work aimed at improvements in code readability and resulted in a 69% smaller compiled code, while behaving similarly from the point of view of performance.*

Resumo. *O avanço do desenvolvimento de dispositivos móveis trouxe à tona o problema de segurança da informação, impondo novos requisitos de sistemas criptográficos para dispositivos cada vez menores, com capacidades de computação e armazenamento reduzidas. Este trabalho apresenta uma implementação compacta do esquema de cifração autenticada Ketje realizada em linguagem C, conforme a versão 1.1 de sua especificação, para chaves e nonces de tamanhos múltiplos de 8 bits. Visando uma melhoria na legibilidade do código, o presente trabalho teve como resultado uma redução no tamanho final do código compilado de até 69% comportando-se de maneira semelhante à referência do ponto de vista de desempenho.*

1. Introdução

Uma das formas mais utilizadas para armazenar informações confidenciais ou trocar mensagens de maneira segura é a criptografia simétrica. Essa técnica exige que exista uma chave secreta que deve ser compartilhada entre o cifrador da mensagem e o decifrador, e que o compartilhamento dessa chave seja dada por meios seguros para garantir sua confidencialidade. Desta forma, uma mensagem pode ser decifrada por uma chave em conjunto com um algoritmo se eles forem os mesmos que foram utilizados na cifração.

Ao longo do desenvolvimento de sistemas criptográficos, muitas formas de ataques por canais laterais – que objetivam identificar características de um sistema criptográfico através de, por exemplo, padrões de consumo de energia e ondas eletromagnéticas – apareceram e se tornaram técnicas cada vez mais frequentes de ataque. Além disso, com a portabilidade sendo cada vez mais desenvolvida, segurança em dispositivos pequenos e móveis tem sido cada vez mais exigida.

Neste contexto, implementações eficientes de novas primitivas criptográficas voltadas à dispositivos com restrição de memória e espaço e com resistência a esses ataques são um tema presente em competições de segurança como a *CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness*[1].

[1]https://competitions.cr.yp.to/caesar.html

No presente trabalho, foi produzida uma implementação[2] alternativa e mais compacta do algoritmo *Ketje* em suas versões recomendadas pela referência, *KetjeJr* e *KetjeSr*, para funcionamento com chaves e *nonces* de tamanho de bits múltiplos de 8 – casos comuns – visando a legibilidade do código, o espelhamento entre a implementação produzida deste trabalho e sua especificação, além da redução do tamanho do código compilado.

É importante citar que este algoritmo foi submetido à CAESAR por tratar dos requisitos da competição, a citar: é um cifrador autenticado com suporte à *associated data* (AEAD), é possível recuperar os dados originais com o criptograma e o *associated data*, possui flexibilidade para aceitar chaves de tamanho não comuns, isto é, 80, 128 ou 256 bits, e especifica dois casos particulares formando assim uma "família" de cifradores autenticados. Também propõe vantagens em relação ao AES-GCM. De acordo com a especificação, *Ketje* possui melhor desempenho e menor código e se comporta de maneira segura perante a ataques de canais laterais, além de apresentar suporte à sessões, isto é, autenticação de sequência de mensagens em série ao invés de apenas uma mensagem.

Para observar a execução da implementação proposta, escolhemos como plataformas alvo processadores que integrassem dispositivos móveis e que fossem conhecidos pela comunidade em geral, que nesses processadores estivesse sendo executado um sistema operacional de código aberto, e que fosse possível coletar dados de desempenho de maneira simples e confiável. Portanto, escolhemos o ArchLinux como sistema operacional e processadores ARM Cortex-A.

2. Preliminares

2.1. Cifração Autenticada

De acordo com [Bellare, Rogaway and Wagner 2003], um esquema de cifração autenticada é um mecanismo de cifração simétrica, no qual uma mensagem M é transformada em um criptograma CT com o objetivo que CT proteja a autenticidade e a privacidade de M. Segundo [Bertoni et al 2014], podemos entender cifração autenticada como um conjunto de operações, constituída de uma função que recebe um cabeçalho A e uma cadeia de caracteres B como entradas, e produz um criptograma C e uma *tag* T como saída – processo chamado de *cifração* –, e do processo inverso, que recebe um cabeçalho A e um criptograma C, produzindo uma cadeia de caracteres B e uma *tag* T, chamado de *decifração*.

2.2. Notação

Neste trabalho, utilizaremos a notação exposta na Tabela 1.

3. Ketje

O esquema de cifração autenticada *Ketje* se baseia na construção *duplex* – variante da construção esponja – na sua versão *MonkeyDuplex*. De acordo com sua definição [Bertoni et al 2011b], esponja é uma forma geral para a geração de funções de *hash* com entradas e saídas de tamanho arbitrário, baseada numa transformação de tamanho fixo que opera em cima de um número fixo de bits. *Duplex*, cuja segurança é semelhante, permite que blocos únicos de saída sejam gerados a medida que blocos de entrada são incorporados ao estado da construção. Assim, permite-se construir esquemas criptográficos requisitando apenas uma chamada dessa transformação por bloco.

[2]Disponível em http://github.com/cgregoreki/ketje-impl-v2

Tabela 1. Símbolos e seus significados

Símbolo	Significado
$\oplus$	Xor bit a bit
$\lfloor . \rfloor l$	Truncagem para l bits
$\lvert K \rvert$	Tamanho de K
$a \lvert\rvert b$	Concatenação de a e b
$f \circ g$	Composição de funções f e g: $f(g)$
$GF(p^n)$	Bits. $GF(2) = \{0, 1\}$

3.1. As construções esponja e duplex

Citando [Bertoni et al 2007], a operação da esponja é descrita em duas partes: *absorbing* e *squeezing*. A primeira é a etapa em que a esponja absorve e incorpora a entrada ao seu estado interno, e a segunda é quando uma saída de tamanho desejado é produzida baseada em tal estado. Desta forma, ao final de sua operação, a esponja terá produzido um *hash* pseudoaleatório de sua entrada.

A construção esponja está detalhada na Figura 1. A entrada M é divida em blocos após um *padding* e é absorvida para depois, então, gerar a saída Z. A função f é uma função de permutação.

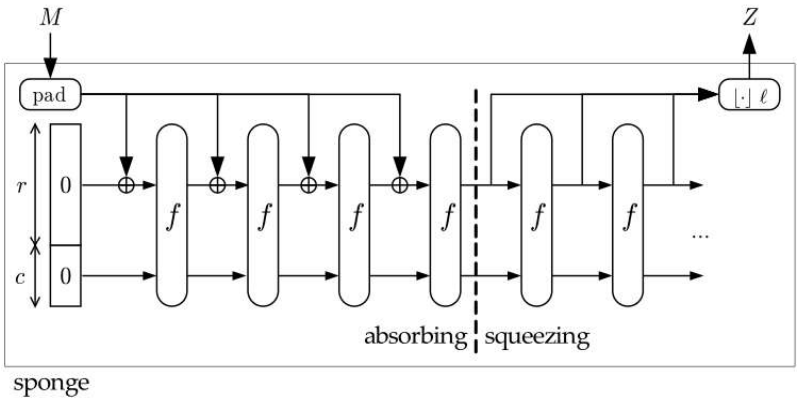

Figura 1. A construção de esponja, adaptado de [Bertoni et al 2011a]

A Figura 2 mostra a construção *duplex*. Aqui, ao invés de gerar uma saída integralmente, a saída é gerada bloco a bloco à medida que os blocos de entrada são incorporados.

3.2. MonkeyDuplex e MonkeyWrap

MonkeyDuplex usa uma função de permutação f com uma quantidade regulável de rodadas, aceita uma *cadeia de bits* como entrada e produz uma *cadeia de bits* como saída. Esta saída depende de todas as entradas recebidas até então. Diferentemente de *duplex*, na qual foi baseada, essa construção aceita dois tipos de chamadas que são diferentes em número de rodadas que são executadas pela função de permutação entre a entrada e a saída. A construção $MonkeyDuplex[f, r, n_{start}, n_{step}, n_{stride}]$ funciona de acordo com o seguinte:

- Uma instância de *MonkeyDuplex*, denotada por *D*, possui um estado de *b* bits igual ao tamanho da largura da função de permutação. Essa instância é inicializada com

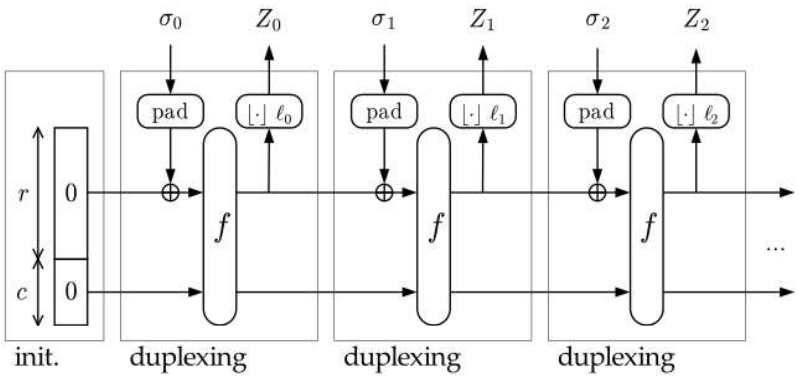

Figura 2. A construção duplex, adaptado de [Bertoni et al 2011a]

uma *string I*, que pode ser de tamanho quase igual a b, sendo que I é estendida até b bits por uma função *multi-rate padding*. Posteriormente, a função de permutação é aplicada n_{start} vezes.

- Em seguida, com chamadas para $D.step(\sigma, l)$ e $D.stride(\sigma, l)$, pode-se injetar no estado uma palavra σ de até $r - 2$ bits. Depois que os bits forem injetados, tanto $f[n_{step}]$ quanto $f[n_{stride}]$ são aplicadas ao estado e os primeiros l bits são extraídos.

Para melhor ilustrar, a construção *MonkeyDuplex* está exposta na Figura 3. O algoritmo é descrito na Figura 4.

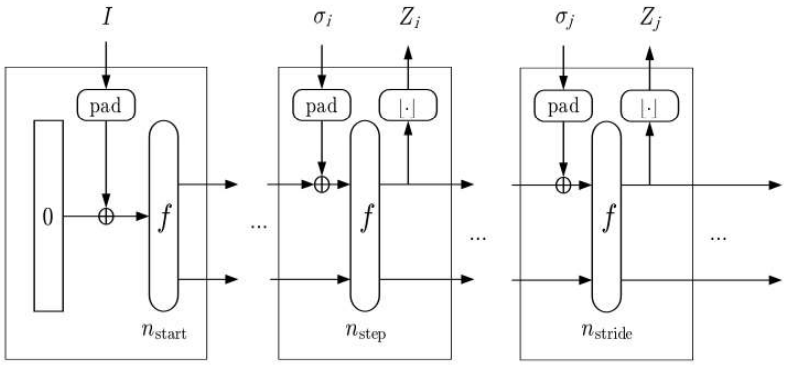

Figura 3. A construção MonkeyDuplex, adaptado de [Bertoni et al 2014]

Por sua vez, *MonkeyWrap* é um modo de cifração autenticada que é construído com a construção *MonkeyDuplex*. Assim, *MonkeyWrap* se vale das operações de *MonkeyDuplex* – *step* e *stride* – para manipular seu estado, utilizando-as nas suas funções de cifração e decifração. Portanto, depois de inicializar seu estado com a chave secreta e com um *nonce*, e adicionar os dados de cabeçalho ao estado fazendo chamadas para a função *step* de *MonkeyDuplex*:

- Para cifrar uma mensagem, *MonkeyWrap* extrai o texto cifrado do estado à medida que vai adicionando blocos de bits da mensagem original e fazendo novas chamadas a *step*. A cada bloco da mensagem original inserida, um bloco da mensagem cifrada é gerada.

Algorithm 1: The $MonkeyDuplex[f, r, n_{start}, n_{step}, n_{stride}]$ construction:

Require: $2 < r < b, n_{step} < n_{stride}$
Require: $s \in \mathbb{Z}_2^b$

Interface: $D.start(I)$ with $I \in \mathbb{Z}_2^{\leq b-2}$
$s = I||pad10 * 1[b](|I|)$
$s = f[n_{start}](s)$

Interface: $Z = D.step(\sigma, l)$ with $l \leq r, \sigma \in \mathbb{Z}_2^{\leq r-2}$ and $Z \in \mathbb{Z}_2^l$
$P = \sigma||pad10 * 1[r](|\sigma|)$
$s = s \oplus (P||0^{b-r})$
$s = f[n_{step}](s)$
return $\lfloor s \rfloor_l$

Interface: $Z = D.stride(\sigma, l)$ with $l \leq r, \sigma \in \mathbb{Z}_2^{\leq r-2}$ and $Z \in \mathbb{Z}_2^l$
$P = \sigma||pad10 * 1[r](|\sigma|)$
$s \oplus (P||0^{b-r})$
$f[n_{stride}](s)$
return $\lfloor s \rfloor_l$

Figura 4. Algoritmo para MonkeyDuplex, adaptado de [Bertoni et al 2014]

- Para decifrar uma mensagem, *MonkeyWrap* recupera o texto original à medida que insere blocos de bits do texto cifrado ao estado (com chamadas a *step*).
- Para cada uma das funções de cifração e decifração, uma *tag* é gerada ao final fazendo chamadas às funções *stride* e *step*.

O algoritmo de *MonkeyWrap* está representado na Figura 5.

3.3. As permutações *Keccak-p*

Como vimos, na construção *MonkeyDuplex*, é necessária a presença de uma função de permutação. Para o *Ketje*, a função de permutação escolhida é a função $Keccak$-p que, de acordo com [Bertoni et al 2011a], é uma derivação da permutação $Keccak$-f com um número regulável de rodadas. Uma permutação $Keccak$-p é definida pela sua largura b em bits e seu número n_r de rodadas. $Keccak$-$p[b, n_r]$ consiste da aplicação das últimas n_r rodadas de *Keccak-f[b]*, onde $b = 25$ x 2^l e $b \in \{25, 50, 100, 200, 400, 800, 1600\}$.

A permutação $Keccak$-$p[b, n_r]$ é descrita como uma sequência de operações realizadas num estado *a* que é uma matriz tridimensional de elementos GF(2), da forma $a[5, 5, w]$, com $w = 2^l$, formada a partir de 5 passos. Isto é, podemos admitir que o estado a é uma matriz tridimensional com 5 linhas, 5 colunas e w células de profundidade – células estas que podem ter valores 0 ou 1 – sendo que w depende diretamente do parâmetro b, sendo mais especificamente $w = (b/25)$. Por exemplo, esse estado, para $Keccak$-$p[400, n_r]$, seria uma matriz com elementos $\in \{0, 1\}$ de dimensões $[5, 5, 16]$.

Uma rodada *R* sobre essa matriz pode ser definida como:

$$R = \iota \circ \chi \circ \pi \circ \rho \circ \theta$$

onde ι, χ, π, ρ e θ são funções definidas na Figura 6, tal que *RC* – conjunto de constantes de rodada – é dado por:

$$RC[i_r][0, 0, 2^j - 1] = rc[j + 7i_r], 0 \leq j \leq l$$

Algorithm 2: The $MonkeyWrap[f, \rho, n_{start}, n_{step}, n_{stride}]$ construction:

Require: $0 < \rho \leq b - 4$
Require: $D = MonkeyDuplex[f, \rho + 4, n_{start}, n_{step}, n_{stride}]$

Interface: W.initialize(K, N) with $K \in \mathbb{Z}_2^{\leq b-18}$, $|K|$ mod $8 = 0$ and $N \in \mathbb{Z}_2^{\leq b-|K|-1}$, D.start(keypack$(K, |K| + 16)||N)$

Interface: $(C, T) = W$.wrap(A, B, l) with $A, B, C \in \mathbb{Z}_2^*$, $l \geq 0$, and $T \in \mathbb{Z}_2^l$
for $i = 0$ to $||A|| - 2$ **do**
    D.step$(A_i||00, 0)$
$Z = D$.step$(A_{||A||-1}||01, |B_0|)$
$C_0 = B_0 \oplus Z$
for $i = 0$ to $||B|| - 2$ **do**
    $Z = D$.step$(B_i||11, |B_{i+1}|)$
    $C_{i+1} = B_{i+1} \oplus Z$
$T = D$.stride$(B|||B|| - 1||10, \rho)$
while $|T| < l$ **do**
    $T = T||D$.step$(0, \rho)$
$T = \lfloor T \rfloor_l$
return (C, T)

Interface: $B = W$.unwrap(A, C, T) with $A, B, C, T \in \mathbb{Z}_2^*$
for $i = 0$ to $||A|| - 2$ **do**
    D.step$(A_i||00, 0)$
$Z = D$.step$(A_{||A||-1}||01, |B_0|)$
$B_0 = C_0 \oplus Z$
for $i = 0$ to $||C|| - 2$ **do**
    $Z = D$.step$(B_i||11, |C_{i+1}|)$
    $B_{i+1} = C_{i+1} \oplus Z$
$T' = D$.stride$(B_{||C||-1}||10, \rho)$
while $|T'| < |T|$ **do**
    $T' = T'||D$.step$(0, \rho)$
$T' = \lfloor T \rfloor_{|T|}$
if $T = T'$ **then**
    return B
else
    return error

Figura 5. Algoritmo para MonkeyWrap, adaptado de [Bertoni et al 2014]

onde i_r é o índice da rodada. Todos os outros valores de $RC[i_r][x, y, z]$ são iguais a zero. Os valores de $rc[t]$ são definidos como o resultado de um *binary linear feedback shift register* (LFSR):

$$rc[t] = (x^t \pmod{x^8 + x^6 + x^5 + x^4 + 1}) \pmod{x}$$

$\theta : a[x, y, z] \leftarrow a[x, y, z] + \sum_{y'=0}^{4} a[x-1, y', z] + \sum_{y'=0}^{4} a[x+1, y', z-1]$,
$\rho : a[x, y, z] \leftarrow a[x, y, z-(t+1)(t+2)/2]$,
with satisfying $0 \leq t < 24$ and $\begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}^t \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix}$ in GF$(5)^{2x2}$,
or $t = -1$ if $x = y = 0$,
$\pi : a[x, y] \leftarrow a[x', y']$, with $\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} x' \\ y' \end{pmatrix}$,
$\chi : a[x] \leftarrow a[x] + (a[x+1] + 1)a[x+2]$,
$\iota : a \leftarrow a + RC[i_r]$.

Figura 6. Funções de Keccak-f, adaptado de [Bertoni et al. 2011].

3.4. *KetjeJr* e *KetjeSr*

Como visto na sua especificação, são recomendadas duas instâncias concretas de *Monkey-Wrap* com função de permutação $Keccak\text{-}p$. A principal proposta é chamada de *KetjeSr*, sendo uma segunda chamada de *KetjeJr*:

$$KetjeSr = MonkeyWrap(Keccak{-}p[400], \rho = 32, n_{start} = 12, n_{step} = 1, n_{stride} = 6)$$

$$KetjeJr = MonkeyWrap(Keccak{-}p[200], \rho = 16, n_{start} = 12, n_{step} = 1, n_{stride} = 6)$$

KetjeSr suporta chaves de até 382 bits, e nonces de até $382 - |K|$. A adoção de chaves de tamanhos maiores que 128 bits (tamanho recomendado) pode ser feita como uma possível contra-medida contra ataques do tipo *multi-target*, onde um agente procura atacar um conjunto de chaves ao invés de uma chave apenas. *KetjeJr* suporta chaves de até 182 bits e nonces de até $182 - |K|$ bits. É recomendada a adoção de chaves de 96 bits.

4. Implementação

A implementação foi feita em C, assim como na referência. Além de manter a fidelidade com a especificação, o presente trabalho se preocupou com a legibilidade do código e seu resultado em tamanho final, a ponto de que ficasse menor que o tamanho do executável gerado pela referência sem, é claro, ter muito a perder com relação à eficiência.

Em termos de legibilidade, é notável que o código de referência não acompanha de forma trivial a especificação, podendo-se notar várias operações condicionais que tornam a leitura onerosa; operações estas que, por decisões de projeto, evidentemente estão presentes em função dos testes que foram criados e do propósito de evitar a chamada de operações em ordem incorreta ou inválida. Por exemplo, executar a cifração antes de adicionar os dados de cabeçalho ao estado. Dessa forma, depois da análise da leitura da referência, foi decidido que as operações condicionais citadas deveriam ser refeitas e até mesmo excluídas para evitar prolixidade na leitura, mantendo a correção do código.

A fim de detalhar o estilo de código adotado que impacta na leitura, um trecho de código da presente implementação está disposto na Listagem 2, enquanto o trecho de

código que realiza as mesmas ações na referência está colocado na Listagem 1. Nestas listagens, fica claro a não linearidade de leitura da referência devido a presença de operações condicionais e de retornos em caso de erro, enquanto a implementação compacta segue suas operações sem tais oscilações.

```
1
2  ...
3    if ( (instance->phase & Ketje_Phase_FeedingAssociatedData) != 0){
4          Ket_Step( instance->state, instance->dataRemainderSize, ↩
               FRAMEBITS01 );
5          instance->dataRemainderSize = 0;
6          instance->phase = Ketje_Phase_Wrapping;
7      }
8      if ( (instance->phase & Ketje_Phase_Wrapping) == 0){
9          return 1;
10     }
11
12     if ( (instance->dataRemainderSize + dataSizeInBytes) > ↩
           Ketje_BlockSize ) {
13   ...
14     if ( dataSizeInBytes > Ketje_BlockSize )
15         {
16           for ...
17         }
18 ...
```

Listagem 1. Trecho de código da referência

```
1  ...
2     Ketje_step(instance->state, instance->dataRemainderSize, ↩
          FRAMEBITS01);
3     instance->dataRemainderSize = 0;
4
5     int nblocks_B = return_ketjeJrSize(dataSizeInBytes_B)/↩
          Ketje_BlockSize;
6     if (nblocks_B > 0){
7           for (i = 1; i <= nblocks_B;i++){
8               ...
9  ...
```

Listagem 2. Trecho de código da implementação compacta

Ademais, a especificação descreve a inserção dos dados de cabeçalho no estado como parte interna dos algoritmos de cifração e decifração. Todavia, a implementação de referência externaliza esse trecho numa função separada, chamando-a antes da execução da cifração e decifração. Assim, ainda prezando pela legibilidade e espelhamento do código com a especificação, decidimos seguir a especificação colocando tal inserção dentro dessas funções.

O código da função de permutação $Keccak$-p utilizado foi o mesmo código de referência com modificações para que o conjunto de constantes de rodada fossem pré carregadas na memória ao invés de gerá-las dinamicamente, visto que essas constantes

são fixas para cada tamanho do estado da função. Dessa forma, é possível prever que o processador execute uma quantidade de ciclos menor.

Quanto aos testes, é evidente que um esquema diferente da referência fosse elaborado pois nela há um gerador de conteúdo para as chaves e *nonces* com aspectos diferentes dos que foram adotados aqui. Isto posto, os testes utilizados neste trabalho, para garantir a correção do objeto final, seguiram a mesma estrutura dos que estão presentes na referência, tendo a parte dos parâmetros do gerador modificada para que funcionasse apenas para tamanhos múltiplos de 8 bits. Para garantir a integridade e confiabilidade dos dados coletados ao executar os testes, o mesmo esquema foi reproduzido no arquivo de referência, substituindo os testes originais, para que executassem os mesmos que a implementação aqui proposta. Foram sobre estes testes que os dados foram coletados.

Assim, os testes para correção e benchmarking são constituídos de chaves e *nonces* dos seguintes tamanhos, respectivamente, para *KetjeJr*: 176 bits e 0 bits, 152 bits e 24 bits, 128 bits e 48 bits, 112 bits e 64 bits, 96 bits e 80 bits, e 96 bits e 16 bits. Para *KetjeSr*, respectivamente: 376 bits e 0 bits, 280 bits e 96 bits, 184 bits e 192 bits, 160 bits e 216 bits, 136 bits e 240 bits, 112 bits e 264 bits, 96 bits e 280 bits, 96 bits e 120 bits, 96 bits e 56 bits. Para cada um desses pares, dados de cabeçalho e palavras a serem cifradas foram geradas por uma função semelhante a que a referência utiliza para o mesmo propósito, sendo que seus comprimentos são variáveis, sempre respeitando o limite do estado. Portanto, um total de 408 testes são executados para *KetjeJr* e 5922 para *KetjeSr*.

5. Resultados Experimentais

Os códigos, tanto de refêrencia quanto da implementação proposta, foram compilados de duas formas: com a flag *-Os*, que é para otimizar tamanho e, separadamente, com a flag *-O3* que serve para otimizar desempenho. O compilador utilizado foi o GCC 6.1.1.

Para observar o desempenho dos arquivos gerados pelo compilador (GCC), utilizamos a *Performance Monitoring Unit* (PMU)[3] disponível nos processadores ARM. Assim, pudemos acessar um registrador de contagem de ciclos por meio de um módulo carregado diretamente no *kernel* do sistema operacional utilizado (ArchLinux). Para o tamanho dos arquivos finais, o comando *size -B <nome_do_arquivo>* acessível nas plataformas Linux foi utilizado, observando sempre a sessão *text* sua saída, a qual traduz o tamanho do código em instruções que o processador irá executar. Tais medições foram realizadas em quatro processadores: Cortex-A8, Cortex-A15, Cortex-A53 com plataforma 32-bit e Cortex-A53 com plataforma 64-bit.

Expomos os dados coletados de tamanho do código compilado na Tabela 2 e os dados de desempenho nas Tabelas 3 e 4. Tanto nessas tabelas quanto nos gráficos desse trabalho, os dados para a implementação de referência possuem o prefixo *Ref*.

Observando a Tabela 2, podemos notar uma diferença significativa entre as reduções de espaço para os dois procedimentos. Isto é, enquanto tivemos uma redução de tamanho de apenas 21% utilizando a flag *-Os*, obtivemos uma redução de até 69% para *KetjeJr* com a flag *-O3*, sendo que o tamanho absoluto do primeiro ainda continua menor que o segundo. Para *KetjeSr*, a redução foi de até 18% com *-Os* e 69% com *-O3*. Em todos os processadores, os tamanhos obtidos foram semelhantes.

[3] Veja http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0388g/BABDEBHF.html

Tabela 2. Comparação: Tamanho de texto de código (código compilado) do executável

Utilizando Flag -Os				
Algoritmo	Ref Jr	Jr	Ref Sr	Sr
Tamanho	7186	5690	7066	5796
Utilizando Flag -O3				
Algoritmo	Ref Jr	Jr	Ref Sr	Sr
Tamanho	30106	9348	32402	10086

Tabela 3. Comparação: Dados da contagem de ciclos - flag -Os

	Cortex-A8	**Cortex A15**	**32-bit Cortex-A53**	**64-bit Cortex-A53**
Ref Jr	103432748	75544229	99213259	58861866
Jr	103382644	70264745	97515256	57050181
Ref Sr	1473854399	1016027233	1362340599	827482656
Sr	1423607979	985182579	1343514261	826662312

Tabela 4. Comparação: Dados da contagem de ciclos - flag -O3

	Cortex-A8	**Cortex A15**	**32-bit Cortex-A53**	**64-bit Cortex-A53**
Ref Jr	36394258	27237526	28214047	25694832
Jr	28168087	24243611	24184896	23212240
Ref Sr	489483314	457071392	484010978	396564278
Sr	453691027	425575715	441179836	401887092

Em termos de desempenho, também podemos notar uma diferença entre os resultados dos dois procedimentos. Enquanto para *-Os* tivemos uma melhora de até 7% com *KetjeJr* e de até 3% com *KetjeSr* no Cortex-A15, para *-O3* podemos observar um ganho de 11% e 7% para o mesmo processador, respectivamente. Portanto, pode-se admitir que o resultado ficou num patamar muito próximo, tendo ganhos pequenos em relação a referência. O gráfico da Figura 7 expõe essa comparação para o procedimento de compilação com *-O3*.

6. Conclusão

Este trabalho propõe uma implementação mais compacta do algoritmo de criptografia *Ketje* para chaves e *nonces* de tamanhos múltiplos de 8 bits, que possui menor tamanho do código compilado comparado ao da referência, sem perder em desempenho. Além disso, contribui para a comunidade em geral ao expôr um código de leitura mais fluída.

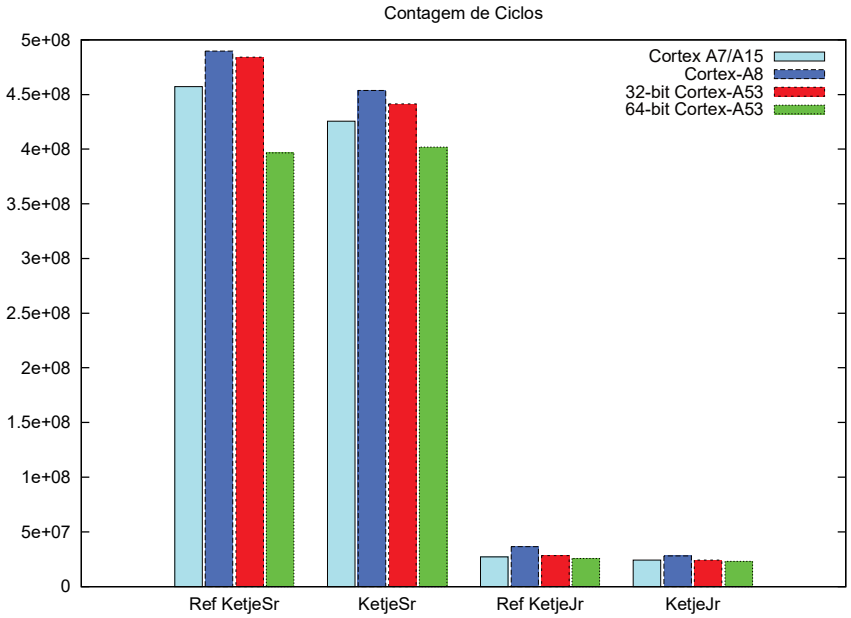

Figura 7. Contagem de Ciclos para KetjeJr e KetjeSr (flag -O3)

Referências

Bertoni, G., Daemen, J., Peeters, M., and Assche, G. (2011a). The Keccak Reference. http://keccak.noekeon.org/.

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

Bertoni, G., Daemen, J., Peeters, M., and Assche, G. (2007). Sponge Functions. ECRYPT Workshop on Cryptographic Hash Functions, May 24-25, 2007.

Bertoni, G., Daemen, J., Peeters, M., Assche, G. and Keer, R. (2014). CAESAR submission: Ketje v1. http://ketje.noekeon.org/.

Bellare, M., Rogaway, P. and Wagner, D. (2003). A conventional authenticated-encryption mode. http://eprint.iacr.org/2003/069/

