Implementação em Software da Cifra LowMC

Roberto Cabral[1], **Julio López**[2]

[1]Universidade Federal do Ceará - UFC/Quixadá

[2]Universidade Estadual de Campinas - UNICAMP

rbcabral@ufc.br, jlopez@ic.unicamp.br

Abstract. *Several symmetric primitives proposals have been built to minimize nonlinear operations, including the LowMC. Although LowMC is a very attractive block cipher for applications that benefit from its low multiplicative complexity, it operates directly on bits, making it very expensive in software. Since some applications need to process multiple messages at a time, by taking advantage of the AVX vector instruction set, we built a parallel version of lowMC that processes multiple messages simultaneously, facilitating its implementation in software and improving the performance by encryption. We achieve a 2.3× speedup when processing eight messages at a time and 3.3× when processing 16 messages for the LowMC instance with full Sbox and 129-bit security. Additionally, for conventional architectures that do not have vector instructions, we built a native implementation of LowMC which is about 25% more efficient than the state-of-art.*

Resumo. *Nos últimos anos surgiram várias propostas de primitivas simétricas construídas visando a minimização das operações não lineares, dentre elas, a família de cifras de bloco LowMC. Embora o LowMC seja uma primitiva muito atraente para aplicações que se beneficiam da sua baixa complexidade multiplicativa, o fato dele operar diretamente sobre bits torna-o muito custoso em software. Visto que algumas aplicações executam múltiplas mensagens por vez; tirando proveito do conjuntos de instruções vetoriais AVX, adaptou-se o algoritmo de modo a processar múltiplas mensagens por vez, facilitando sua implementação em software e melhorando o desempenho por encriptação. Conseguindo uma aceleração de 2,3× quando encriptadas oito mensagens por vez e de 3,3× quando encriptadas 16 mensagens para a instância do LowMC que processa todos os bits do estado por caixas de substituição e segurança de 129 bits. Adicionalmente, pensando em arquiteturas convencionais que não possuem instruções vetoriais, foi desenvolvida uma implementação nativa do LowMC cerca de 25% mais eficiente que o estado da arte.*

1. Introdução

As primitivas simétricas são tão antigas quanto o conceito de criptografia, como a clássica cifra de César que consiste no deslocamento de n posições de cada caractere do texto claro. Segundo Shannon, uma boa cifra simétrica é formada por dois princípios básicos: confusão e difusão (Shannon 1945). Para fornecer esses princípios básicos, as cifras simétricas são normalmente construídas a partir de operações lineares e não lineares.

As operações lineares e não lineares possuem custos semelhantes em implementações de *software* e *hardware*, mas quando implementadas em um protocolo de computação multiparte (MPC - *Secure multi-party computation*), ou em um esquema de encriptação homomórfica (FHE - *Fully homomorphic encryption*) o cenário é completamente diferente . Nestes contextos, operações lineares saem "quase que de graça" , visto que envolvem computação local, enquanto operações não lineares são substancialmente mais caras por envolverem comunicação entre as partes (Albrecht et al. 2015).

Esse cenário motivou a construção de cifras simétricas que minimizem as operações não lineares e concentrem boa parte do processamento em operações lineares. Assim, nos últimos anos surgiram várias cifras com essas características, como LowMC (Albrecht et al. 2015), MiMC (Albrecht et al. 2016), Vision (Aly et al. 2020) and RAIN (Dobraunig et al. 2021).

Dentre as aplicações que se beneficiam dessas cifras, pode-se destacar as assinaturas digitais construídas a partir de provas de conhecimento zero não interativas. Essa classe de esquemas de assinaturas digitais pós-quânticos foi proposta em 2016, por Chase e outros (Chase et al. 2017) e derivam sua segurança inteiramente da segurança das primitivas simétricas. Nestes esquemas, uma assinatura é uma prova de conhecimento zero não interativa.

O principal algoritmo que utiliza esse esquema é o Picnic (Chase et al. 2017), algoritmo candidato da competição do NIST que visa padronizar algoritmos criptográficos assimétricos resistentes à computadores quânticos. Em seu último relatório (Alagic et al. 2020), o NIST descreve o Picnic como um algoritmo que ainda não está maduro o suficiente para ser padronizado, mas que tem potencial de melhorar muito seu desempenho e a confiança em sua segurança em um futuro próximo; o que ressalta a importância de buscar estratégias para melhorar o desempenho e a segurança do Picnic.

A primitiva simétrica usada no Picnic é o LowMC, que é uma família de cifras simétricas altamente parametrizadas, podendo ter um número maior ou menor de caixas de substituição (caixas-S) de acordo com os parâmetros escolhidos. Em (Kales et al. 2017), Kales e outros apresentaram uma técnica de implementação que melhora significativamente o tempo de execução de instâncias do LowMC com um baixo número de caixas-S, conseguindo reduzir em até quatro vezes o tempo de processamento na instância da cifra usada no Picnic1 (instância do LowMC com 10 caixas-S). Entretanto, essa técnica não é efetiva na instância usada no Picnic3, uma vez que todos os bits do estado são processados pelas caixas-S. Quando todos os bits do estado são processados pelas caixas-S, dizemos que a camada de substituição do algoritmo é completa (caixas-S completas).

Todas as versões do Picnic usam instâncias do LowMC como cifra de bloco para gerar o par de chaves e para construir uma prova de conhecimento zero da chave. Dessa forma, o custo da assinatura e da verificação, bem como o tamanho da prova, depende fortemente da cifra de bloco usada (Kales and Zaverucha 2020). Analisando o algoritmo do Picnic, é possível observar que a maior parte do processamento está diretamente relacionado com a execução do LowMC. Por exemplo, na assinatura do Picnic3 uma versão do LowMC é executada dentro das funções *compute_aux* e *mpc_simulate*; cada uma dessas funções são executadas, independentemente, 219, 329 e 438 vezes para as seguranças L1,

L3 e L5, respectivamente (Greg et al. 2020).

Uma das principais dificuldades na implementação do LowMC é que as operações são definidas sobre bits e não sobre registradores (grupo de bits); tendo como resultado, em muitos casos, uma implementação lenta em *software*. Visto que algumas aplicações, como o Picnic, executam diversas encriptações independentes, é possível adaptar o algoritmo para processar múltiplas mensagens por vez, facilitando a implementação e melhorando o desempenho por encriptação. Para tal, este trabalho apresenta novas implementações eficientes para a cifra LowMC com caixas-S completas baseadas na técnica *bitslice*, originalmente desenvolvida por (Biham 1997) para a cifra de bloco DES.

Contribuições. Neste trabalho, foi analisada a estrutura interna do LowMC com caixas-S completas para reorganizar o estado de modo a facilitar sua implementação em *software*. Essa instância do LowMC é usada no Picnic3, que é a versão com melhor custo benefício atualmente (Kales and Zaverucha 2020). Especificamente, foram construídas as seguintes versões:

1. Tirando proveito do conjunto de instruções vetoriais `AVX` foram construídas diferentes versões paralelas da cifra simétrica LowMC. Foram implementadas uma versão 8-way usando registradores de 128 bits, que encripta oito mensagens por vez com uma aceleração de 2,3 e uma versão 16-way que usa registradores de 256 bits para encriptar 16 mensagens por vez com uma aceleração de 3,3.
2. Pensando em arquiteturas convencionais que não possuem instruções vetoriais, foi implementada uma versão do LowMC usando instruções x86 nativas com um desempenho cerca de 25% melhor quando comparada ao estado da arte.

O restante do texto está organizado como segue: na Seção 2 é apresentado uma descrição da cifra de bloco LowMC; na Seção 3 são apresentadas as principais instruções vetoriais utilizadas neste trabalho; na Seção 4 são detalhadas as técnicas de implementação usadas nas implementações vetorizadas do LowMC; na Seção 5 é apresentada a implementação nativa do LowMC; na Seção 6 é feita uma comparação entre as diferentes implementações do LowMC e na Seção 7 são descritas as conclusões e os trabalhos futuros.

2. LowMC

LowMC é uma cifra de bloco baseada na estrutura SPN (*Substitution–Permutation Network*) que é uma idealização direta do conceito de difusão e confusão introduzidos por Shannon para construção de boas cifras simétricas (Albrecht et al. 2015). A cifra LowMC foi construída pensando em aplicações onde operações lineares são muito mais baratas que operações não lineares, como Computação Multiparte (MPC), Provas de Conhecimento Zero (ZKP) e Computação Homomórfica (FHE).

Essa cifra foi projetada para otimizar a quantidade e a profundidade multiplicativa, logo, as operações não lineares são minimizadas ao máximo para um maior desempenho no contexto em que foi proposta. A cifra LowMC é uma cifra altamente parametrizável, sendo composta pelos seguintes parâmetros. Um bloco de tamanho n bits, uma quantidade m de caixas-S, uma chave de tamanho k bits, e um número de rodadas r.

O processo de encriptação do LowMC consiste nos seguintes passos:

1. O primeiro passo é fazer um *key whitening* de chave a partir do texto claro (essa técnica consiste em aplicar a operação XOR entre a chave e o texto claro), para aumentar a complexidade de um ataque de força bruta.
2. A seguir, são executadas várias rodadas de encriptação, de acordo com o número r de rodadas, cada rodada é composta de:
   (a) Caixa-S: consiste de m aplicações paralelas da função de substituição S que processa 3 bits, onde m é a quantidade de caixas-S. Assim, esta camada é composta pelos primeiros $3m$ bits do estado. Os $n-3m$ bits restantes do estado são a identidade. Nas instâncias do LowMC com caixas-S completas todos os bits do estado são processados nesta camada, isto é $3m = n$. Abaixo é descrita a especificação da função S (a caixa-S de 3 bits). Dados três bits do estado, a, b e c, $S(a, b, c)$ é computado por:

$$\boxed{S(a,b,c) \;=\; (a \oplus bc, a \oplus b \oplus ac, a \oplus b \oplus \; c \; \oplus \; ab)} \tag{1}$$

   (b) Adição de constante: é adicionada uma constante de rodada ao estado; o vetor de constante é gerado previamente, de forma uniformemente aleatória, e é único para cada instância do LowMC.
   (c) Camada linear: é a multiplicação do estado por uma matriz binária $n \times n$; a matriz é gerada, previamente, de forma uniformemente aleatória para cada instância do LowMC, com a restrição de que deve ser uma matriz inversível.
   (d) Adição de chave: a chave de rodada é adicionada ao estado (por meio de um XOR). Para gerar as chaves de rodada a chave mestra é multiplicada por uma matriz binária inversível $n \times k$, gerada previamente de forma uniformemente aleatória para cada instância do LowMC.

No Algoritmo 1 é apresentado o pseudocódigo da função de encriptação do LowMC. São usadas as seguintes variáveis: o texto claro e o estado são representados por pt e s, respectivamente, e possuem n bits; a chave key, que possui k bits; a quantidade r que determina quantas rodadas serão executadas durante a encriptação.

Algoritmo 1 LowMC
(Albrecht et al. 2015)

function LowMC(pt, key)
    $s = pt \oplus$ `MultiplyGF2Matrix`(`KMatrix`$(0), key$)
    for $i = 0$ *to* r **do**
        $s =$ `Sboxlayer`(s)
        $s =$ `MultiplyGF2Matrix`(`LMatrix`$(i), s)$
        $s = s \oplus$ `Constants`(i)
        $s = s \oplus$ `MultiplyGF2Matrix`(`KMatrix`$(i), s)$
    end for
    return s
end function

3. Conjunto de Instruções

3.1. Conjunto de Instruções Vetoriais

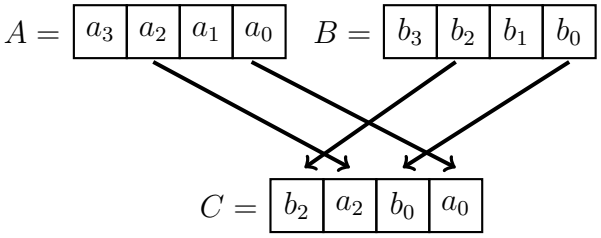
A maioria dos processadores atuais possuem unidades funcionais com capacidade de processar um conjunto de dados com a execução de uma única instrução; esse tipo de processamento encaixa-se no paradigma conhecido como SIMD (*Single Instruction Multiple Data*), introduzido em (Flynn 1972). Atualmente, vários processadores apresentam em sua arquitetura alguns conjuntos de instruções, por exemplo, os conjuntos de instruções `SSE`, `AVX/AVX2`, `AVX-512` nos processadores Intel e `NEON` nos processadores ARM.

As primeiras instruções vetoriais foram direcionadas ao processamento gráfico, mas esse cenário vem mudando nos últimos anos e agora está disponível uma vasta diversidade de instruções. Muitos trabalhos têm utilizado instruções vetoriais para implementação de algoritmos criptográficos, como (Faz-Hernández et al. 2015), (Orisaka et al. 2018), (Cabral and López 2018). Neste trabalho foi utilizado o conjunto de instruções AVX2 para implementar de forma eficiente o algoritmo LowMC; a seguir serão descritas as principais instruções utilizadas neste trabalho.

1. Acesso a memória:
   (a) `LOAD/STORE`. Essas funções permitem carregar/armazenar um conjunto de dados de/para um endereço de memória de/para um registrador vetorial. Vale a pena mencionar que o endereço de memória deve estar alinhado para 32 bits, ou seja, o valor do endereço deve ser um múltiplo de 32, caso contrário o desempenho da aplicação é afetado.
2. Funções lógicas:
   (a) `XOR`, `AND`. Essas funções calculam operações lógicas entre dois registradores A e B produzindo um novo registrador C.

$$A = \boxed{a_3}\boxed{a_2}\boxed{a_1}\boxed{a_0}$$
$$\oplus \quad \oplus \quad \oplus \quad \oplus$$
$$B = \boxed{b_3}\boxed{b_2}\boxed{b_1}\boxed{b_0}$$
$$= \quad = \quad = \quad =$$
$$C = \boxed{c_3}\boxed{c_2}\boxed{c_1}\boxed{c_0}$$

3. Combinação de registradores:
   (a) `UNPACKLO/UNPACKHI`. Essas instruções produzem um registrador C a partir da intercalação das palavras da parte alta/baixa de dois registradores A e B.



As instruções aqui apresentadas operam tanto sobre registradores de 128 bits quanto sobre registradores de 256 bits. Elas não representam todas as instruções pertencentes ao conjunto de instruções AVX2; é recomendado consultar (Corporation 2011) para informação adicional sobre esse conjunto.

3.2. Conjunto de instrução x86

O conjunto de instruções x86-64 (Guide 2011) bits é uma versão de 64 bits do conjunto de instruções x86 que expandiu os registradores de 32 para 64 bits; permitindo que todas as operações aritméticas, lógicas e de memória operarem diretamente sobre inteiros de 64 bits. Além de aumentar o tamanho dos registradores de propósito geral, também houve um aumento significativo no número de registradores; saltando de oito para dezesseis. Dessa forma, agora é possível manter mais informações armazenadas em registradores.

No contexto deste trabalho, destaca-se a instrução `popcnt` que retorna o número de bits ativos em um registrador. Essa instrução pode ser usada para calcular a paridade de um registrador; visto que se o número de uns em um registrador for par, a paridade será zero e caso contrário, a paridade será um.

4. Implementação vetorizada

O LowMC opera sobre bits, o que é muito custoso para implementações em *software*. Visto a necessidade, para algumas aplicações, de executar inúmeras encriptações independentes, foi possível tirar proveito da técnica de implementação *bitslice* para organizar o estado de modo a processar Q entradas por vez; acelerando o *throughput* dos texto encriptados por execução e tornando a implementação mais adequada para a arquitetura alvo.

O *bitslicing* é uma técnica para implementação de algoritmos criptográficos que foi proposta por Biham (Biham 1997) para o algoritmo DES. Posteriormente essa técnica começou a ser usada também no AES (Rebeiro et al. 2006, Matsui and Nakajima 2007, Käsper and Schwabe 2009), visando aumentar o desempenho e a segurança, pois com o *bitslice* as transformações do AES podem ser descritas na forma de circuitos lógicos, que operam bit a bit, fazendo com que o tempo de execução das instruções seja independente dos valores de entrada, o que torna o algoritmo imune a ataques de tempo (Käsper and Schwabe 2009).

A primeira parte do processo para construir tal implementação é criar um estado principal, que será composto pela entrada de Q textos de tamanho N, que serão processados paralelamente. O objetivo é organizar o estado principal de forma que cada palavra a_i contenha Q bits e seja composta pela concatenação do bit Q_n de cada um dos estados. Na Figura 1 é descrito como será a formação do estado principal. Uma vez organizado o estado em N palavras de tamanho Q, é necessário carregá-lo nos registradores vetoriais.

A seguir serão descritas as implementações da instância do LowMC com tamanho de bloco e chave de 129 bits, quatro rodadas e caixas-S completas. Foram implementadas duas versões para cada nível de segurança (129 e 192 bits), a primeira versão usa registradores de 128 bits para processar paralelamente oito mensagens, enquanto a segunda versão utiliza registradores de 256 bits para processar 16 mensagens por vez. Vale destacar que os trechos de códigos apresentados foram organizados para serem legíveis e, consequentemente, essa versão é aproximadamente 25% mais lenta do que a versão otimizada; a versão otimizada usa as mesmas operações com uma melhor organização visando aproveitar melhor as unidades funcionais disponíveis na arquitetura alvo. Todas as implementações deste trabalho estão disponíveis no *github*[1].

[1]https://github.com/rbCabral/LowMC_FULL_SBOX

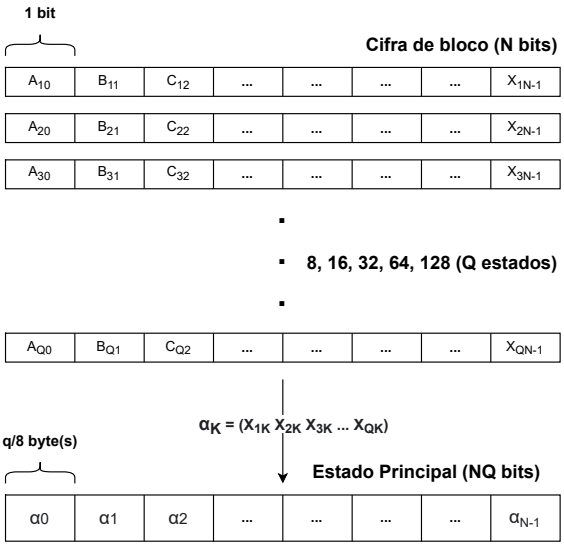


Figura 1. Formação do estado principal

4.1. Implementação 8-way 129 bits

Visando potencializar o uso das instruções vetoriais, o estado foi organizado nos registradores vetoriais objetivando otimizar o processamento e minimizar as permutações durante a execução do algoritmo. Desse modo, foram usados nove registradores vetoriais para organizar os dados, como pode ser visto na Figura 2.

R0	α_0	α_9	α_{18}	α_{27}	α_{36}	α_{45}	α_{54}	α_{63}	α_{72}	α_{81}	α_{90}	α_{99}	α_{108}	α_{117}	α_{126}	0
R1	α_1	α_{10}	α_{19}	α_{28}	α_{37}	α_{46}	α_{55}	α_{64}	α_{73}	α_{82}	α_{91}	α_{100}	α_{109}	α_{118}	α_{127}	0
R2	α_2	α_{11}	α_{20}	α_{29}	α_{38}	α_{47}	α_{56}	α_{65}	α_{74}	α_{83}	α_{92}	α_{101}	α_{110}	α_{119}	α_{128}	0
R3	α_3	α_{12}	α_{21}	α_{30}	α_{39}	α_{48}	α_{57}	α_{66}	α_{75}	α_{84}	α_{93}	α_{102}	α_{111}	α_{120}	0	0
R4	α_4	α13	α_{22}	α_{31}	α_{40}	α_{49}	α_{58}	α_{67}	α_{76}	α_{85}	α_{94}	α_{103}	α_{112}	α_{121}	0	0
R5	α_5	α_{14}	α_{23}	α_{32}	α_{41}	α_{50}	α_{59}	α_{68}	α_{77}	α_{86}	α_{95}	α_{104}	α_{113}	α_{122}	0	0
R6	α_6	α_{15}	α_{24}	α_{33}	α_{42}	α_{51}	α_{60}	α_{69}	α_{78}	α_{87}	α_{96}	α_{105}	α_{114}	α_{123}	0	0
R7	α_7	α_{16}	α_{25}	α_{34}	α_{43}	α_{52}	α_{61}	α_{70}	α_{79}	α_{88}	α_{97}	α_{106}	α_{115}	α_{124}	0	0
R8	α_8	α_{17}	α_{26}	α_{35}	α_{44}	α_{53}	α_{62}	α_{71}	α_{80}	α_{89}	α_{98}	α_{107}	α_{116}	α_{125}	0	0

Figura 2. Organização do estado principal usando registradores de 128 bits

Como foi descrito na Seção 2, as duas funções mais críticas na implementação do LowMC são: (i) multiplicação do estado por uma matriz binária constante $N \times N$; (ii) caixa-S, que consiste na aplicação da Função 1 sobre conjuntos de três bits. A seguir, vamos detalhar como essas funções foram implementadas de forma eficiente utilizando instruções vetoriais.

4.1.1. Multiplicação do Estado por Matriz

A função de multiplicação de matrizes recebe como entrada o estado organizado em registradores vetoriais, como apresentado na Figura 2, e retorna um novo estado com a mesma

organização. Como as matrizes utilizadas são constantes, foi possível pré processá-las para facilitar a manipulação dos dados pelas instruções vetoriais.

A primeira transformação na matriz foi estendê-la de uma matriz de $n \times n$ bits para uma matriz de $n \times n$ bytes, visto que são processadas oito mensagens por vez. O segundo passo foi organizar as linhas das matrizes tal qual foi organizado o estado principal nos registradores vetoriais; vale destacar que foram inseridos alguns zeros em posições estratégicas das matrizes para melhorar o alinhamento dos dados junto à arquitetura alvo.

Dessa forma, a leitura de uma linha completa da matriz implica na leitura de nove blocos de 128 bits, organizados como o estado em registradores vetoriais. Após a leitura é feita uma multiplicação (AND) de cada palavra a_i do estado com cada palavra a_i da linha da matriz correspondente. Por fim, deve-se calcular a soma (XOR) desses resultados; como o conjunto AVX2 não possui uma instrução que aplica um XOR horizontalmente, é necessário fazer um processamento adicional para concluir essa operação. O trecho de código abaixo ilustra a leitura, multiplicação e soma de uma linha i da matriz; vale destacar que para obter o valor a_i resultante dessa operação, faz-se necessário somar todos os elementos do registrador aux[i].

```
aux[i] =  _mm_setzero_si128();
for(j=0;j<9;j++){
    temp = _mm_load_si128((__m128i*)matrix+(i*9+j));
    temp = _mm_and_si128(temp,r[j]);
    aux[i] = _mm_xor_si128(aux[i],temp);
```

A estratégia utilizada para somar os elementos de aux[i] foi processar 16 linhas da matriz por vez e organizar os dados verticalmente de modo que ao fim da soma o estado voltasse a sua organização inicial. Para facilitar o retorno à organização inicial do estado, as linhas da matriz são processadas na ordem do registrador que está sendo calculado. A seguir é ilustrado o processamento para organizar e somar as palavras restantes.

```
for(k=0;k<16;k+=2){
    aux2[k] = _mm_unpacklo_epi8(aux[k],aux[k+1]);
    aux2[k+1] = _mm_unpackhi_epi8(aux[k],aux[k+1]);}

for(k=0,j=1;k<16;k+=4,j+=4){
    aux[k] = _mm_unpacklo_epi16(aux2[k],aux2[k+2]);
    aux[j]= _mm_unpacklo_epi16(aux2[j],aux2[j+2]);
    aux[k+2] = _mm_unpackhi_epi16(aux2[k],aux2[k+2]);
    aux[j+2] = _mm_unpackhi_epi16(aux2[j],aux2[j+2]);}

for(k=0,j=8;k<4;k++,j++){
    aux2[k] = _mm_unpacklo_epi32(aux[k],aux[k+4]);
    aux2[j] = _mm_unpacklo_epi32(aux[j],aux[j+4]);
    aux2[k+4] = _mm_unpackhi_epi32(aux[k],aux[k+4]);
    aux2[j+4] = _mm_unpackhi_epi32(aux[j],aux[j+4]);}

for(k=0;k<8;k++){
    aux[k] = _mm_unpacklo_epi64(aux2[k],aux2[k+8]);
    aux[k+8] = _mm_unpackhi_epi64(aux2[k],aux2[k+8]);
}
```

```
    output[c] = _mm_setzero_si128();
    for(j=0;j<16;j++)
        output[c] = _mm_xor_si128(output[c],aux[j]);
```

4.1.2. Caixas de Substituição

Nesta etapa, como descrito na Seção 2, cada bloco de três bits é atualizado a partir de operações *booleanas* entre os mesmos; a instância do LowMC implementada neste trabalho utiliza caixas-S completas, isto é, todos os bits do estado são processados nessa etapa. A organização do estado descrita na Figura 2 foi pensada para permitir a execução eficiente dessa etapa, como pode ser visto no trecho de código a seguir:

```
void substitution(__m128i *r) {
    __m128i r1,r2,a,b,c;

    for(int i=0;i<3;i++){
        c  = r[i*3+0];
        b  = r[i*3+1];
        a  = r[i*3+2];
        r1 = _mm_xor_si128(a,_mm_and_si128(b,c));
        r2 = _mm_xor_si128(_mm_xor_si128(a,b), \
             _mm_and_si128(a,c));

        r[i*3+0] = _mm_xor_si128(_mm_xor_si128(a,b), \
                   _mm_xor_si128(c, _mm_and_si128(a,b)));
        r[i*3+1] = r2;
        r[i*3+2] = r1;}
}
```

4.2. Implementação 16-way 129 bits

O conjunto de instruções AVX2 possui um alto custo de mover dados entre a parte baixa e alta dos registradores, mas ele possui uma característica muito interessante, que é permitir executar duas operações pelo preço de uma; isto é, dado uma implementação que faz um processamento $h(x)$ eficientemente usando registradores de 128 bits é possível reescrevê-la usando registradores de 256 bits de modo a produzir $h(x_1)$ e $h(x_2)$ pelo "mesmo preço" de calcular $h(x)$.

Isso é possível porque a maioria das instruções de combinação, deslocamento e permutação que operam sobre registradores de 128 bits continuaram com o mesmo comportamento ao serem estendidas para registradores de 256 bits, a única diferença é que agora o processamento realizado sobre registradores de 128 bits foi replicado para tanto na parte baixa quanto na parte alta dos registradores de 256 bits.

Para construir a versão 16-way do LowMC foram usados nove registradores de 256 bits que foram organizados de forma similar à implementação 8-way, como pode ser visto na Figura 3. Os primeiros oito estados foram organizados na parte baixa dos registradores de 256 bits de modo análogo à organização da versão 8-way e estão representados por r_i, para $0 \leq i < 9$; os oito estados restantes foram organizados na parte alta

do registradores seguindo a mesma estrutura e estão representados na Figura 3 por R_i, para $0 \leq i < 9$.

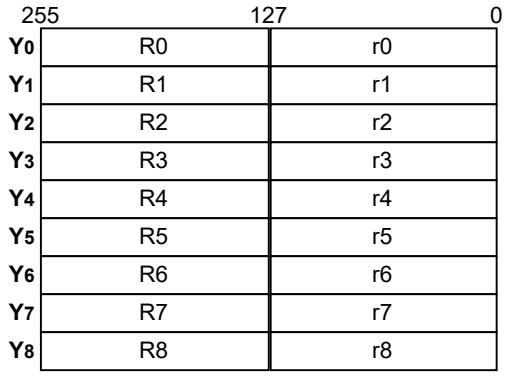


Figura 3. Organização do estado principal usando registradores de 256 bits

Uma vez organizado o estado, esta implementação se comporta de forma similar a versão 8-way. As funções de multiplicação de estado por matriz e caixas-S são implementadas de forma similar, a única diferença de implementação é a troca dos registradores e instruções de 128 bits para os equivalentes em 256 bits. A matriz também teve que ser adaptada para ficar coerente com o estado.

4.3. Implementação paralela de 192 bits

Foram implementadas duas versões do LowMC com tamanho de bloco e chave de 192 bits, quatro rodadas e caixas-S completas; a primeira versão usa registradores de 128 bits para processar paralelamente oito instâncias e a segunda versão usa registradores de 256 bits para processar 16 instâncias. As implementações usam as mesma técnicas apresentadas nas Seções 4.1 e 4.2. A principal diferença está na organização do estado. A versão 8-way usa 12 registradores de 128 bits e a versão 16-way usa 12 registradores de 256 bits, seguindo uma organização similar a apresentada nas Figuras 2 e 3.

5. Implementação Nativa

Em algumas aplicações, como na geração de chaves do Picnic, requer-se a encriptação de uma única mensagem. Adicionalmente, algumas arquiteturas não possuem conjunto de instruções vetoriais; pensando neste cenário, foi construída uma implementação sequencial otimizada da cifra LowMC usando apenas o conjunto de instruções x86-64.

O primeiro passo da implementação foi a organização do estado. Para implementar a caixa-S, cada bloco de três bits é atualizado a partir de operações lógicas entre eles. Assim, pensando em facilitar a implementação, os bits das posições múltiplas de (3) foram acomodados no primeiro registrador, os bits das posições múltiplas de (3+1) foram armazenados no segundo registrador e os bits das posições múltiplas de (3+2) foram acomodados no terceiro registrado. Para a implementação com tamanho de bloco e chave de 129 bits foram necessários apenas 43 bits dos 64 disponíveis em cada registrador; os bits restantes são inicializados com valor zero. Já a implementação com tamanho de bloco e chave de 192 bits utiliza todos os 64 bits disponíveis por registrador. A organização do estado para as seguranças de 129 e 192 bits é ilustrada na Figura 4

(a) Segurança 129 bits **(b)** Segurança 192 bits

Figura 4. Organização do estado principal com registradores nativos de 64 bits

Como nas implementações vetorizadas, as matrizes também foram pré-processadas a fim de melhorar o desempenho da multiplicação do estado pela matriz. Dessa forma, cada linha da matriz ficou com a mesma organização apresentada na Figura 4.

Dada essa organização, é feita a multiplicação de cada bit do estado com cada bit da linha i da matriz; após a multiplicação é calculada a paridade através da soma dos resultados intermediários (acumulados na variável `temp`) e da aplicação da instrução `popcnt64`, que retorna a quantidade de números uns em um registrador. De posse dessa informação, a paridade será zero se tivermos uma quantidade par de números uns e um caso contrário. A multiplicação do estado por uma linha da matriz pode ser feita como segue:

```
temp  = (matrix[i][0] & r0);
temp ^= (matrix[i][1] & r1);
temp ^= (matrix[i][2] & r2);
temp  = (_popcnt64(temp)%2);
```

A implementação da caixa-S se beneficia da organização do estado e pode ser facilmente calculada com o trecho de código a seguir:

```
c = r0; b = r1; a = r2;
r2 = a ^ (b&c);
r1 = (a^b) ^ (a&c);
r0 = (a^b) ^ ((a&b) ^ c);
```

6. Resultados

Nesta seção são apresentados os resultados experimentais das implementações vetorizadas do algoritmo LowMC usando o conjunto de instruções AVX2. Para realização dos testes de desempenho foi utilizado um computador com sistema operacional Ubuntu 21.04, 256GB de SSD e processador i7-1165G7 @ 2.8GHz. Durante os experimentos as tecnologias avançadas do processador (*speed step*, *C-state control*, *turbo-boost* e *Hyper-threading*) foram desabilitadas visando a reprodutibilidade dos experimentos. Todas as implementações que compõem este trabalho foram escritas em linguagem C e foram compiladas usando o compilador GCC 10.3.0 com a flag de compilação -O3.

Nas Tabelas 1 e 2 é possível ver o tempo de encriptação (em ciclos por bytes) por mensagem usando as implementações descritas nas Seções 4.1 e 4.2 comparadas com a implementação presente na versão otimizada do Picnic (Ramacher et al. 2022), vale destacar que a referida implementação também faz uso de instruções vetoriais. O sufixo

Tabela 1. Análise de Desempenho LowMC-129-129-4

		Tempo por Encriptação	Aceleração
8-way	**(Este Trabalho)**	1342	2,3
16-way	**(Este Trabalho)**	953	3,3
1-way	(Ramacher et al. 2022)	3116	1,0

aaa-bbb-c corresponde ao tamanho de bloco, tamanho da chave e número de rodadas, respectivamente. O tempo por encriptação foi calculado pela divisão do tempo de processamento da implementação k-way por k, isto é, o tempo gasto por uma encriptação. A aceleração foi calculada pela divisão do tempo da implementação 1-way pelo tempo por encriptação das implementações paralelas e indica a melhoria no desempenho por encriptação das implementações paralelas sobre a implementação sequencial.

Como pode-se observar na Tabela 1, o tempo por encriptação na versão de 129 bits que computa paralelamente oito mensagens é 2,3$\times$ maior do que a versão sequencial e essa aceleração aumenta significativamente quando dobra-se o número de mensagens processadas por vez. Esse ganho só foi possível por termos aproveitado a característica das matrizes serem constantes para organizá-las de modo a tornar o uso dos registradores vetoriais competitivo. Ao aumentar o nível de segurança de 129 para 192 bits, como pode ser visto na Tabela 2, a aceleração do tempo por encriptação chega a 1,8$\times$ e 2,7$\times$ nas versões paralelas que processam oito e dezesseis mensagens, respectivamente.

Tabela 2. Análise de Desempenho LowMC-192-192-4

		Tempo por Encriptação	Aceleração
8-way	**(Este Trabalho)**	2424	1,8
16-way	**(Este Trabalho)**	1642	2,7
1-way	(Ramacher et al. 2022)	4310	1,0

Como são processadas oito e dezesseis mensagens por vez, esperava-se uma maior aceleração. Analisando os resultados, foi observado que a aceleração não foi maior por causa do alto custo de acesso à memória na leitura da matriz. Embora não exista custo para organizar a matriz, uma vez que a matriz é constante e esse pré-processamento é feito antes da execução, a multiplicação do estado pela matriz exige muitos acessos à memória. O processamento de cada linha da matriz, para segurança de 129 bits, requer a leitura de 144 e 288 bytes nas versões 8-way e 16-way, respectivamente. Para a segurança de 192 bits, a leitura de uma linha da matriz nas versões 8-way e 16-way requer a leitura de 192 e 384 bytes, respectivamente.

Tabela 3. Análise de Desempenho da Implementação Nativa do LowMC

	Implementação x86-64	Tempo por Encriptação
LowMC-129-129-4	(**Este Trabalho**)	4333
	(Ramacher et al. 2022)	5800
LowMC-192-192-4	(**Este Trabalho**)	6359
	(Ramacher et al. 2022)	8063

Como pode ser visto na Tabela 3, a implementação nativa descrita neste trabalho é em torno de 25% mais rápida que a implementação presente na versão otimizada do Picnic (Ramacher et al. 2022) para segurança de 129 bits, quando compilada para arquiteturas que não possuem instruções vetoriais. O ganho da versão com segurança de 192 bits ficou na casa dos 22%.

7. Conclusão

Este trabalho apresentou diferentes técnicas de implementação para acelerar o desempenho da cifra simétrica LowMC. Foi descrita uma implementação nativa 25% mais rápida que a versão disponível na versão otimizada do Picnic, quando compiladas para arquiteturas que não possuem instruções vetoriais; essa implementação pode ser diretamente usada no processo de geração de chaves do Picnic ou em qualquer aplicação que use esta instância da cifra LowMC. Também foram apresentadas duas versões paralelas que processam oito e dezesseis mensagens simultaneamente com aceleração de 2,3 e 3,3 vezes, respectivamente, quando comparadas com a versão mais otimizada do LowMC com caixas-S completas e segurança de 129 bits.

Como trabalho futuro, pretende-se usar as técnicas aqui apresentadas para acelerar o desempenho de assinatura e verificação do algoritmo de assinatura digital pós-quântico Picnic, o qual processa inúmeras encriptações independentes da cifra LowMC. Adicionalmente, pretende-se usar o conjunto de instruções AVX-512 para melhorar ainda mais o tempo de execução do LowMC; tal como explorar técnicas de implementação para acelerar todas as instâncias do LowMC.

Referências

Alagic, G., Alperin-Sheriff, J., Apon, D., Cooper, D., Dang, Q., Kelsey, J., Liu, Y.-K., Miller, C., Moody, D., Peralta, R., et al. (2020). Status report on the second round of the nist post-quantum cryptography standardization process. *US Department of Commerce, NIST*.

Albrecht, M., Grassi, L., Rechberger, C., Roy, A., and Tiessen, T. (2016). Mimc: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 191–219. Springer.

Albrecht, M. R., Rechberger, C., Schneider, T., Tiessen, T., and Zohner, M. (2015). Ciphers for mpc and fhe. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 430–454. Springer.

Aly, A., Ashur, T., Ben-Sasson, E., Dhooghe, S., and Szepieniec, A. (2020). Design of symmetric-key primitives for advanced cryptographic protocols. *IACR Transactions on Symmetric Cryptology*, pages 1–45.

Biham, E. (1997). A fast new des implementation in software. In *International Workshop on Fast Software Encryption*, pages 260–272. Springer.

Cabral, R. and López, J. (2018). Implementation of the sha-3 family using avx512 instructions. In *Anais do XVIII Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais*, pages 25–32. SBC.

Chase, M., Derler, D., Goldfeder, S., Orlandi, C., Ramacher, S., Rechberger, C., Slamanig, D., and Zaverucha, G. (2017). Post-quantum zero-knowledge and signatures from symmetric-key primitives. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 1825–1842, New York, NY, USA. Association for Computing Machinery.

Corporation, I. (2011). Intel® Advanced Vector Extensions Programming Reference. Disponível em ⟨https://software.intel.com/sites/default/files/m/f/7/c/36945⟩.

Dobraunig, C., Kales, D., Rechberger, C., Schofnegger, M., and Zaverucha, G. (2021). Shorter signatures based on tailor-made minimalist symmetric-key crypto. *Cryptology ePrint Archive*.

Faz-Hernández, A., Cabral, R., Aranha, D. F., and López, J. (2015). Implementação Eficiente e Segura de Algoritmos Criptográficos. In *Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais - Minicursos*, volume XV, pages 93–140. Sociedade Brasileira de Computação.

Flynn, M. (1972). Some Computer Organizations and Their Effectiveness. *Computers, IEEE Transactions on*, C-21(9):948–960.

Greg, Z. et al. (2020). The picnic signature algorithm specification version 3.0.

Guide, P. (2011). Intel® 64 and ia-32 architectures software developer's manual. *Volume 3B: System programming Guide, Part*, 2(11).

Kales, D., Perrin, L., Promitzer, A., Ramacher, S., and Rechberger, C. (2017). Improvements to the linear operations of lowmc: A faster picnic. *Cryptology ePrint Archive*.

Kales, D. and Zaverucha, G. (2020). Improving the performance of the picnic signature scheme. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 154–188.

Käsper, E. and Schwabe, P. (2009). Faster and timing-attack resistant aes-gcm. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 1–17. Springer.

Matsui, M. and Nakajima, J. (2007). On the power of bitslice implementation on intel core2 processor. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 121–134. Springer.

Orisaka, G., Aranha, D. F., and López, J. (2018). Finite field arithmetic using avx-512 for isogeny-based cryptography. In *Anais do XVIII Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais*, pages 49–56. SBC.

Ramacher, S., Zaverucha, G., and Kales, D. (2022). Optimized implementation of the picnic signature scheme. Repositório: ⟨https://github.com/IAIK/Picnic⟩. (Jul,2022).

Rebeiro, C., Selvakumar, D., and Devi, A. (2006). Bitslice implementation of aes. In *International Conference on Cryptology and Network Security*, pages 203–212. Springer.

Shannon, C. E. (1945). A mathematical theory of cryptography. *Mathematical Theory of Cryptography*.