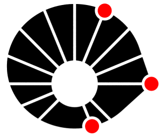

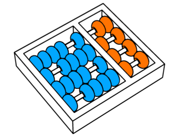

Implementação em software da cifra Elephant

Luis Lacalle *Julio López*

Relatório Técnico - IC-PFG-23-09
Projeto Final de Graduação
2023 - Julho

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

The contents of this report are the sole responsibility of the authors.
O conteúdo deste relatório é de única responsabilidade dos autores.

Implementação em software da cifra Elephant

Luis Lacalle Julio López*

Resumo

Neste trabalho, apresentamos uma implementação em software de Elephant, um algoritmo de criptografia leve projetado como um esquema de autenticação encriptada com dados associados, baseado na construção "encripta-então-autentica" e com modo de operação contador. Seu principal componente é a permutação π_b, uma generalização da permutação utilizada no cifrador PRESENT ($b = 64$) para valores maiores que 64. Apresentamos novas técnicas computacionais para melhorar o desempenho da permutação π_b em três versões: $b = 160$, 176 e 192. Os resultados de nossa implementação em C, no processador M1 Apple (arquitetura ARMv8), mostram uma melhora significativa no desempenho das variantes Dumbo e Jumbo do algoritmo Elephant, ambas alcançando ganhos de até 14 vezes na taxa de encriptação por ciclo, quando comparadas com a implementação de referência. Por fim, notamos que os resultados apresentados neste trabalho também se aplicam para melhorar o desempenho da função de resumo criptográfico SPONGENT.

1 Introdução

Com o crescimento da Internet das Coisas, cresce também a busca por algoritmos criptográficos capazes de entregar um alto nível de segurança com performance, principalmente em dispositivos de baixo poder computacional.

Para isso, o NIST organizou uma competição em 2018, na qual pesquisadores submeteram seus algoritmos para análise e revisão. O objetivo era selecionar o novo padrão para criptografia leve de encriptação autenticada com dados associados. Em 2021, foram apresentados os 10 finalistas desta competição, dentre os quais Elephant [2] foi um dos selecionados.

Projetado para executar de forma paralela, utiliza-se de permutações amplamente conhecidas para compor seu cifrador, sendo elas π_b ($b = 160$ e 176) [3] e Keccak-200 [1]. Neste trabalho, restringimos o estudo às versões do cifrador utilizando π_b, chamadas Dumbo e Jumbo.

No artigo de Reis et al. [7], foi desenvolvido um algoritmo para acelerar a execução da permutação π_b ($b = 64$). A principal ideia foi a decomposição da permutação P (componente central da permutação π_b) como $P = P_0 \circ P_1$, onde P_0 e P_1 são permutações mais simples. A segunda observação importante é $P^2 = P_1 \circ P_0$. Baseados nessas propriedades, para cada duas rodadas de execução da permutação π_b, as duas aplicações da permutação P podem

*Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP.

ser substituídas pelas aplicações das permutações P_0 e P_1, melhorando o desempenho de π_b. Neste trabalho generalizamos essas ideias para serem aplicadas às permutações π_b para valores de b maiores que 64, aplicando-se estas técnicas nas duas versões utilizadas em Elephant e uma terceira, com $b = 192$, como alternativa mais eficiente para Jumbo.

Quanto à estrutura deste trabalho, toda a base teórica do algoritmo é descrita na Seção 2, seguida pela apresentação da implementação e das técnicas utilizadas na Seção 3. Já na Seção 4, estão os detalhes dos testes de performance e resultados obtidos com as implementações. Por fim, o trabalho é concluído com propostas de pesquisas futuras na Seção 5.

2 O algoritmo Elephant

2.1 Introdução

Elephant [2] é um algoritmo de encriptação autenticada com dados associados (AEAD), encripta-então-autentica e com modo de operação contador, tornando-o facilmente paralelizável, como é mostrado na figura 1. Sua arquitetura se baseia em permutações amplamente estudadas e utilizadas tanto em cifragem quanto em funções de resumo.

No artigo de Beyne et al.[2], são apresentadas 3 versões do cifrador com diferentes tamanhos de bloco, sendo elas: Dumbo, de 160 bits, Jumbo de 176 bits e Delirium de 200 bits. Cada uma delas utiliza uma permutação diferente, sendo respectivamente: π_{160}, π_{176} [3] e Keccak [1]. Neste trabalho, serão analisados apenas os modos Dumbo e Jumbo.

Algorithm 1 Algoritmo de cifragem Elephant

Input: $M, K, N, A \in \{0,1\}^c \times \{0,1\}^k \times \{0,1\}^m \times \{0,1\}^a$
Output: $C, T \in \{0,1\}^c \times \{0,1\}^t$

1: $M_1 ... M_{l_M} \leftarrow$ `Partition(M, n)`
2: **for** $i = 1$ `to` l_M **do**
3: $\quad C_i \leftarrow M_i \oplus P(N || 0^{n-l} \oplus \texttt{mask}_K^{i-1,1}) \oplus \texttt{mask}_K^{i-1,1}$
4: **end for**
5: $C \leftarrow \lfloor C_1 \parallel ... \parallel C_{l_M} \rfloor_c$
6: $A_1 ... A_{l_A} \leftarrow$ `Partition(`$N \parallel A \parallel 1$ `, n)`
7: $C_1 ... C_{l_C} \leftarrow$ `Partition(`$C \parallel 1$`, n)`
8: $T \leftarrow A_1$
9: **for** $i = 2$ `to` l_A **do**
10: $\quad T \leftarrow T \oplus P(A_i \oplus \texttt{mask}_K^{i-1,0}) \oplus \texttt{mask}_K^{i-1,0}$
11: **end for**
12: **for** $i = 1$ `to` l_C **do**
13: $\quad T \leftarrow T \oplus P(C_i \oplus \texttt{mask}_K^{i-1,2}) \oplus \texttt{mask}_K^{i-1,2}$
14: **end for**
15: $T \leftarrow P(T \oplus \texttt{mask}_K^{0,0}) \oplus \texttt{mask}_K^{0,0}$
16: **return** $C, \lfloor T \rfloor_t$

- `Partition(X,n)` - particiona X em l_X blocos de n bits, preenchendo o último bloco com 0, caso necessário.
- $\mathtt{mask}_K^{a,b}$ - é o gerador de máscaras da cifragem, definido como:

$$\mathtt{mask}_K^{a,b} = \varphi_2^b \circ \varphi_1^a \circ P(K||0^{n-k}),$$

no qual $\varphi_2 = \varphi_1 \oplus id$, com id sendo a função identidade, P a permutação π_b e φ_1 um `lfsr` de n bits representado por:

$$\varphi_1(X_{160}) = (x_1, x_2, ..., x_{19}, x_0 \lll 3 \oplus x_3 \ll 7 \oplus x_{13} \gg 7) \text{ em Dumbo;}$$
$$\varphi_1(X_{176}) = (x_1, x_2, ..., x_{22}, x_0 \lll 3 \oplus x_3 \ll 7 \oplus x_{19} \gg 7) \text{ em Jumbo.}$$

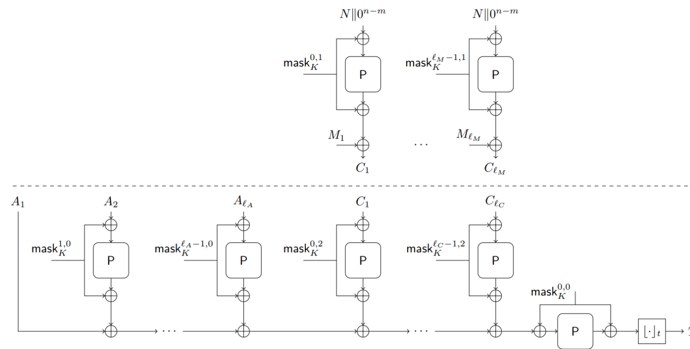

Figura 1: Representação do cifrador Elephant. Fonte:[2]

2.2 Permutação π_b

A permutação π_b, utilizada pelo Elephant nos modos de operação Dumbo e Jumbo, é uma generalização da permutação de PRESENT (b=64) [5], a qual também é aplicada no algoritmo de hash SPONGENT [4], sendo definida como:

Algorithm 2 π_b

Input: $X \in \{0,1\}^b$
Output: $X \in \{0,1\}^b$

1: $c \leftarrow$ `Init`
2: **for** $i = 1$ `to` k **do** ▷ $k = 80, 90$ e 100 para os valores $b = 160, 176$ e 192
3: `CounterStep(`X`,` c`)`
4: `Sbox(`X`)`
5: `P(`X`)`
6: $c \leftarrow$ `lfsr(`c`)`
7: **end for**
8: **return** X

- `lfsr` - baseada no polinômio primitivo $p(x) = x^7 + x^6 + 1$, é calculada como $\texttt{lfsr}(c) = (c_5, c_4, c_3, c_2, c_1, c_0, c_5 \oplus c_6)$, para um contador de 7 bits $c = (c_6, c_5, c_4, c_3, c_2, c_1, c_0)$. No modo Dumbo, c é inicializado com `0b1110101`, enquanto que no Jumbo é inicializado com `0b1000101`.

- `CounterStep` - realiza um `XOR` no primeiro e último byte da mensagem, como descrito abaixo:

$$X \leftarrow X \oplus (c \parallel 0^{b-14} \parallel \texttt{rev}(c)),$$

  sendo $\texttt{rev}(c)$ a função que reverte os bits de c.

- `Sbox` - faz uma substituição dos $\frac{n}{4}$ grupos de 4 bits do bloco através da Tabela 1, adicionando confusão ao algoritmo.

Tabela 1: S-box do Spongent-π

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
S-box(x)	E	D	B	0	2	1	4	F	7	A	8	5	9	C	3	6

  Tipicamente, essa fase é implementada como uma tabela pré-computada em memória. Porém, nossa proposta utiliza circuitos booleanos para realizar a substituição dos bits, garantindo tempo constante de processamento e economia de memória.

- `P` - consiste na permutação dos bits da mensagem, de acordo com a seguinte função:

$$\texttt{P}(j) = \begin{cases} \frac{b}{4} \times j \mod b-1, & 0 \leq j < b-1, \\ b-1, & j = b-1; \end{cases}$$

  sendo $\texttt{P}(j)$ a posição em que o j-ésimo bit será colocado no bloco.

  Para exemplificar a permutação, podemos dispôr os bits de um bloco de mensagem em uma matriz $4 \times \frac{b}{4}$ e observar que o resultado da permutação é uma ordenação vertical dos bits na matriz. Para $b = 192$, ilustramos a permutação P:

$$X = \begin{bmatrix} x_{00} & x_{01} & x_{02} & \cdots & x_{46} & x_{47} \\ x_{48} & x_{49} & x_{50} & \cdots & x_{94} & x_{95} \\ x_{96} & x_{97} & x_{98} & \cdots & x_{142} & x_{143} \\ x_{144} & x_{145} & x_{146} & \cdots & x_{190} & x_{191} \end{bmatrix} \quad P(X) = \begin{bmatrix} x_{00} & x_{04} & x_{08} & \cdots & x_{184} & x_{188} \\ x_{01} & x_{05} & x_{09} & \cdots & x_{185} & x_{189} \\ x_{02} & x_{06} & x_{10} & \cdots & x_{186} & x_{190} \\ x_{03} & x_{07} & x_{11} & \cdots & x_{187} & x_{191} \end{bmatrix}$$

  Isto servirá como base para a implementação apresentada neste trabalho.

3 Implementação

Escritas na linguagem C, as implementações utilizaram a seguinte estrutura para representar palavras de 48 bits:

Listagem 1.1: Estrutura utilizada para permutação π

```
typedef struct word48 {
    uint32_t block[2];
} word;

uint32_t X[6];
word W[4];
```

Esta estrutura é necessária para aplicar as implementações apresentadas na Figura 2.

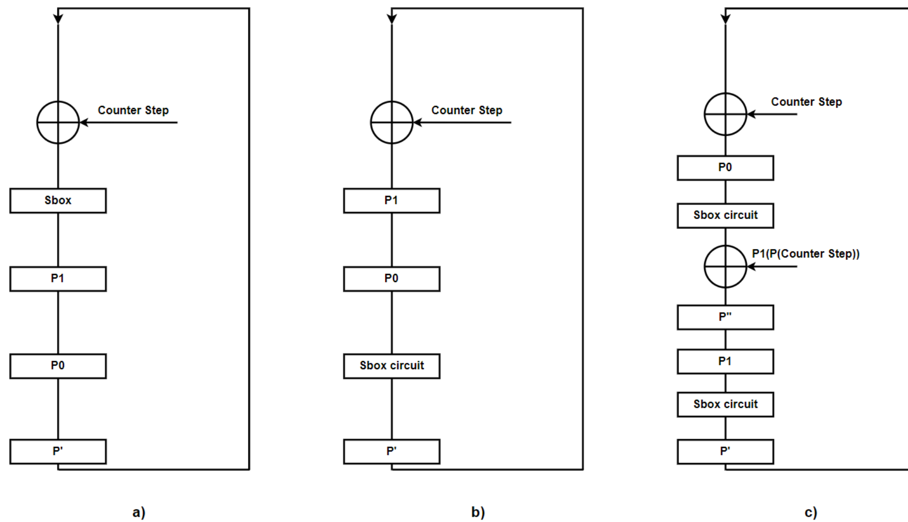

Figura 2: Esquemas das implementações otimizadas da permutação π_b. a) substituição de `P` pelas permutações P_1 e P_0. b) implementação das caixas de substituição como circuitos booleanos. c) inversão de P_0 e P_1, reduzindo o número de execuções de P_0 e P_1 em π_b para cada uma das $\frac{k}{2}$ rodadas.

Assim, os três algoritmos apresentados na Figura 2 serão discutidas nas próximas seções.

3.1 Permutações P_0 e P_1

As permutações P_0 e P_1 foram apresentadas em [7], sendo aplicadas na otimização de PRESENT com $b = 64$. No contexto de Elephant, iremos mostrar como generalizar esta técnica para $b = 160, 176$ e 192.

Listagem 1.2: Macros em C das permutações P_0 e P_1 com palavras de 32 bits. Adaptado de [7].

```
#define PRESENT_PERMUTATION_P0(X0, X1, X2, X3) \
    t = (X0 ^ (X1<<1) ) & 0xAAAAAAAA; \
    X0 = X0^t ; X1 = X1 ^ (t>>1); \
    t = (X2 ^ (X3<<1) ) & 0xAAAAAAAA; \
    X2 = X2^t ; X3 = X3 ^ (t>>1); \
    t = (X0 ^ (X2<<2) ) & 0xCCCCCCCC; \
    X0 = X0^t ; X2 = X2 ^ (t>>2); \
    t = (X1 ^ (X3<<2) ) & 0xCCCCCCCC; \
    X1 = X1^t ; X3 = X3 ^ (t>>2); \
#define PRESENT_PERMUTATION_P1(X0, X1, X2, X3) \
    t = (X0 ^ (X1<<4) ) & 0xF0F0F0F0; \
    X0 = X0^t ; X1 = X1 ^ (t>>4); \
    t = (X2 ^ (X3<<4) ) & 0xF0F0F0F0; \
    X2 = X2^t ; X3 = X3 ^ (t>>4); \
    t = (X0 ^ (X2<<8) ) & 0xFF00FF00; \
    X0 = X0^t ; X2 = X2 ^ (t>>8); \
    t = (X1 ^ (X3<<8) ) & 0xFF00FF00; \
    X1 = X1^t ; X3 = X3 ^ (t>>8); \
```

Para utilizar corretamente estas permutações no nosso algoritmo, foi necessário reorganizar o bloco da mensagem em 4 palavras de 48 bits, permutando as sequências de 16 bits verticalmente pelas palavras, a fim de criar blocos de 4×16 bits, como em PRESENT. Caso o tamanho do bloco seja menor que 192, preenche-se os últimos bits do bloco com 0. A seguir, ilustramos a técnica aplicada a um bloco de 192 bits:

$$M = \begin{bmatrix} x_{00} \; x_{01} \; x_{02} \; x_{03} \; \ldots \; x_{188} \; x_{189} \; x_{190} \; x_{191} \end{bmatrix},$$

$$M'_1 = \begin{bmatrix} x_{00} & x_{01} & \ldots & x_{14} & x_{15} \\ x_{16} & x_{17} & \ldots & x_{30} & x_{31} \\ x_{32} & x_{33} & \ldots & x_{46} & x_{47} \\ x_{48} & x_{49} & \ldots & x_{62} & x_{63} \end{bmatrix} M'_2 = \begin{bmatrix} x_{064} & x_{065} & \ldots & x_{078} & x_{079} \\ x_{080} & x_{081} & \ldots & x_{094} & x_{095} \\ x_{096} & x_{097} & \ldots & x_{110} & x_{111} \\ x_{112} & x_{113} & \ldots & x_{126} & x_{127} \end{bmatrix} M'_3 = \begin{bmatrix} x_{128} & x_{129} & \ldots & x_{142} & x_{143} \\ x_{144} & x_{145} & \ldots & x_{158} & x_{159} \\ x_{160} & x_{161} & \ldots & x_{174} & x_{175} \\ x_{176} & x_{177} & \ldots & x_{190} & x_{191} \end{bmatrix},$$

$$M' = \begin{bmatrix} M'_1 \; M'_2 \; M'_3 \end{bmatrix}.$$

Em seguida, a permutação é feita como em PRESENT de 64 bits [7], aplicando $P_0 \circ P_1$ a cada bloco M'_i de M', resultando na seguinte disposição:

$$P(M) = \begin{bmatrix} x_{00} & x_{04} & x_{08} & \ldots & x_{184} & x_{188} \\ x_{01} & x_{05} & x_{09} & \ldots & x_{185} & x_{189} \\ x_{02} & x_{06} & x_{10} & \ldots & x_{186} & x_{190} \\ x_{03} & x_{07} & x_{11} & \ldots & x_{187} & x_{191} \end{bmatrix}.$$

Ao fim de cada rodada, é necessário permutar novamente as sequências de 16 bits, preparando o bloco para a próxima rodada. Para blocos com preenchimentos de bits 0, essa permutação precisa realocar os bits 0 para o fim do bloco, agrupando a mensagem. Assim, essa permutação é feita através da seguinte função:

$$P' = \begin{bmatrix} X_{00} & X_{01} & X_{02} \\ X_{03} & X_{04} & X_{05} \\ X_{06} & X_{07} & X_{08} \\ X_{09} & X_{10} & X_{11} \end{bmatrix} \rightarrow \begin{bmatrix} X'_{00} & X'_{01} & ... & X'_{10} & X'_{11} \end{bmatrix} \rightarrow \begin{bmatrix} X'_{00} & X'_{04} & X'_{08} \\ X'_{01} & X'_{05} & X'_{09} \\ X'_{02} & X'_{06} & X'_{10} \\ X'_{03} & X'_{07} & X'_{11} \end{bmatrix},$$

onde X'_i é a i-ésima sequência de 16 bits da mensagem, deslocando-se os bits de preenchimento ao final do bloco e agregando as palavras seguintes para preencher os buracos. Vale notar também que, para $b = 192$, não há estes preenchimentos na mensagem, o que torna a permutação P' equivalente à:

$$P' = \begin{bmatrix} X_{00} & X_{01} & X_{02} \\ X_{03} & X_{04} & X_{05} \\ X_{06} & X_{07} & X_{08} \\ X_{09} & X_{10} & X_{11} \end{bmatrix} \rightarrow \begin{bmatrix} X_{00} & X_{04} & X_{08} \\ X_{01} & X_{05} & X_{09} \\ X_{02} & X_{06} & X_{10} \\ X_{03} & X_{07} & X_{11} \end{bmatrix}.$$

3.2 Etapa de substituição

Na etapa de substituição (sbox), o objetivo foi encontrar o menor número de instruções para a função booleana que representa a tabela pré-calculada. Como essa tarefa é computacionalmente custosa (combinatória no tamanho do circuito), foi utilizada a ferramenta **sboxgates** [6] para gerar o seguinte circuito:

```
#define sbox32(x3,x2,x1,x0) \
    t0 = x1;                \
    t1 = ~x2;               \
    t2 = t1 & x1;           \
    t3 = x3 ^ t2;           \
    x0 = t3 ^ x0;           \
    t2 = x0 ^ t1;           \
    t3 = t2 | x3;           \
    t4 = t3 ^ x1;           \
    x1 = t4 ^ x2;           \
    t3 = ~x1;               \
    t4 = t3 | x3;           \
    x2 = t2 ^ t4;           \
    t2 = t1 & x2;           \
    t3 = t2 ^ x3;           \
    t4 = x1 & t0;           \
    x3 = t3 ^ t4;           \
```

Desta forma, é possível realizar esta etapa através de 2 execuções do circuito em palavras de 32 bits, com cada execução resultando em até 32 substituições simultâneas.

3.3 Inversão das permutações: $P_1 \circ P_0$

Para a última implementação, baseando-se na propriedade $P^2_{PRESENT} = P_1 \circ P_0$ [7], foi implementada uma permutação adicional P'' para satisfazer $P^2 = P_1 \circ P'' \circ P_0$. Esta permutação P'', semelhante à P', realiza o agrupamento da mensagem em sequências de 16 bits verticalmente para possibilitar a aplicação correta de P_1, porém lidando com os bits de preenchimento permutados por P_0. Para preencher esses buracos, movimenta-se os respectivos bits das próximas palavras. A seguir ilustramos a aplicação de P'':

$$P'' = \begin{bmatrix} X_{00} & X_{01} & X_{02} \\ X_{03} & X_{04} & X_{05} \\ X_{06} & X_{07} & X_{08} \\ X_{09} & X_{10} & X_{11} \end{bmatrix} \rightarrow \begin{bmatrix} X''_{00} & X''_{01} & ... & X''_{10} & X''_{11} \end{bmatrix} \rightarrow \begin{bmatrix} X''_{00} & X''_{04} & X''_{08} \\ X''_{01} & X''_{05} & X''_{09} \\ X''_{02} & X''_{06} & X''_{10} \\ X''_{03} & X''_{07} & X''_{11} \end{bmatrix},$$

onde X''_i é a i-ésima sequência de 16 bits da mensagem, deslocando-se os bits de preenchimento ao final do bloco e agregando os bits respectivos das palavras seguintes, preenchendo os buracos. Vale notar novamente que não é necessário lidar com os preenchimentos no caso de $b = 192$, tornando $P'' = P'$. Isso faz com que só seja necessário a implementação de P', reduzindo o tamanho do código e acelerando a execução do algoritmo.

Por fim, a etapa de $CounterStep$ em [7] é realizada uma vez antes de P_0 e uma vez após P_1 aplicando $P(CounterStep)$. No nosso caso, adicionamos uma modificação, realizando a segunda etapa antes de P_1. Isto implica na necessidade de adicionar uma permutação P_1^{-1} em $P(CounterStep)$ para manter o alinhamento, chegando na expressão:

$$P_1^{-1} \circ P(c \parallel 0^{b-14} \parallel \texttt{rev}(c)) = P_1 \circ P(c \parallel 0^{b-14} \parallel \texttt{rev}(c)).$$

O motivo que justifica esta decisão é evitar a realização do cálculo de $P_1 \circ P(c \parallel 0^{b-14} \parallel \texttt{rev}(c))$ toda rodada. Para isso, este valor é pré computado e armazenado em uma tabela de $\frac{k}{2}$ elementos, notando que é necessário armazenar apenas os 16 bits iniciais e finais, uma vez que a disposição dos 7 bits de c e $\texttt{rev}(c)$ em $P_1 \circ P(c \parallel 0^{b-14} \parallel \texttt{rev}(c))$ estão contidas nos extremos do bloco, indicando que a operação ocorrerá na primeira e última palavra apenas, como mostrado a seguir:

$$\begin{bmatrix} c_{00} \; c_{04} \; 0 \; 0 \; c_{01} \; c_{05} \; 0 \; 0 \; c_{02} \; c_{06} \; 0 \; 0 \; c_{03} \; 0 \; 0 \; 0 \; ... \; 0 \; 0 \; 0 \; c_{03} \; 0 \; 0 \; c_{06} \; c_{02} \; 0 \; 0 \; c_{05} \; c_{01} \; 0 \; 0 \; c_{04} \; c_{00} \end{bmatrix}.$$

4 Resultados

As implementações foram avaliadas em um ambiente com processador **Apple M1** (ARMv8), octa-core ARM 3.2 GHz.

Para realizar os testes de performance de π_b, as implementações foram executadas 100 vezes em sequência, mensurando-se o total de ciclos destas execuções e obtendo a média de cada execução. Foram realizadas 10 medições para cada versão e tamanho de bloco, obtendo-se a mediana destas medições. As implementações testadas foram:

- Ref - uma implementação em C do código de referência citado no artigo de Beyne et al. [2];

- I - uma implementação em C da Figura 2 a), aplicando-se as permutações P_0 e P_1 no lugar de P;
- II - uma implementação em C da Figura 2 b), aplicando os circuitos booleanos ao invés de tabelas de consulta na etapa de substituição;
- III - uma implementação em C da Figura 2 c), com a aplicação de $P_1 \circ P_0$ e da tabela de consulta para $P_1 \circ P(c \parallel 0^{b-14} \parallel \texttt{rev}(c))$.

Tabela 2: Desempenho das implementações π_b em ciclos.

Implementações em C	b=160	b=176	b=192
Ref	39059	36387	43899
I	5814	6656	7271
II	3541	4313	4723
III	2656	3169	3105

Como podemos ver, as versões com a implementação III foram as que apresentaram melhor desempenho. Nos casos de $b = 160$ e 192, alcançaram uma redução de até 93% no número de ciclos em comparação à implementação de referência. Com relação ao tamanho do bloco, $b = 160$ foi o que apresentou melhor desempenho em ciclos, enquanto que $b = 192$ a menor taxa de ciclos por byte permutado dentre os 3.

Para os testes de performance de cifragem de Elephant, foi confrontado o algoritmo de referência de Elephant com a nossa implementação III. Utilizando a mesma estratégia de obtenção das medidas, foram avaliados 4 cenários de tamanho de mensagem e dados associados:

- Mensagem M com 64 bytes e dados associados A com 64 bytes;
- Mensagem M com 64 bytes e dados associados A com 1024 bytes;
- Mensagem M com 1024 bytes e dados associados A com 64 bytes;
- Mensagem M com 1024 bytes e dados associados A com 1024 bytes.

Tabela 3: Taxa de encriptação de Elephant referência, em ciclos/byte.

M (bytes)	**A (bytes)**	**b** = 160	**b** = 176	**b** = 192
64	64	7988	6766	6765
64	1024	37506	33816	31355
1024	64	4188	3807	3497
1024	1024	6030	5498	5032

Tabela 4: Taxa de encriptação de Elephant com π_b otimizado, em ciclos/byte.

M (bytes)	**A (bytes)**	**b** = 160	**b** = 176	**b** = 192
64	64	547	463	463
64	1024	2555	2304	2138
1024	64	286	260	239
1024	1024	412	375	343

Podemos observar que Elephant apresentou menores taxas de encriptação quando usados blocos de 192 bits, apresentando taxas de até 239 ciclos/bytes para mensagens grandes com poucos dados associados. Isto sugere que implementar Jumbo com 192 bits, ao invés de 176 bits, representa um ganho de até 9% na taxa de encriptação de Elephant. Além disso, comparado à referência, nossa implementação apresentou ganhos nas taxas de encriptação de até 14 vezes em todos os cenários testados.

5 Conclusão

Este trabalho apresentou diferentes técnicas de otimização para o algoritmo Elephant. Foram descritas 3 implementações da permutação π_b, cada uma com versões de 160, 176 e 192 bits, sendo esta última a que apresentou melhor desempenho nos testes e sendo uma melhor alternativa para a versão Jumbo de Elephant. Como trabalho futuro, pretende-se usar estas técnicas para avaliar o desempenho de Elephant em diferentes processadores ARM de 32 bits, como também aprimorar o desempenho do algoritmo SPONGENT.

Referências

[1] G. Bertoni, J. Daemen, and M. Peeters. Keccak sponge function family main document. *Submission to NIST (Round 2)*, 3, 01 2009.

[2] T. Beyne, Y. L. Chen, C. Dobraunig, and B. Mennink. Dumbo, jumbo, and delirium: Parallel authenticated encryption for the lightweight circus. *IACR Transactions on Symmetric Cryptology*, 2020(S1):5–30, Jun. 2020.

[3] A. Bogdanov, M. Knežević, G. Leander, D. Toz, K. Varıcı, and I. Verbauwhede. SPONGENT: A lightweight hash function. In B. Preneel and T. Takagi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2011*, pages 312–325, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[4] A. Bogdanov, M. Knezevic, G. Leander, D. Toz, K. Varici, and I. Verbauwhede. SPONGENT: The design space of lightweight cryptographic hashing. Cryptology ePrint Archive, Paper 2011/697, 2011. `https://eprint.iacr.org/2011/697`.

[5] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe. PRESENT: An ultra-lightweight block cipher. In P. Paillier and I. Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007*, pages 450–466, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[6] M. Dansarie. sboxgates: A program for finding low gate count implementations of s-boxes. *Journal of Open Source Software*, 6(62):2946, 2021.

[7] T. B. S. Reis, D. F. Aranha, and J. López. PRESENT Runs Fast: Efficient and Secure Implementation in Software. In W. Fischer and N. Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017*, pages 644–664, Cham, 2017. Springer International Publishing.