UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS DE QUIXADÁ

CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

SAMUEL EVANGELISTA DE AQUINO JÚNIOR

ESTUDO E OTIMIZAÇÃO DO LOWMC

QUIXADÁ

2022

SAMUEL EVANGELISTA DE AQUINO JÚNIOR

ESTUDO E OTIMIZAÇÃO DO LOWMC

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação do Campus de Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Ciência da Computação.

Orientador: Prof. Me. Roberto Cabral Rabêlo Filho

QUIXADÁ

2022

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

A669e Aquino Junior, Samuel Evangelista de.
Estudo e otimização do LowMC / Samuel Evangelista de Aquino Junior. – 2022.
49 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Quixadá, Curso de Ciência da Computação, Quixadá, 2022.
Orientação: Prof. Me. Roberto Cabral Rabêlo Filho.

1. Otimização. 2. Criptografia. 3. Prova de Conhecimento Zero. 4. Privacidade. I. Título.
CDD 004

SAMUEL EVANGELISTA DE AQUINO JÚNIOR

ESTUDO E OTIMIZAÇÃO DO LOWMC

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação do Campus de Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Ciência da Computação.

Aprovada em: ___/___/___

BANCA EXAMINADORA

Prof. Me. Roberto Cabral Rabêlo Filho (Orientador)
Universidade Federal do Ceará (UFC)

Profª. Drª. Amanda Cristina Davi Resende
Universidade Federal de Catalão (UFCAT)

Prof. Dr. Wladimir Araújo Tavares
Universidade Federal do Ceará (UFC)

Para todas as pessoas que de forma direta ou indireta influenciaram e incentivaram essa trajetória.

AGRADECIMENTOS

Gostaria de agradecer primeiramente a Deus por ter me dado forças e guiado meus sonhos e caminhos até aqui.

A minha família que sempre me apoiou em toda a minha jornada, em especial a minha mãe por sempre me dar forças e me apoiar incondicionalmente.

Ao professor e orientador M.e Roberto Cabral Rabêlo Filho, por todos incentivos, conselhos e ensinamentos.

À professora Drª. Amanda Cristina Davi Resende e ao professor Dr. Wladimir Araújo Tavares pela disponibilidade e dedicação em contribuir com este trabalho.

Aos meus amigos que me acompanharam durante todo o período acadêmico, e aos outros que de forma indireta me incentivaram. Muito do que sou hoje é uma média de todos os grandes amigos que tenho.

Dentre esses gostaria de ressaltar os que participaram comigo da bolsa do VTEX Lab e os que estão comigo desde o ensino médio. Foram muitos acertos e erros, vitorias e derrotas, porém a amizade sempre se manteve firme.

Aos demais professores e servidores da UFC – Quixadá que contribuíram de forma direta ou indireta para a minha jornada pessoal e acadêmica.

Por fim, agradeço a todos aqueles que não citei porém contribuiu de alguma forma para este momento, com um incentivo, uma palavra de conforto ou qualquer outro tipo de ajuda.

Meu muito obrigado!

“Sempre fui sonhador e é isso que me mantém vivo.”

(Racionais MC’s)

RESUMO

A evolução da computação é acompanhada sempre de novos desafios e um desses são os problemas considerados intratáveis para a computação clássica, que se tornam tratáveis na computação quântica. Um desses problemas afeta diretamente uma subárea de computação, que é a criptografia; fundamental na comunicação segura de dispositivos digitais. Em 1994, Peter Shor apresentou um algoritmo quântico capaz de solucionar o problema da fatoração de inteiros e do logaritmo discreto. Tal descoberta foi um marco para determinar o inicio dos estudos na área da criptografia pós-quântica. Tais problemas, advindo da teoria dos números, são base para primitivas criptográficas assimétricas. Nesse contexto, surgiu um algoritmo de assinatura digital baseado no conceito de Prova de Conhecimento Zero e Computação Multiparte chamado Picnic, tal algoritmo é considerado seguro mesmo com o surgimento de um computador quântico de proposito geral, porém esse algoritmo ainda é relativamente lento. Dentro do Picnic existe uma cifra simétrica que é executada algumas centenas de vezes de forma independente, o LowMC. Assim, este trabalho tem como objetivo buscar otimizações nesta primitiva criptográfica especifica, buscando contribuir com a criptografia em geral. A principal técnica aplicada é a paralelização a nível de instrução, de modo a utilizar instruções vetoriais, para processar vários estados paralelamente.

Palavras-chave: Otimização. Criptografia. Prova de Conhecimento Zero. Privacidade.

ABSTRACT

The evolution of computing is always accompanied by new challenges and one of these are the problems considered intractable for classical computing that become treatable in quantum computing. One of these problems directly affects a subarea of computing, which is cryptography; fundamental in the secure communication of digital devices. In 1994, Peter Shor presented a quantum algorithm capable of solving the problem of factoring integers and the discrete logarithm. This discovery was a milestone to determine the beginning of studies in the area of post-quantum cryptography. Such problems, arising from number theory, are the basis for asymmetric cryptographic primitives. Thus, a digital signature algorithm emerged based on the concept of Proof of Knowledge and Multipart Computing called Picnic, such an algorithm is considered secure even with the emergence of a general-purpose quantum computer, but this algorithm is still relatively slow. Within Picnic there is a symmetric cipher that runs a few hundred times independently. Thus, this work aims to seek optimizations in this specific cryptographic primitive, seeking to contribute to cryptography in general. The main technique applied is instruction-level parallelization, to use instruction vectors, to process on amounts of bits in parallel.

Palavras-chave: Optimization. Cryptography. Zero Knowledge Proof. Privacity.

LISTA DE ILUSTRAÇÕES

SUMÁRIO

1 INTRODUÇÃO

Com a constante evolução da computação, novos conceitos e tecnologias se tornam cada vez mais reais em nosso cotidiano. Um desses novos conceitos é a Computação Quântica — ciência que estuda as teorias e as propriedades da mecânica quântica com intuito de aplicá-las na Ciência da Computação — que busca desenvolver hardwares, algoritmos e softwares com base em conceitos da física quântica (NICOLIELLO, 2009).

Nesse cenário, as tecnologias precisam reinventar-se para acompanhar tal avanço. A computação quântica se mostrou bastante eficiente na resolução de alguns problemas tidos como intratáveis na computação clássica (VIGNATTI *et al.*, 2004), tais como, a fatoração de números inteiros e a resolução de logaritmos discretos. Por conseguinte, algumas grandes áreas da tecnologia precisaram acompanhar tal avanço, visto que alguns conceitos da computação clássica tornaram-se obsoletos, é o caso da criptografia que está evoluindo as primitivas para serem resistente à computadores quânticos.

A criptografia, é a arte/ciência/engenharia que estuda técnicas para fornecimento de serviços de segurança, como confidencialidade, autenticação de origem, anonimato, integridade e irretratabilidade, essencialmente em sistemas computacionais. Alguns algoritmos criptográficos baseiam-se em problemas da teoria dos números, como os algoritmos de chave pública. RSA *(Rivest-Shamir-Adleman)* (RIVEST *et al.*, 1978) e CCE *(Criptografia de Curvas Elípticas)* (MILLER, 1985), são amplamente utilizados para construção de protocolos de troca de chaves e assinatura digital. A segurança desses algoritmos baseiam-se na dificuldade de fatorar números inteiros e de resolver logaritmo discreto; problemas esses tidos como intratáveis (para números inteiros grandes) na computação clássica (SCHNEIER, 1996).

Atualmente, os problemas citados anteriormente são denominados NP (Classe de problemas não resolvíveis em tempo polinomial determinístico). Contudo, com a computação quântica é possível mover esses problemas NP para a generalização quântica de P (Classe de problemas resolvíveis em tempo polinomial determinístico) (OLIVEIRA *et al.*, 2010), chamada BQP (Classe de problemas resolvíveis em tempo polinomial quântico de erro limitado), que é uma classe de problemas solúveis por um computador quântico em tempo polinomial, com uma taxa de erro de quase $\frac{1}{4}$ (AARONSON, 2010).

Tal descoberta sobre a generalização quântica BQP, em 1994, tornou-se um marco importante para a criptografia, quando o matemático Peter Shor demonstrou que o problema da fatoração de inteiros e logaritmos discretos são BQP (SHOR, 1994). Deste modo, deu-se o

estopim para os estudos na área de criptografia pós-quântica; que são algoritmos criptográficos seguros mesmo com o advento de um computador quântico de propósito geral.

Em 2016 um novo um algoritmo de assinatura digital resistente à computadores quânticos surgiu, o Picnic (CHASE *et al.*, 2017). Esse algoritmo utiliza conceitos de Computação Multiparte (Multi-party Computation - MPC) e Prova de Conhecimento Zero (Zero-Knowledge Proof - ZKP) para construir assinaturas digitais resistentes à computadores quânticos. No conceito de MPC, as operações lineares como a `XOR` (OU exclusivo) são quase "de graça", no entanto operações não lineares como a `NAND` (não E) demandam muito processamento.

O Picnic utiliza de um protocolo denominado Computação Multiparte na Cabeça (*MPC-in-the-head*) para a ZKP. Esse protocolo é baseado em MPC, que se beneficia de cifras com poucas operações não lineares; estando o tamanho da assinatura diretamente relacionado com a quantidade de operações não lineares. Contudo, é necessário haver um equilíbrio entre a quantidade de operações não lineares e operações lineares. A cifra utilizada dentro do Picnic o LowMC (ALBRECHT *et al.*, 2016), foi projetada para minimizar a quantidade de operações não lineares garantindo a segurança e eficiência da cifra.

Nessa aplicação específica, o LowMC é executado algumas centenas de vezes dentro do Picnic, e essas execuções são independentes entre si, fazendo com que essa técnica seja bastante dispendiosa computacionalmente, por conta de todas as operações executadas no LowMC. Sendo assim, é extremamente relevante encontrar otimizações para o LowMC, uma vez que tais otimizações implicam diretamente no ganho de desempenho do Picnic. Dado os avanços na computação quântica, é necessário estar preparado para tempos em que computadores quânticos terão uma quantidade suficiente de **qubits** (bit quântico), ao ponto de, quebrar a segurança clássica.

Desta forma, no trabalho presente foi implementado a paralelização da cifra LowMC a nível de instrução, ou seja, como o LowMC executa as operações bit a bit existe um desperdício de bits que poderiam ser utilizados dado a arquitetura de computadores que temos hoje. Então foi utilizado instruções vetoriais Intel Intrisincs para paralelizar a nível de instruções as operações utilizadas nas funções do LowMC; assim, mais bits são operados simultaneamente. A partir disso foi alcançado um resultado satisfatório de desempenho nas analises feitas, como também foram propostas técnicas de otimizações.

1.1 Objetivo Geral

Portanto, o principal objetivo deste trabalho é estudar e implementar otimizações na cifra LowMC e analisar o desempenho dessas otimizações.

1.1.1 Objetivos Específicos

a) Implementar versões paralelas, com encriptações de q textos por chamada, do LowMC;

b) Buscar otimizações para a camada afim;

c) Análise de desempenho e benchmark;

O restante deste trabalho está organizado no seguinte formato. No Capítulo 2, são apresentados os conceitos essenciais para o desenvolvimento deste trabalho, que são a assinatura digital Picnic e suas demais especificidades, a cifra LowMC e as instruções vetoriais; no Capítulo 3, realiza-se a apresentação e discussão dos trabalhos relacionados, tais como, suas conformidades e inconformidades em relação ao trabalho aqui proposto; no Capítulo 4, a metodologia a ser empregada para o desenvolvimento deste trabalho é explicada, as otimizações encontradas como também a forma de paralelização da cifra LowMC; no Capítulo 5 é abordado a implementação da paralelização de forma detalhada, as análises realizadas, como também as otimizações encontradas para otimizar alguns pontos de gargalo da cifra; e, por fim, o Capítulo 6 conclue este trabalho, relatando as contribuições realizadas e os trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Nesta seção, serão apresentados alguns conceitos de suma importância para a compreensão e desenvolvimento da solução proposta neste trabalho. Na Seção 2.5, é explicado todo o conteúdo teórico necessário para embasar o conhecimento sobre a assinatura digital Picnic. Na Seção 2.4, é explicado o algoritmo no qual este trabalho é baseado, o LowMC, de forma detalhada. Na Seção 2.6, é apresentado o conceito de instruções vetoriais, no qual é utilizado para as otimizações proposta neste trabalho.

2.1 Prova de Conhecimento Zero

ZKP é um método de prova em que duas partes, o provador P e o verificador V, tentam confirmar um dado secreto S, de forma a não revelar tal dado secreto. Então P precisa provar para V, que ele conhece o dado secreto S de outras maneiras, sem revelar nada do processo, somente que ele tem o conhecimento do dado S.

Uma maneira simples de entender ZKP é pensar no seguinte cenário: "Uma casa que somente pode ser aberta com um chave secreta e nada mais. Um provador P quer provar para um verificador V que ele conhece essa chave secreta da casa sem revelar a chave ou qualquer outra informação, com intuito de evitar falha de integridade. Então, se o V confirmou que existe um determinado objeto na casa, P pode trazer o objeto e consequentemente provar que ele é tem o conhecimento da chave. Esse processo é a ZKP, onde o 'conhecimento' é a chave (HUQING; ZHIXIN, 2013)."

2.2 Computação Multiparte

Computação Multiparte é uma técnica para o processamento de dados distribuídos por várias partes, de modo que esses dados sejam mantidos privados durante todo o processamento. O objetivo desta técnica é computar um valor y, onde y é o resultado de uma função com o conteúdo de cada parte, ou seja, $y = f(x_1, ..., x_n)$. Cada parte x_i processada pela função f vem das i informações a serem processadas pela Computação Multiparte. É necessário também satisfazer duas condições principais durante esse processamento, que são: i) exatidão, no qual o valor correto de y é computado, ii) privacidade, onde y é a única informação nova que será divulgada (CRAMER *et al.*, 2015).

Dentro do Picnic a computação Multiparte é utilizado como um protocolo, computa-

ção Multiparte na Cabeça, citado na subseção 2.3, que serve para construir sistemas de ZKP. Tal protocolo utiliza o LowMC como sua cifra de encriptação, por conta da característica dessa cifra minimizar a quantidade de operações não lineares (GELLERSEN *et al.*, 2020).

2.3 Computação Multiparte na Cabeça

Computação Multiparte na Cabeça é um protocolo utilizado para obter uma ZKP a partir da técnica de processamento MPC (SIDORENCO *et al.*, 2021). Para isso, o protocolo processa com o MPC o dado secreto localmente e disponibiliza após a computação as partes solicitadas pelo verificador para confirmar a consistência. O verificador por sua vez solicita $n-1$ partes para validar sua consistência, caso seja coerente, ele aceita a ZKP e rejeita caso contrário. Tal verificação ocorre m vezes, visto que, toda vez que ocorre uma interação de verificar a consistência da prova, a probabilidade de dados inconsistentes diminuí em um fator de $\left(\frac{1}{2}\right)^m$. O verificador pode testar quantas vezes for necessário para confirmar a consistência da prova.

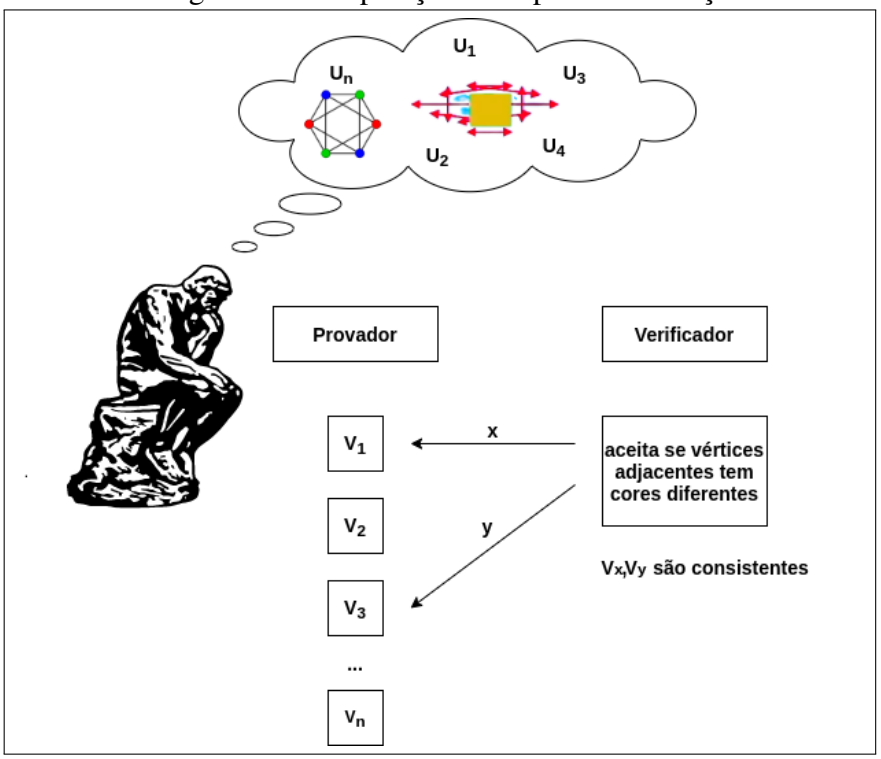
Na Figura 1, temos o exemplo de uma iteração de como funciona o protocolo Computação Multiparte na Cabeça, onde a informação a ser processado pelo MPC é um grafo 3-colorável. Após execução do MPC, são disponibilizadas as visualizações, que são o resultado das partes processadas pelo MPC, dispostas pelo Provador podendo ser verificadas. Logo, o verificador pode solicitar as visualizações que são o resultado do MPC de um grafo 3-colorável, ou seja, caso solicitado duas partes é aceito se os nós tiverem cores diferentes e rejeitado caso contrário. Desta forma, é possível provar uma informação sem revelá-la por completa, mesmo que a probabilidade nunca chegue aos 100%, tenderá à isso.

2.4 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 introduzido por Shannon para construção de boas cifras simétricas (ALBRECHT *et al.*, 2016). A cifra LowMC foi construída com objetivo de diminuir a complexidade multiplicativa que implica na diminuição do tempo de execução de algumas técnicas, tais como MPC e ZKP. Essa cifra foi projetada para otimizar a quantidade e a profundidade multiplicativa, logo, as operações não lineares são minimizadas ao máximo para a maior performance da cifra no contexto em que é proposta.

A cifra LowMC é uma cifra altamente parametrizável, sendo composta pelo seguintes

Figura 1 – Computação Multiparte na Cabeça



Fonte: Autor deste trabalho baseado em Yuval Ishai

parâmetros. Um bloco de tamanho n, uma quantidade m de *S-boxes* — componente básico que realiza substituição para ocasionar confusão na cifra (NYBERG, 1991), técnica criptográfica utilizada para aumentar a imprecisão da cifra —, uma chave de tamanho k, e um número de rodadas r.

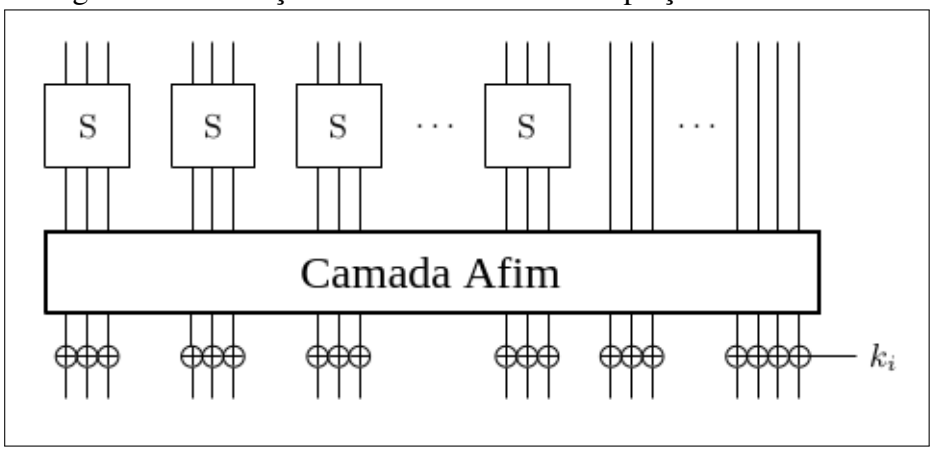
Para reduzir a complexidade multiplicativa, a seguinte estratégia é empregada: aplicar o número de *S-boxes* em paralelo reduz a complexidade multiplicativa, tornando uma parte da camada de substituição como o mapeamento de identidade, assim como na Figura 2. Da mesma forma, um gerador pseudo aleatório de matrizes binárias é utilizado para agregar um alto grau de difusão da cifra, técnica criptográfica aplicada para ampliar a redundância da cifra com intuito de obscurecer a estrutura estatística da cifra, assim, minimizando tentativas de deduzir a chave, tornando-a segura apesar da baixa complexidade multiplicativa.

O processo de encriptação do LowMC consiste nos seguintes passos:

1) O primeiro passo é fazer um "clareamento" 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, e cada rodada é composta de:

Figura 2 – Descrição de uma rodada de encriptação com LowMC



Fonte: (ALBRECHT *et al.*, 2016)

`Rodada do LowMC =`

`Camada de Sbox` $\circ$ `Adição de constante(i)` $\circ$ `Camada linear(i)` $\circ$ `Adição de chave(i)`[1]

a) Camada de *Sbox*: é uma camada de $3m$ aplicações paralelas de *Sbox*, onde m é a quantidade de *Sbox*. Logo, a camada de *Sbox* é composta pelos $3m$ bits iniciais do estado. Os $n-3m$ bits restantes do estado são a identidade. Abaixo é descrito a especificação da implementação da *Sbox*. Onde a, b e c são bits da cifra:

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

b) Adição de constante: é adicionada uma constante de rodada ao estado, o vetor de constante é gerada previamente e é único para cada instância de bits do LowMC. A constante é escolhida independentemente e uniformemente aleatória de um vetor binário de tamanho n gerado previamente ao processamento do LowMC;

c) Camada linear: é a multiplicação do estado por uma matriz binária $n \times n$, que é gerada previamente e é única para cada instância de bits do LowMC. A matriz é escolhida independentemente e uniformemente aleatória de uma matriz $n \times n$ inversível. A função que executa esse passo é $MultiplyGF2Matrix(LMatrix(i), estado)$;

d) Adição de chave: a chave de rodada é adicionada ao estado (por meio de uma XOR). Para gerar as chaves de rodada a chave mestre é multiplicada por uma matriz binaria $n \times k$. A matriz é escolhida independentemente e uniformemente aleatória de uma matriz $n \times k$ gerada previamente ao processamento do LowMC.

A seguir, será mostrado o pseudocódigo do Algoritmo 1 de encriptação LowMC.

[1] A notação $\circ$ está sendo utilizada para individualizar a execução de cada função dentro da rodada do LowMC

Mas antes disso precisamos definir algumas variáveis, o texto claro e o estado são variáveis de n de bits. A chave possui k bits, que pode ser menor que o valor do n, e r é a quantidade de rodadas que serão executadas durante a encriptação.

Algoritmo 1 LowMC

```
1: função LowMC(texto_claro, chave)
2:     estado = texto_claro + MultiplyGF2Matrix(KMatrix(0), chave)
3:     para todo i = 0 to r faça
4:         // Camada de Substituição
5:         estado = Sboxlayer(estado)
6:         // Camada Afim
7:         estado = MultiplyGF2Matrix(LMatrix(i), estado)
8:         estado = estado + Constants(i)
9:         // Gerar chave de rodada e adicionar no estado
10:        estado = estado + MultiplyGF2Matrix(KMatrix(i), estado)
11:    fim para todo
12:    devolve estado
13: fim função
```

Fonte: Albrecht *et al.* (2016)

2.5 Picnic

Picnic é um algoritmo de assinatura digital resistente à computadores quânticos (CHASE *et al.*, 2017). Atualmente, é desenvolvido em colaboração com pesquisadores de várias universidades e pesquisadores da Microsoft. O Picnic, diferente de outros algoritmos de chave pública, não é baseado em problemas da teoria dos números. De outro modo, é baseado nos conceitos ZKP, MPC, e primitivas simétricas como funções de resumo e cifras de bloco (SONI *et al.*, 2021).

No Picnic, a chave pública é um par (C, p), onde $C = E(sk, p)$ e p é o texto claro gerado de forma aleatória. O parâmetro E é uma cifra de bloco para a encriptação do texto claro p utilizando a chave privada sk. A cifra de bloco E utilizada no Picnic é o LowMC. O processo para a criação de uma assinatura é composto pelos passos de geração de chave, geração da assinatura e validação da assinatura.

– O primeiro passo do algoritmo é a geração das chaves pública e privada. A chave privada é gerada a partir de um gerador de números aleatórios. Após a geração da chave privada é gerada a chave pública que é o par (C, p). Como retorno, temos o par de chaves pública e

privada (sk, pk);

– Logo após a geração da chave, temos o processo de gerar a assinatura, que depende da mensagem m, do seu tamanho e da chave sk que foi gerada no primeiro passo. Para gerar a assinatura é utilizado o protocolo Computação Multiparte na Cabeça para criar uma ZKP da mensagem m, e ao final temos a assinatura s e o seu respectivo tamanho;
– Por fim, no processo de verificação da assinatura é verificado se os parâmetros da chave pública estão coerentes. Após isso, o desafio é computado para a comparação com o desafio recebido, o desafio recebido são as $n-1$ partes solicitadas após execução do protocolo Computação Multiparte na Cabeça, se ambos forem iguais a assinatura é aceita, caso contrário é rejeitada.

2.6 Instruções Vetoriais

As instruções vetoriais são unidades funcionais com capacidade de processar um conjunto de dados com a execução de uma única instrução (RABÊLO *et al.*, 2015). Esse conceito também é conhecido como SIMD (*Single Instruction Multiple Data*). Existem vários tipos de instruções vetoriais, como, por exemplo, `XOR`, `AND`, `STORE`, `LOAD` e muitas outras, e todas operam sobre matrizes unidimensionais de dados. Atualmente, várias CPUs (Central Processing Unit) apresentam em sua arquitetura alguns conjuntos de instruções, por exemplo, os conjuntos de instruções *MMX* (*Multimedia eXtensions*), *SSE* (*Streaming SIMD eXtensions*), *AVX* (*Advanced Vector eXtensions*) e muitos outros.

As instruções vetoriais não é um conceito novo. No fim da década de 1990, esforços foram concentrados por fabricantes de processadores para explorar mais sobre o paralelismo (RABÊLO *et al.*, 2015). Logo os primeiros conjuntos de instruções começaram aparecer, foi o caso do conjunto lançado pela Intel em 1997, o MMX. Desde então, as instruções vetoriais evoluem sempre com intuito de otimizar desempenho de técnicas dispendiosas. Recentemente foram adicionadas instruções de 512 bits nos processadores de última geração. Tais instruções são muito utilizadas para processamento de imagens e algoritmos matemáticos, com a finalidade de paralelizar o processamento de dados. Tal técnica é denominada paralelismo a nível de instrução. Uma única instrução vetorial, equivale a um loop de execução da mesma instrução *bit a bit*.

Utilizamos essa técnica para paralelizar o algoritmo do LowMC. Como as funções operam sobre o estado, e o estado é um vetor, podemos utilizar instruções Intrinsics para executar

de forma paralela as operações dessas funções. A partir de um conjunto de estados, iremos construir um estado principal usando *Bit slicing* que é uma técnica de combinar módulos do processador de largura de bits menor, com o objetivo de aumentar o comprimento da palavra (BENADJILA *et al.*, 2013). As operações do LowMC vão utilizar instruções vetoriais para operar no estado principal. Utilizamos instruções vetoriais lógicas `XOR`, `AND`; transferência de dados `LOAD`, `STORE`; e algumas de permutação `BLEND`, `PERMUT`, `UNPACK` para organizar o estado.

3 TRABALHOS RELACIONADOS

Nesta seção, serão apresentados alguns trabalhos relacionados que fundamentaram o conteúdo teórico e prático deste trabalho.

3.1 Ciphers for MPC and FHE

Em Albrecht *et al.* (2016), é proposto o estudo de primitivas de chave simétrica com intuito de minimizar o tamanho multiplicativo e a profundidade de sua especificação. Tal trabalho foi motivado dado o recente avanço em instanciações práticas de MPC, Criptografia Totalmente Homomórfica (Fully Homomorphic Encryption - FHE) e ZKP onde as operações lineares, são basicamente "de graça", em comparação as operações não lineares. O trabalho de Albrecht *et al.* (2016) propõe uma família de cifra de bloco "LowMC", que supera, de longe, todas a cifras existentes em relação as métricas anteriormente citadas.

Neste trabalho, utilizamos de alguns artifícios de otimização a nível de software para aprimorar algumas etapas de computação do trabalho de Albrecht *et al.* (2016). A partir da cifra LowMC, aplicamos duas técnicas de otimização para conseguir aprimorar o desempenho do algoritmo. A primeira otimização realiza uma paralelização a nível de instrução com intuito de otimizar a eficiência das funções especificas do "LowMC". A segunda aplica otimizações na camada afim, que é uma das partes da cifra que demandam mais processamento.

3.2 Improvements to the Linear Operations of LowMC: A Faster Picnic

Em Kales *et al.* (2017), são estudadas várias formas de implementações otimizadas da cifra LowMC. Ao decompor a computação da chave de rodada, isso baseado nos efeitos da chave, na camada S-box e em otimizações no geral foi perceptível a redução no tempo de execução em um fator de dois, além de diminuir o tamanho das matrizes em aproximadamente 45% comparado a implementação original.

Outras duas modificações apresentadas em Kales *et al.* (2017) são a decomposição da multiplicação na camada linear em partes que dependem do efeito na S-box, exigindo com que a matriz da camada linear tenha uma submatriz inversa, porém reduz significativamente o tempo de execução em cerca de 400%. A segunda proposta utiliza uma estrutura de Fesistel para substituir a grande operação de multiplicação de matriz que resta na camada linear. Tais alterações, aumentam em 60% a eficiência de multiplicação de matrizes no Picnic.

O trabalho Kales *et al.* (2017) é o mais comparável com as otimizações que realizadas neste trabalho. Estudamos e implementamos técnicas que otimizaram o processamento do LowMC, de forma a utilizar instruções vetoriais para isso, implementando paralelização a nível de instrução. Acreditamos que a comparação entre este trabalho e o trabalho (KALES *et al.*, 2017) trará grandes resultados para a implementação do LowMC.

4 METODOLOGIA

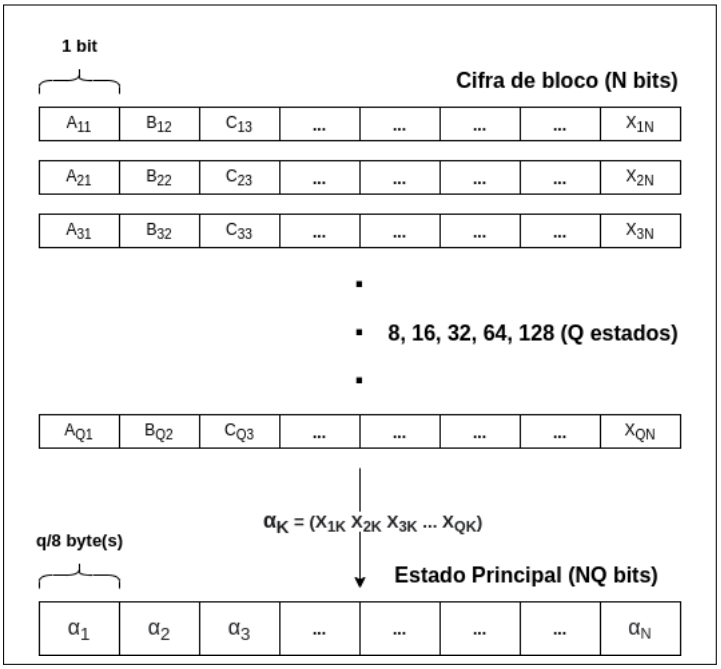
Nesta seção, relatamos um pouco sobre como está disposto os métodos científicos que são implementados e a sua respectiva ordem de execução.

4.1 Implementação da paralelização a nível de instrução para q textos

É utilizado o conceito SIMD, que é um método para que as operações de computadores sejam executadas simultaneamente, ou seja, utilizar instruções vetoriais para paralelizar as operações que são executadas nas funções a nível de instrução. O objetivo principal dessa proposta é fazer com que a encriptação de vários textos claros tenham uma melhor desempenho a partir da utilização de instruções vetoriais para o processamento paralelo.

A primeira parte do processo para realizar tal implementação é criar um estado principal, que conterá a entrada de Q textos de tamanho N, que serão processados paralelamente. O objetivo é organizar o estado principal de forma que cada palavra contenha Q bits e seja a junção de todos Q_n bits dos outros estados. Na Figura 3 é demonstrado como deverá ocorrer a formação do estado principal.

Figura 3 – Formação do estado principal



Fonte: Autor deste trabalho

Após formar um estado principal, todas as operações serão feitas por um conjunto de instruções vetoriais. Logo, todas as funções de camada do LowMC, como, camada de

substituição, camada afim, e a função de adição de chave de rodada terão suas funções alteradas para que as operações como, `XOR`, `AND`, dentre outras sejam a partir de instruções vetoriais. Dado essa proposta acreditamos reduzir o tempo de encriptação de um conjunto N entradas sequenciais.

A partir do processamento paralelo ao invés do processamento sequencial, vamos ter que lidar com o gerenciamento do estado principal, contudo esperamos minimizar substancialmente o tempo de encriptação sequencial de várias mensagens. Após o processamento iremos refazer os Q textos encriptados para obtermos as mensagens especificas de cada texto.

4.2 Buscar otimizações para a camada afim

Iremos buscar por otimizações que possam ser aplicadas na camada afim, onde ocorre a multiplicação de uma matriz $n \times n$ por um vetor n. Essa camada ainda é o maior gargalo de processamento dentro do algoritmo.

4.3 Análise de desempenho e benchmark

Após todas implementações propostas por esse trabalho iremos comparar os resultados a partir de um benchmark que será realizado. Vamos validar nossas implementações com o algoritmo de referência e analisar o desempenho comparando com as versões mais otimizadas do LowMC. O atual artigo do Picnic tem a versão mais atualizada e otimizada do LowMC (KALES; ZAVERUCHA, 2020). Neste trabalho algumas alterações feitas no LowMC fora cruciais para diminuir o tempo de processamento do Picnic em 1,4 a 1,8 vezes e também o tamanho da assinatura.

5 RESULTADOS

Neste capítulo, são apresentados os resultados alcançados a partir da execução dos procedimentos metodológicos. Inicialmente, são descritos os processos metodológicos de forma detalhada, implementação e as análises realizadas. Por fim, são descritos os resultados e as comparações das implementações.

Em relação ao ambiente de execução, os testes foram realizados em um notebook Dell Latitude 3410 com processador $Intel\ Core\ i7-10510U\ CPU\ @\ 1.80GHz \times 8$, memória RAM de 8 GB, SSD de 256 GB e sistema operacional Ubuntu 20.04.3 LTS. Foi utilizado as seguintes configurações do LowMC: Número de rodadas, 4; Tamanho do bloco da cifra, 129 bits; e camada Sbox cheia.

A linguagem de programação utilizada foi C, com as seguintes flags de compilação O3, -march=native. Alguns scripts para teste ou organização de dados foram escritos em Python. Também foi utilizado o conjunto de instruções SSE2 Intrinsics da Intel para implementar a paralelização.

5.1 Implementação da paralelização a nível de instrução para q textos

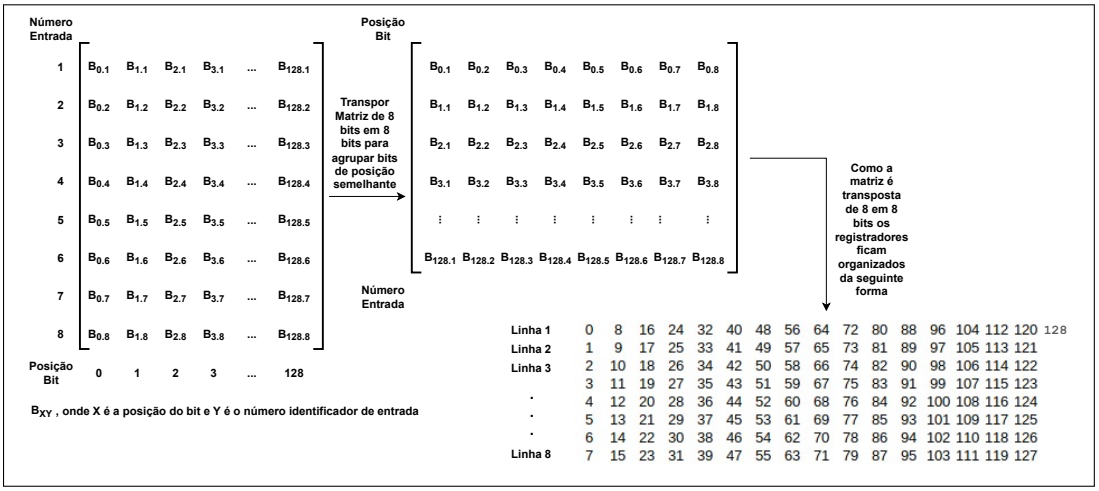
O processo de paralelização a nível de instrução foi decomposto e realizado da seguinte forma. Primeiro, a organização do estado principal que irá conter os bits dos q textos em questão. Após a organização, é necessário reescrever o algoritmo de encriptação do LowMC para encriptar os 8 textos em paralelo usando as instruções vetoriais. O algoritmo é melhor descrito na seção 2.4, mas ele é composto pelas funções de substituição (Sbox), adição de constante, multiplicação de matriz e adição de chave. Por fim, é necessário reorganizar os q textos processados.

5.1.1 Organização do estado principal

O estado principal é organizado no formato de matriz que depende de quantos q textos são processados paralelamente e o tamanho do bloco b em bits que são encriptados na cifra LowMC; porém, para fins de representação na implementação, b é representado em bytes. Portanto, o estado principal é representado como uma matriz $M_{q\lceil b \rceil}$. A notação $\lceil b \rceil$ tem o intuito de reservar espaço para alocar o byte sobressalente no caso de algumas versões do LowMC, como por exemplo, a de 129 bits.

A função de organização do estado funciona da seguinte forma. As linhas da matriz são organizadas de forma a agrupar os bits de mesma indexação, por exemplo, agrupar todos os bits de índice 0, de índice 1, e assim sucessivamente. É necessário organizar os bits para executar o processamento paralelo a partir das instruções vetoriais. A Figura 4, ilustra o funcionamento da função de organização do estado principal e o estado principal após a execução da função.

Figura 4 – Organização dos bits nos 8 registradores



Fonte: Autor deste trabalho

O código abaixo ilustra a função de organização do estado.

```
1  void mainStateGenerator(uint8_t in[][17], uint8_t out[][17]){
2      int j, k;
3      uint8_t comp = 0;
4      uint64_t aux = 0;
5      for(j=0; j<16; j++){
6          for(k=0; k<8; k++){
7              aux = aux << 8 | in[k][j];
8          }
9          aux = transposeMatrix8x8(aux);
10         for(k=7; k>=0; k--){
11             out[k][j] = aux;
12             aux = aux >> 8;
13         }
14     }
15     j = 0;
16     for(k=0; k<8; k++){
```

```
17          comp = (comp) | ((in[k][16] & 0x80) >> j++);
18      }
19      out[0][16] = comp;
20  }
```

Em resumo, a função transpõe a matriz de entrada com intuito de agrupar os bits de mesma indexação de forma paralela. O código da função que transpõe a matriz é melhor descrito no Anexo A deste trabalho. Por ser uma função de transposição de matriz 8×8, assim cada linha da matriz armazena os bits da entrada dado uma progressão aritmética de razão 8. Sendo assim, linha 1 armazena os q bits 0, 8, 16, e assim sucessivamente; a linha 2 armazena os q bits 1, 9, 17, e assim sucessivamente.

Por fim, realiza uma análise de desempenho da função de organização do estado para validar se tal esforço se justifica na hora da contabilização geral do processamento paralelo. A função que executa o benchmark de desempenho esta melhor descrita no Apêndice A. Dado a variabilidade dos resultados relacionado aos ciclos de clock do benchmark, foi coletado 100 amostras para ter uma média base de desempenho da função.

O gráfico de resultado e o vetor de dados das amostras podem ser verificados no Anexo B. A partir dos resultados, a média de execução da função foi de 609 ciclos, sendo o valor mínimo de 547 ciclos, o valor máximo de 699 ciclos e a mediana de 599 ciclos. Assim, temos o custo médio de organizar o estado principal.

5.1.2 Processamento paralelo com vetor de instrução

Neste tópico, será abordado um pouco mais detalhado as modificações necessária nas funções de encriptação do LowMC para paralelizar o algoritmo. Vale ressaltar, que nos tópicos abaixa será citado muito o estado armazenado do LowMC, que é a variável que acumula o resultado das funções e é operada por todas as funções de encriptação do LowMC.

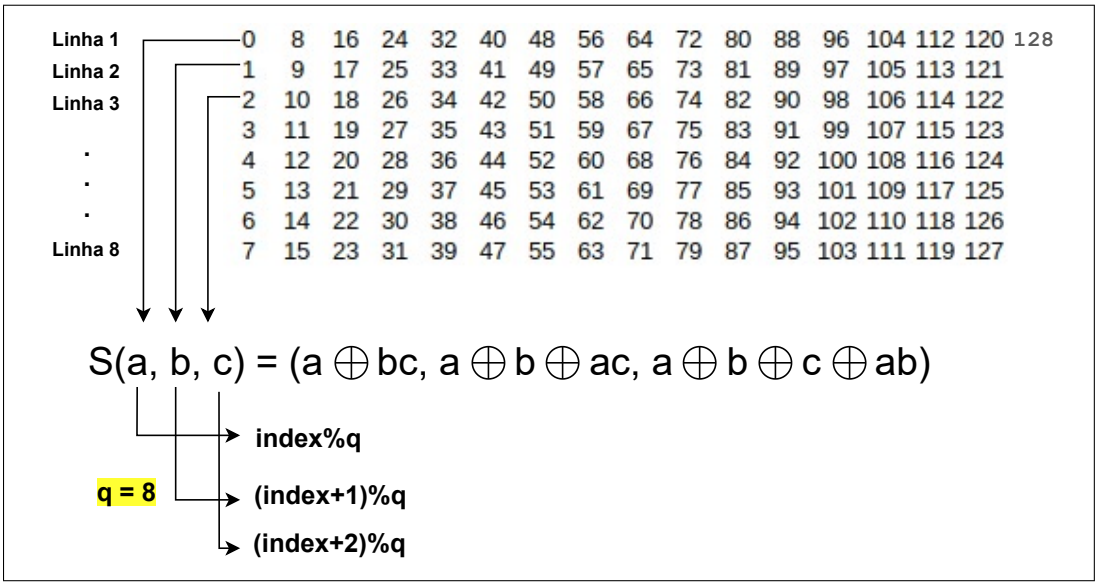
5.1.2.1 Função de substituição

A função de substituição recebe um único parâmetro de entrada que é o estado armazenado do LowMC. Essa mesma função, também conhecida como função Sbox, aplica a confusão entre os dados do estado armazenado utilizando a seguinte operação entre os bits. A cada interação 3 bits são processados pela seguinte função, $S(a,b,c) = (a \oplus bc, a \oplus b \oplus ac, a \oplus$

$b \oplus c \oplus ab$). Como são processados q textos em paralelo, onde q é igual a 8 nessa implementação, a cada iteração são processados 24 bits.

É necessário algumas observações nesta função por conta da forma que o estado principal está organizado. Os bits estão agrupados por indexação, logo, uma linha do estado não contém bits em sequencia e sim de mesmo índice. O bit posterior a qualquer atual está na linha abaixo dele e assim sucessivamente, então é necessário acessar 3 linhas toda iteração para pegar os bits a, b e c. A Figura 5 mostra quais bits capturar e executar na função de substituição para uma iteração.

Figura 5 – Iteração da função de substituição no estado principal



Fonte: Autor deste trabalho

O cálculo para pegar o bit correto é simples, necessitando apenas da iteração do index modulo q que é a quantidade de textos processados paralelamente. O outro índice necessário depende do loop mais externo que passa por todos os bytes que depende da quantidade de bits da versão do LowMC em bytes. O código abaixo descreve como funciona a função de substituição no estado principal.

```
1 static void substitution(uint8_t in[][17]) {
2     uint8_t a, b, c;
3     int index;
4     for(int i=0;i<15;i+=3){
5         index = 0;
6         for(int j=0; j<8; j++) {
7             a = in[(index+2)%8][i+((index+2)/8)];
```

```
8               b = in[(index+1)%8][i+((index+1)/8)];
9               c = in[index%8][i+(index/8)];
10              in[(index+2)%8][i+((index+2)/8)] = a ^ (b & c);
11              in[(index+1)%8][i+((index+1)/8)] = a ^ b ^ (a & c);
12              in[index%8][i+(index/8)] = a ^ b ^ c ^ (a & b);
13              index+=3;
14          }
15      }
16      index = 0;
17      for(int j=0; j<3; j++) {
18          a = in[(index+2)%8][15+((index+2)/8)];
19          b = in[(index+1)%8][15+((index+1)/8)];
20          c = in[index%8][15+(index/8)];
21          in[(index+2)%8][15+((index+2)/8)] = a ^ (b & c);
22          in[(index+1)%8][15+((index+1)/8)] = a ^ b ^ (a & c);
23          in[index%8][15+(index/8)] = a ^ b ^ c ^ (a & b);
24          index+=3;
25      }
26  }
```

5.1.2.2 Função de multiplicação de matriz

A função de multiplicação de matriz recebe quatro parâmetros de entrada, a variável de saída, a variável do estado armazenado do LowMC, o ponteiro para a matriz de multiplicação e o ponteiro para a matriz de multiplicação contendo os últimos bits. A matriz contendo os últimos bits é para tratar o byte sobressalente, que contém todos os bits de indexação 129 de todos os q textos.

Para realizar a multiplicação de forma eficiente, é preciso alterar a matriz instanciada pelo LowMC para o mesmo formato do estado principal. No estado principal, cada linha contém os bits organizados por mesma indexação, tal armazenamento segue uma progressão aritmética de módulo 8, como visto na Figura 4. Logo, é necessário também alterar a matriz para seguir a mesma organização do estado principal. Tal alteração é pré-processada, não afetando no desempenho do algoritmo.

Após organização da matriz o processo de multiplicação se torna intuitivo. Essa organização só é possível porque a matriz é fixa, uma vez que carregado as linhas do estado armazenado do LowMC e as linhas necessárias da matriz é só realizar uma operação AND entre as linhas e após isso realizar a operação XOR entre o estado armazenado. Abaixo, é ilustrado o código que realiza a função descrita anteriormente.

```
void matrix_multiplication(
    uint8_t output[][17],
    uint8_t state[][17],
    const uint8_t* matrix,
    const uint8_t* matrixLast)
{
    __m128i r[8], temp[8];
    int i,j,k;
    uint8_t out[129];
    uint8_t stateLast, last;
    for(i=0;i<8;i++){
      r[i] =_mm_loadu_si128((__m128i*)state[i+0]);
    }
    stateLast = state[0][16];
    for(i=0;i<129;i++){
        for(j=0;j<8;j++){
            temp[j] = _mm_loadu_si128((__m128i*)matrix+(i*8+j));
            temp[j] = _mm_and_si128(temp[j],r[j]);
        }
        last = stateLast & matrixLast[i];
        out[i] = parity128(temp, last);
    }
    for(i=0;i<8;i++){
        k = 0;
        for(j=i;j<128;j+=8){
            output[i][k++] = out[j];
        }
    }
    output[0][16] = out[128];
```

```
30 }
```

5.1.2.3 *Função de adição da constante de rodada*

A função de adição da constante de rodada recebe três parâmetros de entrada, a variável de saída, a variável do estado armazenado do LowMC e o ponteiro para constante da rodada em questão. Dentro desta função, ocorreram duas modificações para amparar a paralelização a nível de instrução.

A primeira alteração e mais importante é modificar o loop de adição da constante, que executa a mesma quantidade de vezes do número de bits da versão do LowMC que esta sendo processado. Por exemplo, para a versão do LowMC de 129 bits, uma implementação ingênua, executaria 129 vezes para fazer a operação XOR bit a bit entre o estado armazenado do LowMC e a constante da rodada em questão. Este método é alterado para utilizar vetor de instrução que executa a operação XOR entre os registradores em 1 ciclo de clock.

É utilizado a instrução *_mm_loadu_si*128 para carregar os dados para os registradores e depois a instrução *_mm_xor_si*128 para executar a operação XOR entre os registradores, tais instruções são do conjunto de instruções *SSE*2. Vale ressaltar que a constante da rodada em questão é iterada dentro do loop para pegar a constante de rodada correta para cada texto.

A segunda alteração é um loop que depende da quantidade de q textos que são paralelizados para processar cada estado por vez. Abaixo, é ilustrado o código que realiza a função descrita anteriormente.

```
1  void xor_roundConst(uint8_t out[][17], uint8_t state[][17], const
       uint8_t *roundConst)
2  {
3      int i;
4    __m128i r1,r2;
5      for(i = 0;i<8;i++){
6          r1 = _mm_loadu_si128((__m128i*)roundConst[i]);
7          r2 = _mm_loadu_si128((__m128i*)state + (i));
8          r1 = _mm_xor_si128(r1,r2);
9          _mm_storeu_si128((__m128i*)out[i],r1);
10     }
11     out[0][16] = roundConst[0][16] ^ state[128];
```

```
12 }
```

5.1.2.4 Função de adição da chave de rodada

A função de adição da chave de rodada é semelhante a função anterior, recebendo três parâmetros de entrada, a variável de saída, a variável do estado armazenado do LowMC e ao invés da constante da rodada em questão recebe a chave da rodada em questão.

Na implementação, a única diferença é que a chave da rodada é uma variável que está no mesmo formato do estado armazenado do LowMC. Abaixo, é ilustrado o código que realiza a função descrita anteriormente.

```
1  void xor_roundKey(uint8_t out[][17], uint8_t state[][17], uint8_t
       roundKey[][17])
2  {
3      int i;
4    __m128i r1,r2;
5      for(i = 0;i<8;i++){
6          r1 = _mm_loadu_si128((__m128i*)state[i]);
7          r2 = _mm_loadu_si128((__m128i*)roundKey[i]);
8          r1 = _mm_xor_si128(r1,r2);
9          _mm_storeu_si128((__m128i*)out[i],r1);
10     }
11     out[0][16] = state[0][16] ^ roundKey[0][16];
12 }
```

5.1.3 Reorganizar os q textos processados

Após o processamento paralelo do estado principal dentro do LowMC, é necessário reorganizar os q textos encriptados. A função que executa o processo de reorganização dos textos é extremamente semelhante à função que organiza o estado principal. A única diferença da função são as ultimas linhas que tratam o bit de índex 128. Abaixo é ilustrado o código que devolve tal bit para seus textos.

```
1      i = 7;
```

```
2     for (k=7; k>=t; k--, i--)
3         out[k][16] = (in[0][16] << i) & 0x80;
```

Por fim, a função que reorganiza os textos reagrupa os bits de cada texto de acordo com sua indexação. É possível concluir que o mesmo levantamento de processamento computacional realizado no tópico 5.1.1 é válido também para essa função.

5.2 Buscar otimizações para a camada afim

Durante o período de pesquisa e análise por um método para aprimorar a camada afim, alguns algoritmos foram avaliados para otimizar o produto da matriz pelo vetor. Um deles, foi o "Mailman algorithm" (LIBERTY; ZUCKER, 2009), porém, este algoritmo é mais adequado quando aplicado a valores pertencentes ao conjunto dos reais. Existem outros métodos mais eficientes encontrados como "Strassen-Winograd method" (HUSS-LEDERMAN *et al.*, 1996), porém, não fazem sentido dado a dimensão que estamos considerando no LowMC.

Um método encontrado e que pode ser implementado na camada afim para otimizar o produto da matriz pelo vetor é o "method of the four Russians" (AHO *et al.*, 1972). Com tal método, é possível reduzir a complexidade assintótica do produto de $O(n^2)$ para $O(n^2/log(n))$, sendo mais efetivo que o método padrão de multiplicação.

O método aplica o conceito partição de matriz e tabela "lookup" para acelerar o produto de duas matrizes. A partir da partição de $log\ n$ segmentos da matriz $n \times n$ como visto na Figura 6, é possível verificar a codificação dos bits e pesquisar na tabela o resultado pré processado desse arranjo de bits.

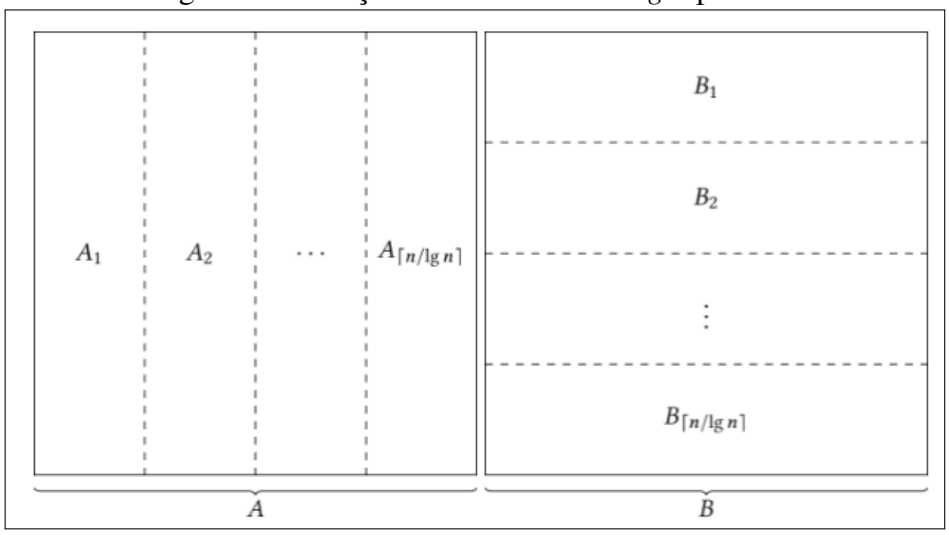
Logo, temos que o resultado do produto de matrizes $AB = \sum_{i=1}^{\lceil logn \rceil} A_i B_i$. Para validar a eficiência do método, apresentado foi realizado um benchmark das implementações, baseadas no trabalho de (Spyridon Pongas), com a finalidade de validar a preeminência assintótica do "method of the four Russians" em ciclos de clock. A Tabela 1, ilustra como foi o desempenho de ambas implementações em ciclos de clock para a multiplicação de matrizes binárias $n \times n$.

Tabela 1 – Benchmark métodos de multiplicação matriz binária

Bits / Ciclos de clock	8	16	24	32	48	64	80	96	112	129
Método simples(ciclos)	8×10^3	$5{,}6\times10^4$	$1{,}9\times10^5$	$4{,}8\times10^5$	$1{,}6\times10^6$	$3{,}8\times10^6$	$7{,}8\times10^6$	$1{,}3\times10^7$	$2{,}2\times10^7$	$3{,}1\times10^7$
Método dos 4 Russos(ciclos)	$1{,}1\times10^4$	$3{,}5\times10^4$	$7{,}3\times10^4$	$1{,}5\times10^5$	$3{,}4\times10^5$	$6{,}2\times10^5$	$1{,}1\times10^6$	$1{,}6\times10^6$	$2{,}5\times10^6$	$2{,}9\times10^6$

Fonte: Autor deste trabalho

Figura 6 – Partição das matrizes em *log n* partes



Fonte: Livro do autor (AHO *et al.*, 1974)

É importante ressaltar que para nossa implementação, o resultado esperado de desempenho seria um pouco abaixo do que será mostrado a seguir por conta que no LowMC é o produto de uma matriz $n \times n$ por um vetor n. Porém, é nítido a vantagem de utilizar o "method of the four Russians", a discrepância de resultados fica ainda mais nítida quando apresentado o gráfico de desempenho.

No futuro, é possível utilizar essa técnica para melhorar o desempenho da camada afim diminuindo assim o gargalo existente.

5.3 Análise de desempenho e benchmark

Nesta seção será mostrado de forma geral a comparação de desempenho entre a versão paralela e a versão de referência do LowMC, como também resultados de benchmark entre elas.

Foi realizada uma análise de desempenho do algoritmo LowMC em sua versão de referência, proposta em Albrecht *et al.* (2016), para a encriptação de um único texto. Dado a variabilidade dos resultados relacionado aos ciclos de clock do benchmark, foram coletadas 100 amostras para ter uma média base de desempenho da função. O gráfico de resultado e o vetor de dados das amostras podem ser verificados no Anexo C. A partir dos resultados, a média de execução da função foi de 121294 ciclos, sendo o valor mínimo de 109455 ciclos, o valor máximo de 147513 ciclos e a mediana de 121294 ciclos. Assim, temos o custo médio do processamento de cada texto de 129 bits na versão de referência do LowMC.

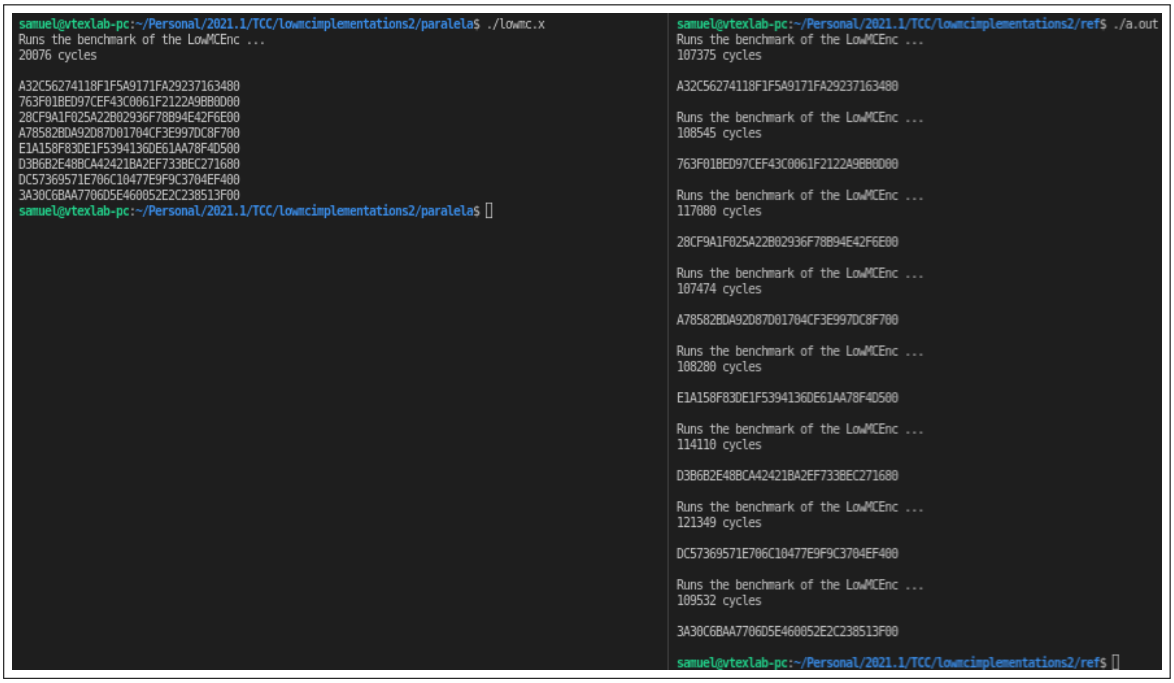
Também foi realizado uma análise de desempenho do algoritmo LowMC em sua

versão paralela, implementada neste trabalho. A versão paralela em questão encripta 8 textos de 129 bits, dado a variabilidade dos resultados relacionado aos ciclos de clock do benchmark, foram coletadas 100 amostras para ter uma média base de desempenho da função. O gráfico de resultado e o vetor de dados das amostras podem ser verificados no Anexo D. A partir dos resultados, a média de execução da função foi de 17637 ciclos, sendo o valor mínimo de 15729 ciclos, o valor máximo de 20985 ciclos e a mediana de 17101 ciclos. Assim, temos o custo médio do processamento de 8 textos de 129 bits encriptados paralelamente.

Logo, se multiplicarmos a mediana do processamento da versão de referência do LowMC por 8, teríamos o custo de processar 8 textos sem ser em paralelo, igual a 970352 ciclos. Para a versão paralela temos que considerar a média de processamento da versão paralela do LowMC com o valor de processamento de organizar o estado e reorganizar os q textos. Multiplicando a mediana do processamento do estado por 2, a mediana de organização do estado e reorganização dos q textos é 599, logo o valor adicional é 1198. Então o processamento necessário seria de 18299 ciclos.

Temos que a versão implementada neste trabalho tem uma eficiência de desempenho a versão de referência de 53 vezes para a encriptação de 8 textos. Na Figura 7 é possível vê o resultado da versão paralela a esquerda, com o benchmark e os textos processados. E a direita a versão de referência com o benchmark de cada texto e os textos também processados.

Figura 7 – Resultado comparando a integridade dos dados



Fonte: Autor deste trabalho

Uma das propostas deste trabalho era comparar este trabalho com o trabalho de Kales *et al.* (2017) que atualmente tem a versão mais otimizada do LowMC. É válido ressaltar que este trabalho implementa algumas otimizações algébricas para otimizar a camada afim e também utiliza instruções vetoriais para otimizar o desempenho. Foi realizado a análise de desempenho desta versão para o LowMC de 129 bits, , dado a variabilidade dos resultados relacionado aos ciclos de clock do benchmark, foram coletadas 100 amostras para ter uma média base de desempenho da função.

O gráfico de resultado e o vetor de dados das amostras podem ser verificados no Anexo E. A partir dos resultados, a média de execução da função foi de 2518 ciclos, sendo o valor mínimo de 2213 ciclos, o valor máximo de 3037 ciclos e a mediana de 2454 ciclos. Assim, temos o custo médio do processamento de 8 textos de 129 bits encriptados paralelamente.

Por fim, comparamos os resultados da versão implementada neste trabalho com a versão do trabalho de Kales *et al.* (2017). Se multiplicarmos a mediana do processamento da versão de Kales *et al.* (2017) do LowMC por 8, teríamos o custo de processar 8 textos sem ser em paralelo, igual a 19632 ciclos.

Para a versão paralela implementada neste trabalho e já comparada com a versão de referência sabemos que a mediana do processamento necessário é 18299 ciclos.

Por fim, comprovamos que a versão implementada neste trabalho é ligeiramente mais eficiente que a versão do trabalho de Kales *et al.* (2017) para a encriptação de 8 textos. Vale destacar que a implementação de kales foi altamente otimizada e a nossa implementação ainda temos margem de implementar otimizações em algumas camadas como, por exemplo, Sbox e aprimorar gerenciamento de memória.

6 CONCLUSÕES E TRABALHOS FUTUROS

Visando buscar otimizações na cifra criptográfica LowMC com intuito de indiretamente aumentar o desempenho da assinatura digital Picnic, este trabalho propôs a implementação de uma versão paralela do LowMC tendo em vista que este mesmo algoritmo é executado algumas dezenas de vezes de forma independente dentro do Picnic.

A versão do LowMC desenvolvida neste trabalho, busca trazer maior eficiência para algoritmos ou métodos que utilizam desta cifra. A primórdio foi desenvolvida para otimizar o Picnic, porém com a constante evolução da tecnologia e criptografia, dado as vantagens de obter poucas operações não-lineares a cifra LowMC pode se tornar cada vez mais relevante para o meio acadêmico.

Outro resultado extremamente importante encontrado neste trabalho é a otimização na camada afim onde ocorre o maior gargalo do algoritmo. Mesmo não tendo implementado diretamente no LowMC o método “methodof the four Russians“ é nítido a otimização na operação de produto entre matriz e o vetor.

Como trabalhos futuros o objetivo é aplicar o método de multiplicação “methodof the four Russians” para confirmar a sua eficiência no LowMC, como também implementar outras versões paralelas do LowMC como por exemplo, 16 textos, 32 textos, e assim por diante. Além disso, alterar também as versões em bits do LowMC, 192 bits, 256 bits.

REFERÊNCIAS

AARONSON, S. BQP and the polynomial hierarchy. In: **Proceedings of the forty-second ACM symposium on Theory of computing**. [*S. l.*]: ACM, 2010. p. 141–150.

AHO, A. V.; GAREY, M. R.; ULLMAN, J. D. The transitive reduction of a directed graph. **SIAM J. Comput.**, [S.l], v. 1, n. 2, p. 131–137, 1972.

AHO, A. V.; HOPCROFT, J. E.; ULLMAN, J. D. **The Design and Analysis of Computer Algorithms**. [*S. l.*]: Addison-Wesley, 1974.

ALBRECHT, M. R.; RECHBERGER, C.; SCHNEIDER, T.; TIESSEN, T.; ZOHNER, M. Ciphers for MPC and FHE. **IACR Cryptol. ePrint Arch.**, [S.l], p. 687, 2016.

BENADJILA, R.; GUO, J.; LOMNÉ, V.; PEYRIN, T. Implementing lightweight block ciphers on x86 architectures. In: **Selected Areas in Cryptography**. [*S. l.*]: Springer, 2013. (Lecture Notes in Computer Science, v. 8282), p. 324–351.

CHASE, M.; DERLER, D.; GOLDFEDER, S.; ORLANDI, C.; RAMACHER, S.; RECHBERGER, C.; SLAMANIG, D.; ZAVERUCHA, G. Post-quantum zero-knowledge and signatures from symmetric-key primitives. In: **Proceedings of the 2017 acm sigsac conference on computer and communications security**. [*S. l.*: *s. n.*], 2017. p. 1825–1842.

CRAMER, R.; DAMGÅRD, I.; NIELSEN, J. B. **Secure Multiparty Computation and Secret Sharing**. Cambridge University Press, 2015. ISBN 9781107043053. Disponível em: http://www.cambridge.org/de/academic/subjects/computer-science/cryptography-cryptology-and-coding/secure-multiparty-computation-and-secret-sharing?format=HB&isbn=9781107043053. Acesso em: 17 fev. 2021.

GELLERSEN, T.; SEKER, O.; EISENBARTH, T. Differential power analysis of the picnic signature scheme. **IACR Cryptol. ePrint Arch.**, [S.l], v. 2020, p. 267, 2020.

HUQING, W.; ZHIXIN, S. Research on zero-knowledge proof protocol. **International Journal of Computer Science Issues (IJCSI)**, Citeseer, [S.l], v. 10, n. 1, p. 194, 2013.

HUSS-LEDERMAN, S.; JACOBSON, E. M.; JOHNSON, J. R.; TSAO, A.; TURNBULL, T. Implementation of strassen's algorithm for matrix multiplication. In: IEEE. **Supercomputing'96: Proceedings of the 1996 ACM/IEEE Conference on Supercomputing**. [*S. l.*], 1996. p. 32–32.

KALES, D.; PERRIN, L.; PROMITZER, A.; RAMACHER, S.; RECHBERGER, C. Improvements to the linear operations of lowmc: A faster picnic. **Cryptology ePrint Archive**, [S.l], 2017.

KALES, D.; ZAVERUCHA, G. Improving the performance of the picnic signature scheme. **IACR Transactions on Cryptographic Hardware and Embedded Systems**, [S.l], p. 154–188, 2020.

LIBERTY, E.; ZUCKER, S. W. The mailman algorithm: A note on matrix–vector multiplication. **Information Processing Letters**, Elsevier, [S.l], v. 109, n. 3, p. 179–182, 2009.

MILLER, V. S. Use of elliptic curves in cryptography. In: SPRINGER. **Conference on the theory and application of cryptographic techniques**. [*S. l.*], 1985. p. 417–426.

NICOLIELLO, H. **Introdução à computação quântica**. [S.l: s.n], 2009.

NYBERG, K. Perfect nonlinear s-boxes. In: SPRINGER. **Workshop on the Theory and Application of of Cryptographic Techniques**. [*S. l.*], 1991. p. 378–386.

OLIVEIRA, I. C. *et al.* **Complexidade computacional e o problema p vs np**. [S.l: s.n], 2010.

RABÊLO, R. C. F. *et al.* **Implementação eficiente das funções de resumo criptográfico:**SHA-3 e QUARK. [S.l: s.n], 2015.

RIVEST, R. L.; SHAMIR, A.; ADLEMAN, L. A method for obtaining digital signatures and public-key cryptosystems. **Communications of the ACM**, ACM New York, NY, USA, v. 21, n. 2, p. 120–126, 1978.

SCHNEIER, B. **Applied cryptography** - protocols, algorithms, and source code in C. 2nd. Wiley, [S.l], 1996. Disponível em: https://www.worldcat.org/oclc/32311687. Acesso em: 17 fev. 2021.

SHOR, P. W. Algorithms for quantum computation: discrete logarithms and factoring. In: IEEE. **Proceedings 35th annual symposium on foundations of computer science**. [*S. l.*], 1994. p. 124–134.

SIDORENCO, N.; OECHSNER, S.; SPITTERS, B. Formal security analysis of mpc-in-the-head zero-knowledge protocols. **IACR Cryptol. ePrint Arch.**, [S.l], v. 2021, p. 437, 2021.

SONI, D.; BASU, K.; NABEEL, M.; AARAJ, N.; MANZANO, M.; KARRI, R. Picnic. In: **Hardware Architectures for Post-Quantum Digital Signature Schemes**. [*S. l.*]: Springer, 2021. p. 121–139.

VIGNATTI, A. L.; NETTO, F.; BITTENCOURT, L. F. **Uma introdução à computação quântica**. Departamento de Informática. UFPR, 2004.

WARREN, H. S. **Hacker's delight**. [*S. l.*]: Pearson Education, 2013.

APÊNDICE A – CÓDIGO DE BENCHMARK DE DESEMPENHO

A.0.1 Código relativo ao benchmark.c

```
#include <stdio.h>
#include "bench.h"
/* Private definitions */
// Timer type.
typedef unsigned long long bench_t;
static inline bench_t cycles(void) {
  unsigned int hi, lo;
  __asm volatile ("rdtsc\n\t":"=a" (lo), "=d"(hi));
  return ((bench_t) lo) | (((bench_t) hi) << 32);
}
// Stores the time measured before the execution of the benchmark.
static bench_t before;
// Stores the time measured after the execution of the benchmark.
static bench_t after;
// Stores the sum of timings for the current benchmark.
long long total;
/* Public definitions */
void bench_reset() {
  total = 0;
}
void bench_before() {
  before = cycles();
}
void bench_after() {
  long long result;
  after = cycles();
  result = (after - before);
  total += result;
}
void bench_compute(int benches) {
  total = total / benches;
```

```
}
void bench_print() {
  printf("%lld cycles\n", total);
  printf("\n");
}
unsigned long long bench_get_total() {
  return total;
}
```

A.0.2 Código relativo ao benchmark.h

```
#ifndef _BENCH_H_
#define _BENCH_H_
// Bench a function FUNCTION that receives many __VA_ARGS__
   parameters.
#define BENCH_FUNCTION(FUNC, ...)\
do{\
    int _bench_i_ = 0, _bench_j_ = 0;\
    BENCH_BEGIN(#FUNC, BENCH){\
        BENCH_ADD(FUNC(__VA_ARGS__), BENCH);\
    }\
    BENCH_END(BENCH);\
}while(0)
/*
  Runs a new benchmark once.
  @param[in] LABEL - the label for this benchmark.
  @param[in] FUNCTION - the function to benchmark.
*/
#define BENCH_ONCE(LABEL, FUNCTION)\
  bench_reset();\
  printf("Initializes the %s ... \n", LABEL);\
  bench_before();\
  FUNCTION;\
/*
```

```
23    Runs a new benchmark.
24    @param[in] LABEL - the label for this benchmark.
25  */
26  #define BENCH_BEGIN(LABEL, BENCH)\
27    bench_reset();\
28    printf("Runs the benchmark of the %s ... \n", LABEL);\
29    for(_bench_i_ = 0; _bench_i_ < BENCH; _bench_i_++){\
30      // Prints the mean timing of each execution in nanoseconds.
31      #define BENCH_END(BENCH)\
32    }\
33    bench_compute(BENCH * BENCH);\
34    bench_print();\
35  /*
36    Measures the time of one execution and adds it to the benchmark
       total.
37    @param[in] FUNCTION - the function executed.
38  */
39  #define BENCH_ADD(FUNCTION, BENCH)\
40    FUNCTION;\
41    bench_before();\
42    for (_bench_j_ = 0; _bench_j_ < BENCH; _bench_j_++){\
43      FUNCTION;\
44    }\
45    bench_after();\
46  /* Function prototypes */
47  /*
48    Resets the benchmark data.
49    @param[in] label - the benchmark label.
50  */
51  void bench_reset(void);
52  // Measures the time before a benchmark is executed.
53  void bench_before(void);
54  // Measures the time after a benchmark was started and adds it to
     the total.
```

```
55 void bench_after(void);
56 /*
57   Computes the mean elapsed time between the start and the end of a
       benchmark.
58   @param benches - the number of executed benchmarks.
59 */
60 void bench_compute(int benches);
61 // Prints the last benchmark.
62 void bench_print(void);
63 /*
64   Returns the result of the last benchmark.
65   @return the last benchmark.
66 */
67 unsigned long long bench_get_total(void);
68 #endif
```

ANEXO A – FUNÇÃO PARA TRANSPOR MATRIZ 8X8

```
#include <inttypes.h>
uint64_t transposeMatrix8x8(uint64_t x) {
    x = (x & 0xAA55AA55AA55AA55LL) | ((x & 0x00AA00AA00AA00AALL) <<
        7)  | ((x >> 7) & 0x00AA00AA00AA00AALL);
    x = (x & 0xCCCC3333CCCC3333LL) | ((x & 0x0000CCCC0000CCCCLL) <<
        14) | ((x >> 14) & 0x0000CCCC0000CCCCLL);
    x = (x & 0xF0F0F0F00F0F0F0FLL) | ((x & 0x00000000F0F0F0F0LL) <<
        28) | ((x >> 28) & 0x00000000F0F0F0F0LL);
    return x;
}
```

Fonte: Capítulo 7.3 do livro (WARREN, 2013)

ANEXO B – VETOR E GRÁFICO AMOSTRAL DOS RESULTADOS DO BENCHMARK DA ORGANIZAÇÃO DO ESTADO PRINCIPAL

Figura 8 – Gráfico amostral dos resultados

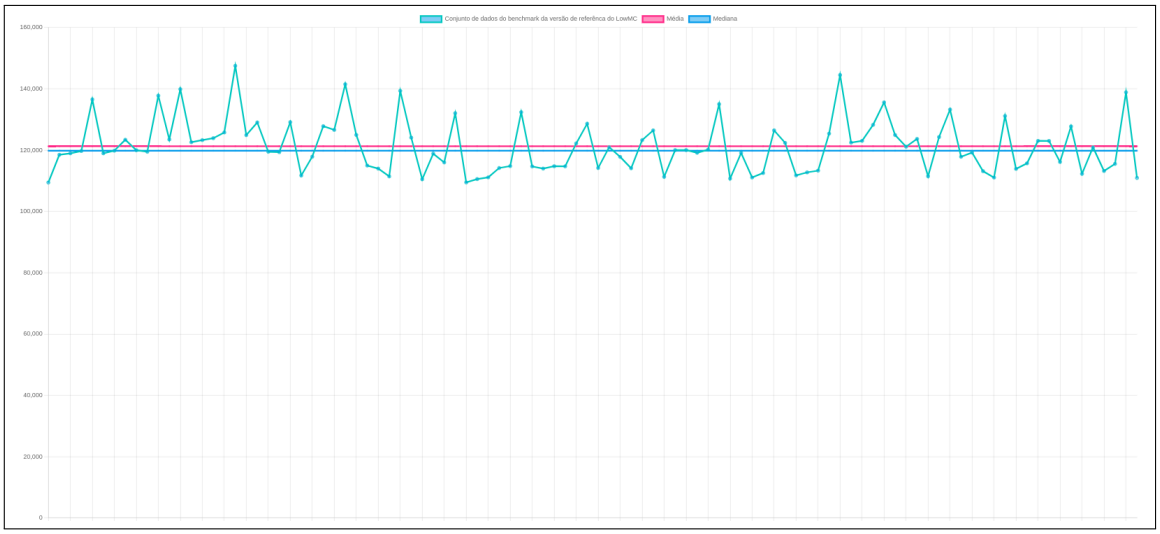
Fonte: Autor deste trabalho em chartjs.org

B.0.1 Vetor das amostras

Vetor das amostras (valor em ciclos de clock): [612, 589, 566, 674, 668, 695, 653, 650, 660, 686, 699, 670, 651, 656, 661, 620, 599, 595, 603, 608, 612, 594, 594, 595, 596, 602, 571, 591, 594, 599, 603, 594, 583, 602, 579, 597, 595, 572, 587, 589, 611, 590, 683, 652, 670, 661, 627, 601, 623, 667, 565, 656, 575, 616, 598, 580, 598, 589, 600, 592, 594, 594, 639, 603, 591, 590, 603, 606, 591, 578, 624, 594, 611, 577, 600, 599, 600, 608, 593, 592, 594, 617, 601, 594, 606, 614, 594, 589, 588, 600, 597, 627, 592, 547, 591, 619, 595, 594, 598, 599]

ANEXO C – VETOR E GRÁFICO AMOSTRAL DOS RESULTADOS DO BENCHMARK DA VERSÃO DE REFERÊNCIA DO LOWMC

Figura 9 – Gráfico amostral dos resultados da versão de referência



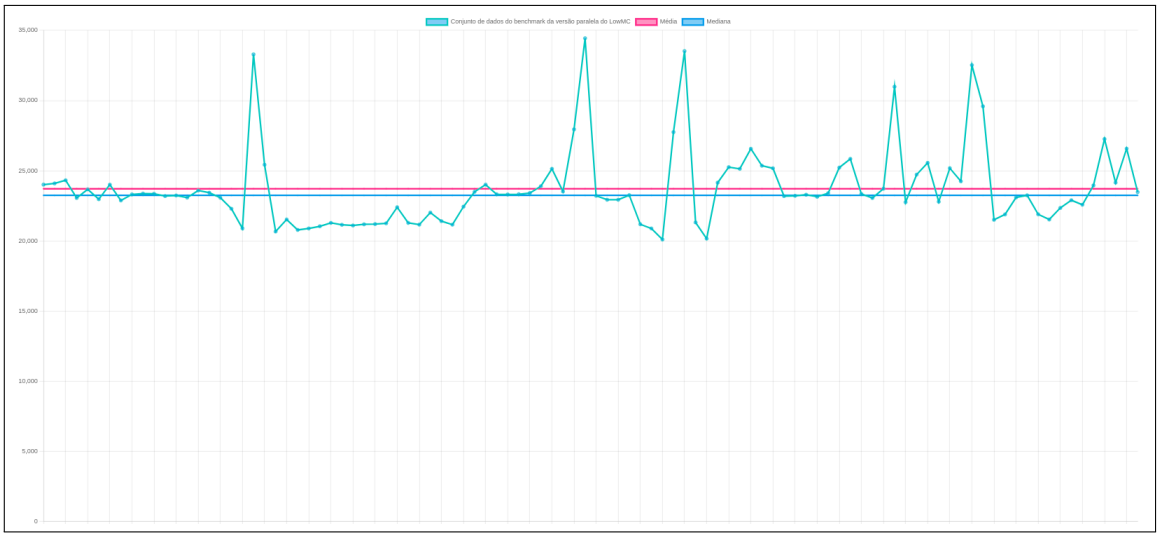
Fonte: Autor deste trabalho em chartjs.org

C.0.1 Vetor das amostras

Vetor das amostras (valor em ciclos de clock): [109455, 118508, 118967, 119728, 136560, 118976, 119827, 123394, 119983, 119487, 137795, 123502, 139893, 122600, 123285, 123917, 125788, 147513, 124918, 129053, 119490, 119350, 129140, 111734, 117878, 127838, 126621, 141518, 124982, 114979, 113999, 111446, 139354, 124103, 110522, 118872, 116011, 132065, 109477, 110580, 111139, 114190, 114809, 132405, 114690, 114003, 114764, 114733, 122202, 128642, 114198, 121002, 117821, 114109, 123293, 126456, 111302, 120018, 120024, 119156, 120181, 135019, 110751, 119116, 111095, 112558, 126467, 122391, 111770, 112766, 113329, 125403, 144518, 122463, 123047, 128268, 135592, 124946, 121127, 123681, 111491, 124279, 133261, 117856, 119229, 113149, 111082, 131140, 113913, 115722, 123038, 123023, 116171, 127750, 112289, 120945, 113225, 115536, 138865, 110952]

ANEXO D – VETOR E GRÁFICO AMOSTRAL DOS RESULTADOS DO BENCHMARK DA VERSÃO PARALELA DO LOWMC

Figura 10 – Gráfico amostral dos resultados da versão paralela



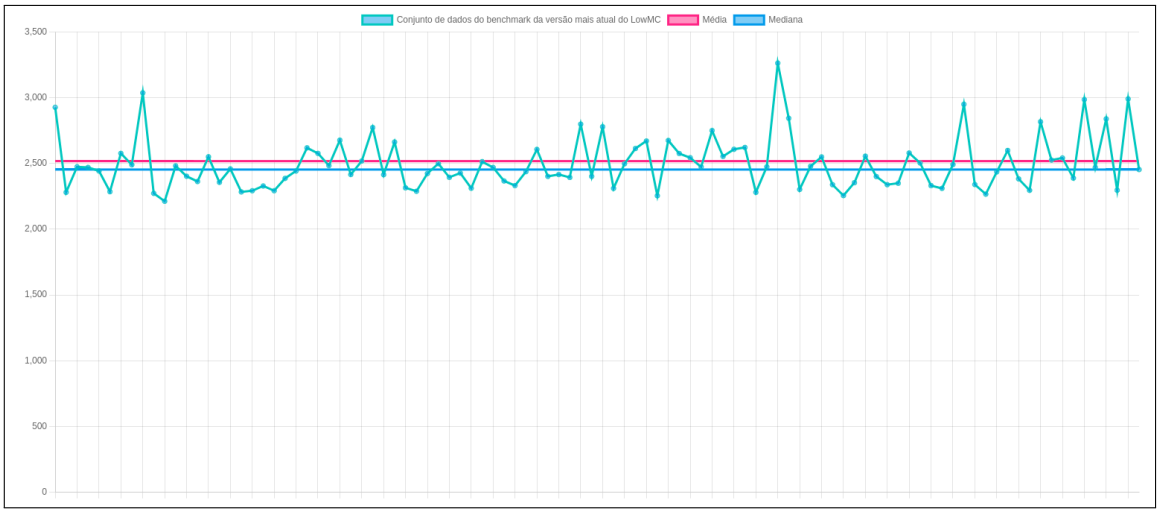
Fonte: Autor deste trabalho em chartjs.org

D.0.1 Vetor das amostras

Vetor das amostras (valor em ciclos de clock): [24016, 24100, 24322, 23063, 23678, 22975, 24011, 22886, 23318, 23374, 23362, 23197, 23242, 23090, 23596, 23436, 23080, 22295, 20897, 33292, 25428, 20673, 21532, 20788, 20892, 21046, 21288, 21154, 21105, 21186, 21204, 21257, 22402, 21288, 21166, 22023, 21413, 21163, 22445, 23501, 24014, 23326, 23312, 23330, 23414, 23900, 25140, 23526, 27949, 34438, 23217, 22938, 22938, 23262, 21189, 20891, 20105, 27753, 33525, 21319, 20167, 24153, 25257, 25138, 26569, 25361, 25177, 23196, 23219, 23296, 23148, 23404, 25226, 25848, 23350, 23062, 23725, 30987, 22761, 24730, 25565, 22795, 25185, 24253, 32526, 29590, 21511, 21891, 23102, 23252, 21899, 21530, 22355, 22902, 22583, 23957, 27260, 24155, 26577, 23484]

ANEXO E – VETOR E GRÁFICO AMOSTRAL DOS RESULTADOS DO BENCHMARK DA VERSÃO DE KALES

Figura 11 – Gráfico amostral dos resultados da versão de Kales



Fonte: Autor deste trabalho em chartjs.org

E.0.1 Vetor das amostras

Vetor das amostras (valor em ciclos de clock): [2927, 2281, 2475, 2470, 2443, 2286, 2577, 2490, 3037, 2273, 2213, 2482, 2403, 2363, 2551, 2357, 2459, 2284, 2293, 2329, 2293, 2387, 2443, 2619, 2577, 2484, 2677, 2416, 2519, 2773, 2415, 2663, 2315, 2289, 2424, 2499, 2395, 2427, 2312, 2514, 2470, 2367, 2332, 2438, 2607, 2402, 2416, 2394, 2799, 2402, 2779, 2311, 2498, 2614, 2671, 2255, 2675, 2576, 2544, 2474, 2750, 2554, 2608, 2622, 2281, 2475, 3263, 2844, 2303, 2479, 2550, 2339, 2256, 2354, 2556, 2401, 2339, 2350, 2580, 2504, 2332, 2311, 2492, 2950, 2340, 2267, 2436, 2599, 2383, 2296, 2817, 2526, 2542, 2389, 2986, 2470, 2839, 2297, 2991, 2454, 2297]