UNIVERSIDADE FEDERAL DE MATO GROSSO DO SUL
FACULDADE DE COMPUTAÇÃO
TRABALHO DE CONCLUSÃO DE CURSO

GABRIEL BARBOSA ALCÂNTARA
GABRIEL HIROFUMI OKANO

ESTUDO E IMPLEMENTAÇÃO DO NOVO PADRÃO DE CRIPTOGRAFIA LEVE ASCON

Campo Grande - MS
4 de dezembro de 2023

GABRIEL BARBOSA ALCÂNTARA
GABRIEL HIROFUMI OKANO

ESTUDO E IMPLEMENTAÇÃO DO NOVO PADRÃO DE CRIPTOGRAFIA LEVE ASCON

Trabalho de Conclusão do Curso de Engenharia da Computação da Faculdade de Computação da Universidade Federal de Mato Grosso do Sul, como requisito para obtenção do diploma de bacharel em Engenharia da Computação

Orientador(a): Dra. Ana Karina Dourado Salina de Oliveira

Campo Grande - MS
4 de dezembro de 2023

Resumo

Esse trabalho tem como objetivo principal apresentar conceitos fundamentais de criptografia, explorando as áreas de criptografia leve e pós-quântica. O foco foi dirigido para o estudo e testes do algoritmo ASCON, que é o padrão de algoritmo de criptografia leve, destacando sua relevância na literatura atual. A análise abrange conceitos-chave, como autenticação criptográfica, resistência criptográfica, funções hash e destaca o ASCON como um componente significativo no cenário criptográfico contemporâneo. Por fim, são apresentadas direções futuras para trabalhos de pesquisa.

Sumário

1 Introdução

A Segurança da Informação desempenha um papel crucial na era digital, onde a crescente troca e armazenamento de dados demandam medidas robustas de proteção. A criptografia emerge como ferramenta essencial para assegurar a confidencialidade e integridade das informações, sendo fundamental para preservar a confiança nas transações, comunicações, na privacidade individual e na proteção contra ameaças cibernéticas, garantindo que apenas usuários autorizados tenham acesso aos dados sensíveis. Essa prática, codifica dados, garantindo acesso autorizado e resguardando a confiança no ambiente digital.

Como pilar fundamental da Segurança da Informação, a criptografia consiste na prática de codificar e decodificar dados para proteger sua confidencialidade e autenticidade. Essa ciência, baseada em algoritmos matemáticos complexos, é crucial para preservar a privacidade e segurança das informações no cenário digital em constante evolução, onde a troca e armazenamento de dados ocorrem exponencialmente.[1]

Com o aumento exponencial do uso de dispositivos com recursos limitados, como sensores IoT (Internet das Coisas) e dispositivos embarcados, a necessidade de garantir a segurança desses sistemas tornou-se urgente. Esses dispositivos, muitas vezes com capacidades de processamento e armazenamento restritas, demandam soluções de criptografia específicas para garantir a proteção adequada dos dados que manipulam. Tais algoritmos são adaptados para implementação nesses ambientes limitados, como sensores, cartões inteligentes, dispositivos de cuidados de saúde, entre outros, uma vez que implementar algoritmos tradicionais acaba se tornando uma tarefa desafiadora.[2]

2 Conceitos Básicos de Criptografia

Neste capítulo, apresentaremos alguns conceitos básicos de criptografia, necessários para compreensão.

2.1 Termos utilizados na segurança da informação

Três termos da segurança da informação são frequentemente utilizados, considere dois pontos A e B:

- **Autenticidade** - A tem certeza que está de fato comunicando com B, e não um agente impostor, e vice-versa;
- **Confidencialidade** - Caso a comunicação entre estes dois pontos for interceptada por um agente externo, será impossível recuperar o conteúdo da mensagem;
- **Integridade** - Garante que a mensagem não foi alterada durante o transporte entre A e B;

2.2 Cifras de Bloco e Cifra de Fluxo

De forma geral, as cifras são classificadas em duas famílias: [3]

- **Cifras de Fluxo** - Dados são processados bit por bit, a chave é utilizada como *seed* de um *random number generator* (RNG), que servirá para cifrar e decifrar uma mensagem;
- **Cifras de Bloco** - Dados são separados em blocos de tamanho fixo, e encriptados bloco por bloco, a chave é utilizada para cifrar e decifrar cada bloco;

2.3 Criptografia Simétrica

O esquema de criptografia simétrica, permite a comunicação segura entre duas partes, que compartilham da mesma chave K, que é utilizada para cifrar e decifrar uma mensagem M. Conforme ilustrado na Figura 1, Alice envia a mensagem "Hello Word", encriptando com uma chave K, que será decriptado por Bob usando a mesma chave. Esse processo é idêntico ao reverso, se Bob enviar uma mensagem para Alice o processo será o mesmo. Se a mensagem for interceptada por Eve (eavesdropper), ela não conseguirá entender o conteúdo da mensagem.

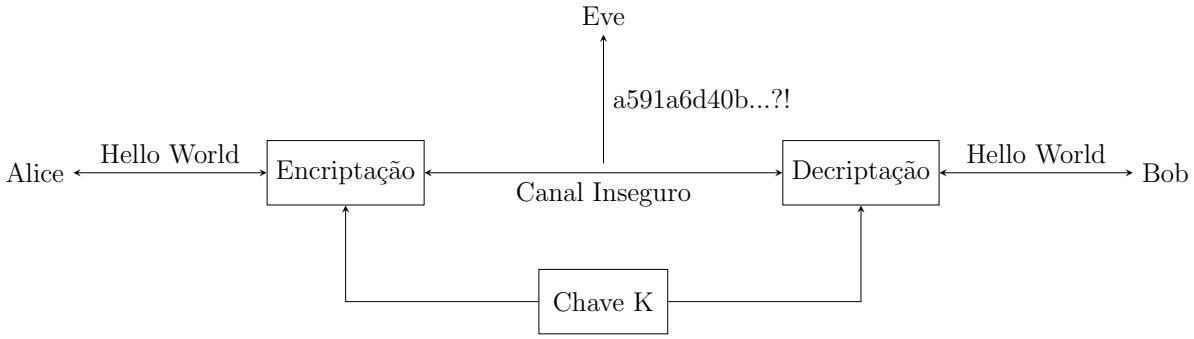


Figura 1: Alice e Bob se comunicam com a mesma chave K

A criptografia simétrica provê o serviço de confidencialidade e tem a vantagem de ser computacionalmente mais barato em relação a criptografia assimétrica, por realizar operações que são mais rápidas.[4][5]

2.4 Criptografia Assimétrica

Um dos desafios da criptografia simétrica, é o compartilhamento da chave de forma segura. Esse problema é resolvido com os pares de chaves pública e privada, também conhecido como criptografia assimétrica. No esquema de criptografia assimétrica, é gerado um par de chaves, onde cada chave pública tem uma chave privada correspondente. A chave pública é revelada publicamente e a chave privada é mantida em segredo. Quando uma é utilizada para cifrar, a outra é utilizada para decifrar. Para confidencialidade, utiliza-se a chave pública do receptor para cifrar a mensagem. Para assinatura, utiliza-se a chave privada do emissor para cifrar a mensagem.[4][5]

Na Figura 2, Alice envia para Bob a mensagem "Hello World"encriptando com a chave pública de Bob, que ao receber usa sua chave privada para decriptar a mensagem, obtendo assim o serviço de confidencialidade.

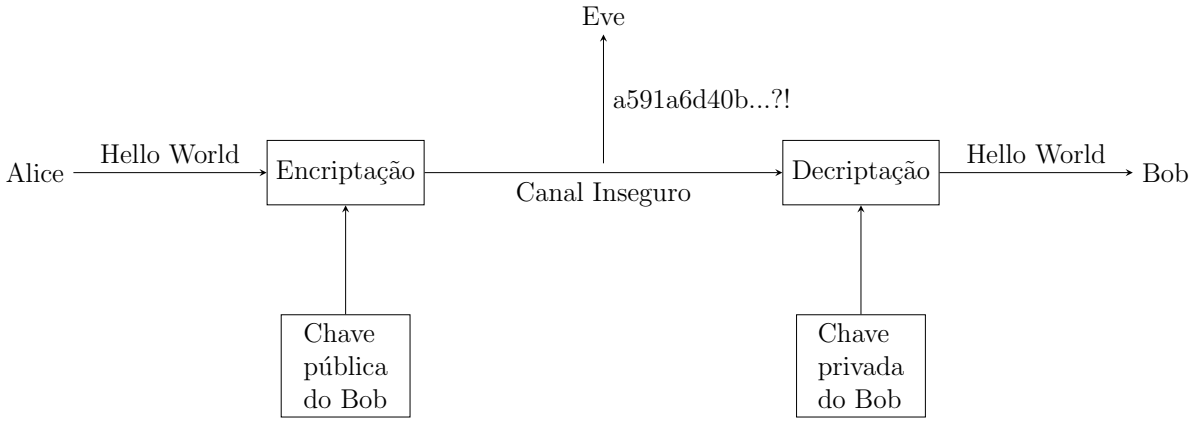


Figura 2: Alice envia mensagem para Bob

Na Figura 3, vemos o serviço de assinatura, onde Bob quer garantir que ele mandou a mensagem, então Bob cifra a mensagem com sua chave privada e Alice decifra a mensagem com a chave pública de Bob.

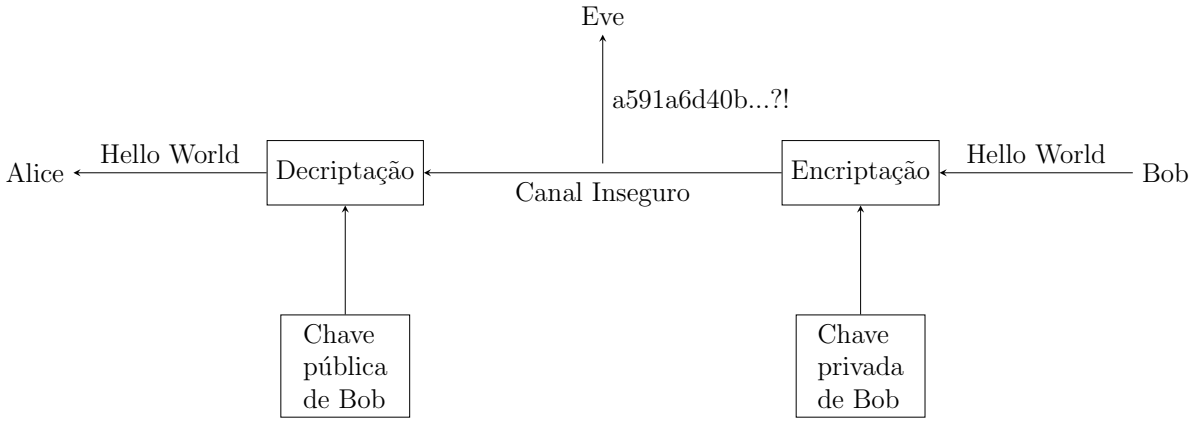


Figura 3: Bob assina a mensagem com sua chave privada.

2.5 Funções Hash

As funções de hash são funções unidirecionais, existindo assim somente uma operação possível e não reversível, o cálculo do hash.

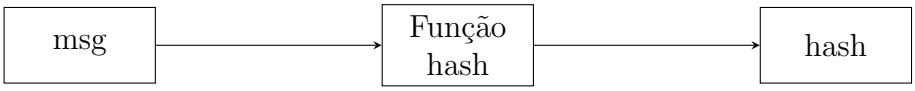


Figura 4: Diagrama de blocos de uma função hash

A mensagem é utilizada como entrada para a função de hash que então retorna o valor do hash, ou apenas o hash da mensagem.

$$h = H(msg)$$

Uma vez calculado o hash é impossível, a partir do hash, recuperar a mensagem original *msg*. Não importando o tamanho ou formato da mensagem *msg*, o hash calculado terá sempre um tamanho fixo em bits (tipicamente 128, 160, 256 ou 512 bits). Ou seja, ao utilizar uma função hash de 256 bits, uma mensagem *msg* de 320KB ou uma mensagem de 8GB, terão valores de hash com o mesmo tamanho de 256 bits.[4][5]

Toda função hash é necessário apresentar essas 3 propriedades:

- Resistência pré-imagem

  É necessário ser computacionalmente inviável reverter uma função hash, ou seja, dado um h e uma função $h = H(msg)$ será inviável encontrar o valor msg que gerou h. Isso protege de um ataque em que o atacante possui apenas o hash e está procurando o valor que gerou esse hash.

- Resistência de segunda pré-imagem

  Dada uma entrada x e um hash $H(x)$, será computacionalmente inviável encontrar um y, tal que $H(x) = H(y)$. Caso um atacante possua uma entrada e um hash, essa propriedade impede que a entrada original seja substituída.

- Resistência de colisões

Dada uma função hash, deve ser computacionalmente inviável encontrar duas entradas diferentes x e y tal que $H(x) = H(y)$. Essa propriedade garante que as colisões ainda são possíveis, mas são extremamente improváveis de ocorrer. Além disso, se uma função possui resistência de colisões, logo ela também possui resistência de segunda pré-imagem.

2.6 Assinatura Digital

Conforme foi explicado na seção anterior, as chaves publicas e privadas são inversas entre si, ou seja, se uma for usada para encriptação, a outra será usada para decriptação, e vice-versa. Se a chave privada do emissor for utilizada, o destinatário tem certeza de quem enviou a mensagem, e não foi alterado por um ataque *man in the middle*.

Para garantir a integridade de uma assinatura, pode-se utilizar uma função hash (Resumo) em conjunto com a assinatura. A função hash é utilizada para processar o documento, produzindo um pequeno pedaço de dados, chamado de hash. A função hash gera um resumo de tamanho fixo.

Para se garantir uma assinatura digital tem-se que verificar as seguintes propriedades:

- Um usuário não pode forjar a assinatura de outro usuário e as assinaturas digitais devem ser únicas para cada usuário;
- O remetente de uma mensagem não pode invalidar a assinatura de uma mensagem;
- O destinatário da mensagem não pode modificar a assinatura contida na mensagem;
- Um usuário não pode ser capaz de retirar a assinatura de uma mensagem e colocar em outra.

O esquema de Assinatura Digital calcula o hash da mensagem e depois criptografa o hash com a chave privada do emissor. A assinatura digital serve como uma garantia de que o documento é uma cópia verdadeira e correta do original, garantindo também a autoria da mensagem. As assinaturas digitais, assim como outras convencionais, podem ser forjadas, a diferença é que a assinatura digital pode ser matematicamente verificada.[4][5]

2.7 Função Esponja

Uma função esponja é um modo de operação que recebe uma sequência binária de tamanho arbitrário, e gera uma sequência binária de tamanho n definido pelo usuário:

$$\{0,1\}^* \rightarrow \{0,1\}^n$$

A função esponja é uma generalização de funções hash (tamanho fixo de saída), e cifras de fluxo (tamanho fixo de entrada). Foi criada para ser utilizado no **Keccak**, que subsequentemente foi escolhido pelo **NIST** (*US National Institute of Standards and Technology*) para se tornar o **SHA-3**.[6]

A Figura 5 ilustra a operação da função esponja do **Keccak**, onde há múltiplas permutações f operando em tamanho fixo de $r + c$ bits, em que o r é chamado de *bit rate* e c é a capacidade. A entrada é dividida em blocos de r bits, e o estado interno é inicializado com zero, em seguida procede em duas fases:

- *Absorbing*: É feito um XOR entre o estado atual e os blocos de r bits, seguido de uma permutação f. Quando todos os blocos forem processados, avança para a próxima fase.
- *Squeezing*: Os primeiros r bits do estado são retornados como blocos de saída, intercalados com aplicações da função f, o número de blocos da saída é escolhido pelo usuário.

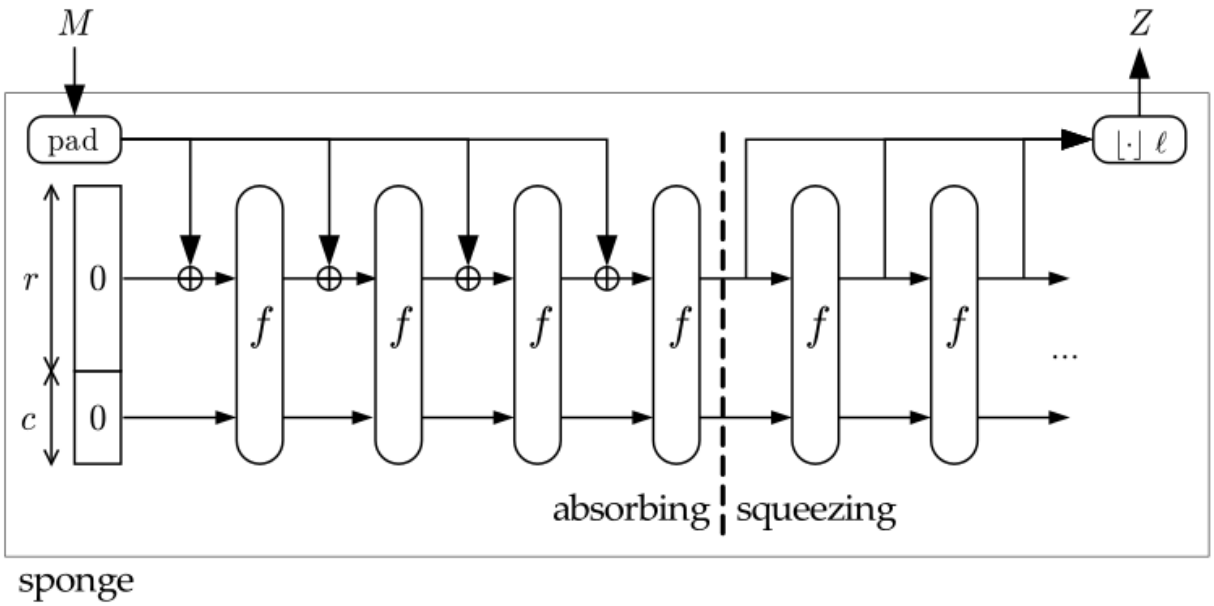


Figura 5: Função esponja do **Keccak**[6]

2.8 *Authenticated Encryption* (AE)

Authenticated Encryption é uma forma de unificar a criptografia com a assinatura digital, ou seja, uma junção da **confidencialidade** com a **autenticidade**, as Figuras 6 e 7 ilustram a diferença.

Confidencialidade é a garantia que os dados não possam ser lidos por terceiros não autorizados. No exemplo abaixo, Alice envia uma mensagem para Bob que é interceptada por Eve, porém Eve não consegue ler a mensagem.

Autenticidade é a garantia que o remetente e o destinatário se comuniquem de forma segura, o destinatário consegue verificar a identidade do remetente, e garantir que a mensagem não foi alterada em trânsito. A Figura 7 representa uma falha de autenticidade, Alice envia uma mensagem para Bob, que é interceptada por um agente malicioso Mallory, que substitui a mensagem original por outra, Bob então recebe a mensagem sabotada acreditando ser de Alice.

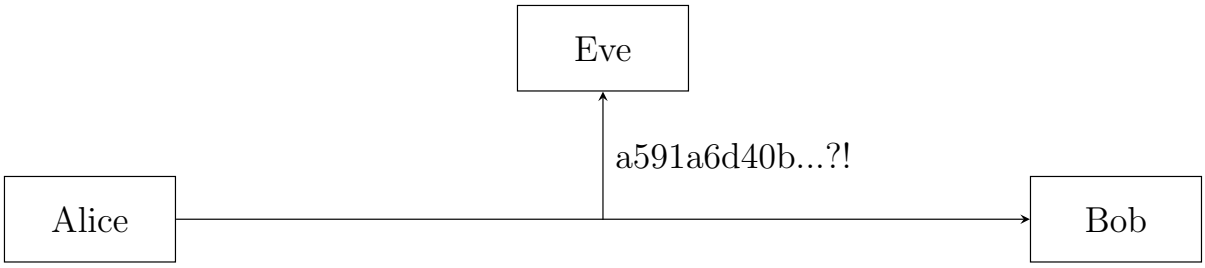


Figura 6: Confidencialidade

Figura 7: Falha de autenticidade

Algoritmos como o AES são comumente utilizados em conjunto com o *Galois/Counter Mode* (GCM) para formar o AES-GCM, garantindo a confidencialidade, integridade e autenticidade.[3]

3 Criptografia pós-quântica

Criptografia pós-quântica, do inglês *Post-Quantum Cryptography* (PQC), é o desenvolvimento de algoritmos resistentes a computadores quânticos. Se computadores quânticos forem construídos em larga escala, será possível quebrar esquemas criptográficos de chaves público-privada como o RSA, isso pode comprometer a confidencialidade e integridade de comunicações. Desde 2016, o NIST está promovendo uma competição para escolher o algoritmo para se tornar o padrão PQC, esses algoritmos devem ser capazes de promover segurança tanto para computadores clássicos e quânticos.[7]

3.1 Algoritmo de Shor

O algoritmo de Shor é um algoritmo quântico para encontrar fatores primos de números inteiros, desenvolvido em 1994 por Peter Shor. De forma bem simplificada, este algoritmo funciona buscando o período de sequências numéricas, através da aplicação de uma transformada de Fourier quântica. Considere a sequência:

$$2, 4, 8, 16, 32, 64, 128, 256, 512, 1024...$$

Ao computar $2^n \mod 21$ temos:

$$2, 4, 8, 16, 11, 1, 2, 4, 8, 16....$$

Como pode ser observado, após 6 iterações os valores se repetem, esse é o valor do período da sequência, e possível prever esse valor através da função φ de Euler. Como sabemos que N é um produto de dois números primos inteiros p e q da sequência:

$$x \mod N,\ x^2 \mod N,\ x^3 \mod N,\ x^4 \mod N....$$

Então, contanto que x não seja divisível por p ou q, então o valor do período r pode ser calculado por:

$$r = \frac{(p-1) \times (q-1)}{2}$$

No exemplo acima, os fatores primos que fatoram 21 são 3 e 7, então $r = (3-1) \times (7-1)/2 = 6$. Portanto, com o auxílio de um computador quântico, é possível obter r, e recuperar N testando valores aleatórios para x em superposição quântica, e obter o resultado com alto grau de probabilidade, pois a transformada quântica de Fourier irá filtrar os resultados improváveis.
citeshorELI5

Porém, computadores quânticos ainda estão longe de serem adotados, até 2023 o maior valor fatorado por um algoritmo de Shor foi **21**, realizado em 2012.[8]

4 Criptografia Leve

Criptografia leve é um algoritmo ou protocolo de criptografia designado para dispositivos com recursos limitados. Tais dispositivos são comumente encontrados em IoT, e possuem pouca memória, poder computacional e consumo de energia.[9]

A competição CAESAR (*Competition for Authenticated Encryption: Security, Applicability, and Robustness*) destaca-se como um marco importante nesse contexto. Iniciada em 2014, a competição visava identificar os melhores algoritmos AE, porém em 2017 a competição mudou o foco para algoritmos de criptografia leve capazes de proporcionar segurança eficaz em dispositivos com recursos restritos.[10]

Os finalistas foram: ASCON (vencedor), Elephant, GIFT-COFB, Grain-128AEAD, ISAP, PHOTON-Beetle, Romulus, SPARKLE, TinyJambu e Xoodyak, demonstrando inovação e eficiência na proteção de dados em ambientes desafiadores. Esses algoritmos foram submetidos a rigorosos testes de segurança, desempenho e implementação prática, refletindo a complexidade e a diversidade de desafios enfrentados pelos dispositivos leves na preservação da integridade das informações. O resultado da competição não apenas consolidou os esquemas escolhidos como padrões de referência, mas também impulsionou pesquisas contínuas para garantir a segurança em dispositivos com recursos restritos.[11]

5 ASCON

ASCON[12][13] é uma família de cifras de criptografia leve que foi selecionada pelo NIST para ser o padrão de criptografia leve. Essa família consiste das cifras AE: ASCON-128 e ASCON-128a, que foram os vencedores do CAESAR, e uma variante ASCON-80pq que oferece resistência *post quantum*. Além disso, também oferece quatro funções hash: ASCON-hash, Ascon-hasha, Ascon-Xof e Ascon-Xofa. O NIST recomenda o uso de ASCON-128 com ASCON-hash, ou ASCON-128a com ASCON-hasha. Todos os algoritmos citados usam a mesma permutação interna de 320 bits p^a e p^b, com número diferente de rodadas a e b.

5.1 Especificações

As seguinte funções serão utilizadas no ASCON *Authenticated Encryption*:

$$\mathcal{E}_{k,r,a,b}(K, N, A, P) = (C, T)$$

$$\mathcal{D}_{k,r,a,b}(K, N, A, C, T) \in (P, \perp)$$

$\mathcal{E}, \mathcal{D}$ são as operações de encriptação e decriptação, respectivamente. Ambas são parametrizadas por uma chave de tamanho $k \leq 160$, o tamanho do bloco r, e os valores a e b do número de *rounds* da permutação p^a e p^b, que serão discutidos adiante.

A encriptação recebe uma chave K com k bits, Nonce N com 128 bits, dados Associados A com tamanho arbitrário e um texto plano P de tamanho arbitrário. A saída consiste de um texto cifrado C e uma tag de autenticação T com 128 bits.

O procedimento de decriptação recebe a chave K, Nonce N, dados Associados A, texto cifrado C e a tag T, e produz o texto plano P se a verificação da tag estiver correta, ou erro $\perp$ caso a tag falhe.

Entre as três variantes:

- Ascon-128: $\mathcal{E}, \mathcal{D}_{128,64,12,6}$
- Ascon-128a: $\mathcal{E}, \mathcal{D}_{128,64,12,8}$
- Ascon-80pq: $\mathcal{E}, \mathcal{D}_{160,64,12,6}$

Os algoritmos de *hashing* serão definidos pela seguinte função:

$$\mathcal{X}_{h,r,a,b}(M, \ell) = H$$

A função hash $\mathcal{X}$ é parametrizada pelo tamanho limite de bits da saída h ($h = 0$ para saída sem limite de tamanho), o tamanho de bloco r e o número de rodadas a e b. $\mathcal{X}_{h,r,a,b}$ recebe a mensagem de entrada M e um tamanho de saída ℓ, e gera uma saída H do tamanho especificado $\ell \leq h$.

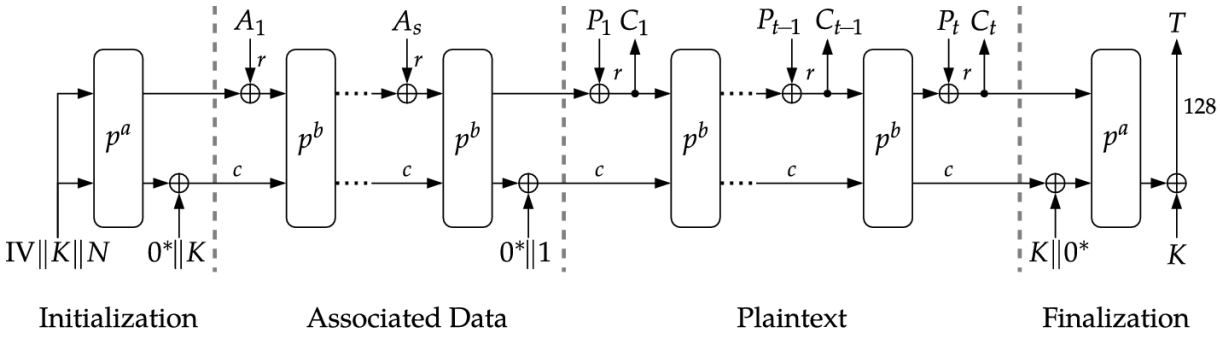
Os parametros recomendados são:

- Ascon-hash: $\mathcal{X}_{256,64,12,12}$ com $\ell = 256$.
- Ascon-hasha: $\mathcal{X}_{256,64,12,8}$ com $\ell = 256$.
- Ascon-Xof: $\mathcal{X}_{0,64,12,12}$ com ℓ arbitrário.
- Ascon-Xofa: $\mathcal{X}_{0,64,12,8}$ com ℓ arbitrário.

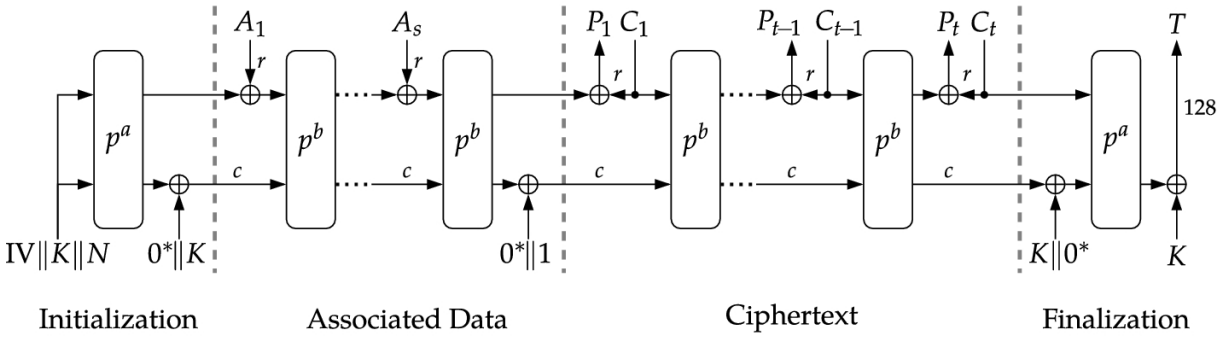
5.2 *Authenticated Encryption with Associated Data* (AEAD)

AEAD é um variante de AE, os dados associados AD não são encriptados, apenas autenticados. Isso é importante em pacotes de dados, os dados são encriptados, mas o *header* que contém as informações de endereço não podem ser encriptados. A principal vantagem do AEAD em relação aos métodos tradicionais é que a encriptação/decriptação e autenticação usam dois algoritmos nos métodos tradicionais, enquanto que em AEAD são feitas em apenas um único algoritmo, reduzindo o custo computacional, importante em dispositivos de baixo consumo como IoT.

A Figura 8 representa o diagrama de blocos do AEAD:



(a) Encryption $\mathcal{E}_{k,r,a,b}$



(b) Decryption $\mathcal{D}_{k,r,a,b}$

Figura 8: Diagrama de blocos AEAD[13]

5.2.1 Inicialização

O estado inicial S de 320 bits do Ascon recebe o valor inicial IV, concatenado com a chave K de k bits e um nonce N de 128 bits:

$$IV_{k,r,a,b} \leftarrow \begin{cases} 80400c0600000000 & \text{Ascon-128} \\ 80800c0800000000 & \text{Ascon-128a} \\ a0400c06 & \text{Ascon-128pq} \end{cases}$$

$$S \leftarrow IV_{k,r,a,b} \parallel K \parallel N$$

Na inicialização, será utilizada a permutação p^a, seguida de um XOR com a chave K:

$$S \leftarrow p^a(S) \oplus (0^{320-k} \parallel K)$$

5.2.2 Processamento de Dados Associados

Ascon divide os dados A em s blocos de r bits. Concatena 1 e faz *padding* caso necessário para obter um múltiplo de r bits. Caso A seja vazio, não é aplicado *padding* e $s = 0$:

$$A_1, ..., A_s \leftarrow \begin{cases} \text{r bits blocos de } A \parallel 1 \parallel 0^{r-1-(|A| \bmod r)} & \text{se } |A| > 0 \\ \emptyset & \text{se } |A| = 0 \end{cases}$$

Cada bloco A_i com $i = 1, ..., s$ é feito um XOR com os primeiros r bits, seguido de uma permutação p^b:

$$S \leftarrow p^b((S_r \oplus A_i) \parallel S_{320-r}), \quad 1 \leq i \leq s$$

Em seguida, é feito um XOR entre S e 1:

$$S \leftarrow S \oplus (0^{319} \parallel 1)$$

5.2.3 Processamento de Texto Plano

O texto plano P é processado em blocos de r bits. Concatena 1 e faz *padding* se necessário para completar o bloco de r bits. O resultante é dividido em t blocos de r bits, $P_1 \parallel ... \parallel P_t$:

$$P_1, ..., P_t \leftarrow r\text{-bit blocos de } P \parallel 1 \parallel 0^{r-1-(|P| \bmod r)}$$

Encriptação:

$$C_i \leftarrow S_r \oplus P_i$$
$$S = \begin{cases} p^b(C_i \parallel S_c) & \text{se } 1 \leq i < t \\ C_i \parallel S_c & \text{se } 1 \leq i = t \end{cases}$$

Decriptação:

$$P_i \leftarrow S_r \oplus C_i$$
$$S \leftarrow p^b(C_i \parallel S_c), \quad 1 \leq < t$$

No último bloco, o procedimento é diferente:

$$\tilde{P}_i \leftarrow \lfloor S_r \rfloor_\ell \oplus \tilde{C}_t$$
$$S \leftarrow (S_r \oplus (\tilde{P}_t \parallel 1 \parallel 0^{r-1-\ell})) \parallel S_c$$

5.2.4 Finalização

Na finalização, é feita uma permutação p^a do XOR entre o estado S e a chave K. A tag T consiste do XOR entre os 128-bits menos significativos do estado S com a chave K:

$$S \leftarrow p^a(S \oplus (0^r \parallel K \parallel 0^{c-k}))$$
$$T = \lceil S \rceil^{128} \oplus \lceil K \rceil^{128}$$

A encriptação retorna o tag T com o texto cifrado C, a decriptação retorna o texto plano P se o tag computado for igual ao tag recebido, caso contrário retorna erro.

5.3 hashing

O modo de operação para hashing é baseado em esponjas.[14] Tanto as funções de hash com tamanho de saída fixo quanto as funções de saída expansível com tamanho de saída variável usam internamente o mesmo algoritmo $\mathcal{X}_{h,r,a,b}$.

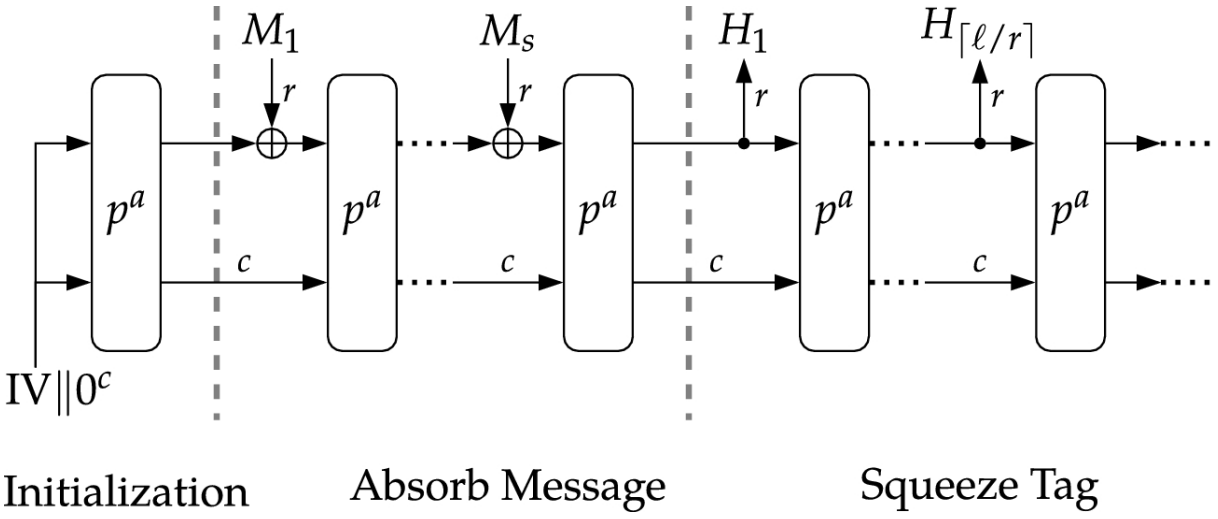


Figura 9: Diagrama de blocos hash[13]

5.3.1 Inicialização

O estado inicial de 320 bits, é definido por uma constante chamada *InitialValue*. Essa constante estabelece os parâmetros iniciais, representados por quatro números inteiros de 8 bits cada: k, r, a e o valor resultante da diferença entre a e b. A saída do algoritmo é representada por h bits, onde h é igual a ℓ, que é 256 bits, para Ascon-hash e Ascon-hasha. No caso de Ascon-Xof e Ascon-Xofa, h é configurado como 0, indicando uma saída ilimitada.

A permutação de a rodadas, denotada por p^a, é aplicada para inicializar o estado S:

$$IV_{h,r,a,b} \leftarrow 0^8 \parallel r \parallel a \parallel a-b \parallel h = \begin{cases} 00400c0000000100 & \text{Ascon-hash} \\ 00400c0400000100 & \text{Ascon-hasha} \\ 00400c0000000000 & \text{Ascon-Xof} \\ 00400c0400000000 & \text{Ascon-Xofa} \end{cases}$$

$$S \leftarrow p^a(IV_{h,r,a,b} \parallel 0^{256})$$

5.3.2 Absorbing Message

O processamento da mensagem M ocorre em blocos de r bits, e o processo de preenchimento segue a mesma abordagem utilizada para o texto plano do Ascon. Ele consiste em anexar um único '1', seguido pelo menor número de '0's necessário para que o comprimento da mensagem preenchida seja um múltiplo de r bits. A mensagem resultante, após o preenchimento, é dividida em s blocos de r bits, denotados como $M_1 \parallel ... \parallel M_s$:

$$M_1, ..., M_s \leftarrow \text{blocos de } r\text{-bit de } M \parallel 1 \parallel 0^{r-1-(|M| \mod r)}$$

Para processar cada bloco de mensagem M_i com $i = 1, .., s$, é combinado com os primeiros r bits S_r do estado S usando a operação XOR. Em seguida, é aplicada a permutação p^b em S se $i < s$

$$S \leftarrow \begin{cases} p^b((S_r \oplus M_i) \parallel S_c) & \text{se } 1 \leq i < s \\ (S_r \oplus M_i) \parallel S_c) & \text{se } 1 \leq i = s \end{cases}$$

Aqui, S representa o estado da construção esponja Ascon, S_r denota o rate de r-bit de S, e S_c representa os bits de capacidade de S. Este processo assegura que cada bloco de mensagem seja incorporado ao estado S seguindo as regras especificadas de preenchimento e processamento.

5.3.3 Squeezing

Antes de se extrair a saída hash, S é transformado pela permutação p^a:

$$S \leftarrow p^a(S)$$

Em seguida, a saída de hash é extraída do estado em blocos de r-bit representados como H_i, até que o comprimento de saída solicitado $\ell \leq h$, seja completado após $t = \lceil \ell/r \rceil$ blocos.

$$H_i \leftarrow S_r$$

Após cada extração, o bloco H_i é atribuído aos r-bits mais significativos do rate S_r e o estado interno S é transformado pela permutação de b rodadas p^b:

$$S \leftarrow p^b(S), \quad 1 \leq i \leq t = \lceil \ell/r \rceil$$

O último bloco de saída H_t é truncado para conter apenas os $\ell \mod r$ bits finais, e a saída final $H = H_1 \parallel ... \parallel \tilde{H}_t$

$$\tilde{H}_t \leftarrow \lfloor H_t \rfloor_{\ell \mod r}$$

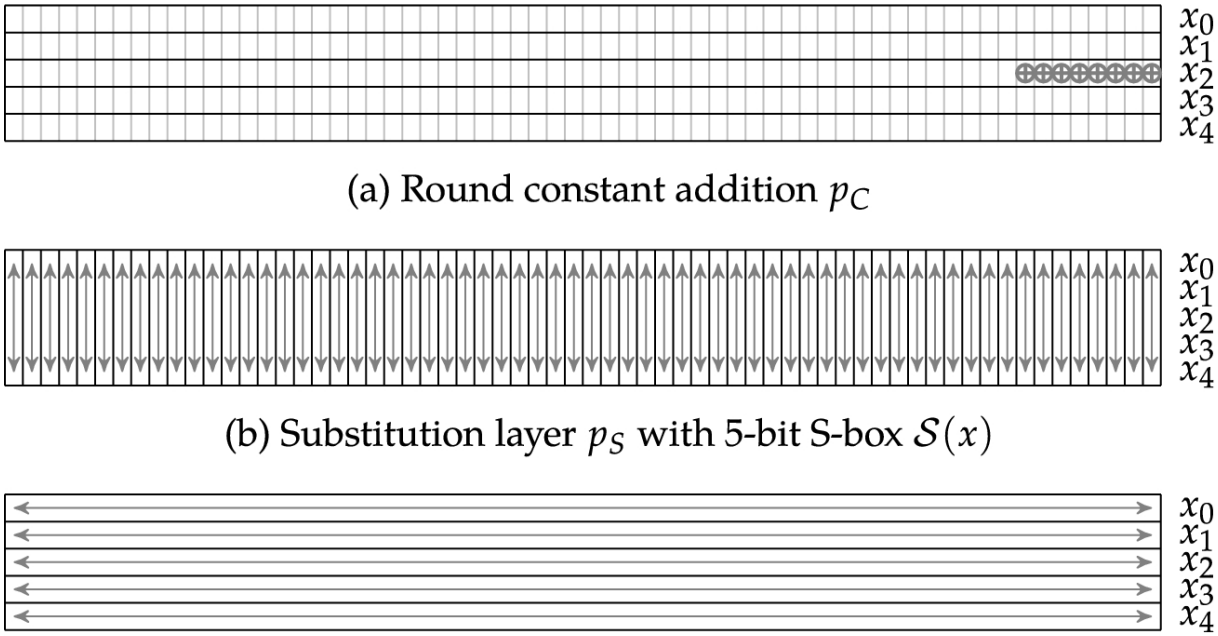
5.4 Permutação

As permutações p^a e p^b, diferem apenas no número de *rounds*, e são divididos em três etapas:

- Adição de Constantes (p_c)
- Camada de Substituição (p_s)
- Camada Difusão Linear (p_l)

O estado S de 320-bits é dividido em cinco blocos de 64-bits:

$$S = x_0 \parallel x_1 \parallel x_2 \parallel x_3 \parallel x_4$$



(a) Round constant addition p_C

(b) Substitution layer p_S with 5-bit S-box $\mathcal{S}(x)$

(c) Linear layer with 64-bit diffusion functions $\Sigma_i(x_i)$

Figura 10: Representação da etapa de permutação[13]

O pseudo-código 1 representa a etapa de permutação:

```
typedef unsigned long long int u64; // variavel 64 bits

u64 x[5]; // estado de 5 blocos x 64 bits = 320 bits
u64 constants[12] = {0xF0, 0xE1, 0xD2, 0xC3, 0xB4, 0xA5,
                     0x96, 0x87, 0x78, 0x69, 0x5A, 0x4B};

void additionOfConstants(u64 x[5], int i){
    x[2] ^= constants[i];
}

void sbox(u64 x[5]){
    u64 t[5] = {0};

    x[0] ^= x[4]; x[4] ^= x[3]; x[2] ^= x[1];
    t[0] = x[0] & (~x[4]); t[1] = x[2] & (~x[1]);
    x[0] ^= t[1]; t[1] = x[4] & (~x[3]);
    x[2] ^= t[1]; t[1] = x[1] & (~x[0]);
    x[4] ^= t[1]; t[1] = x[3] & (~x[2]);
    x[1] ^= t[1]; x[3] ^= t[0];
    x[1] ^= x[0]; x[3] ^= x[2];
    x[0] ^= x[4]; x[2] = ~x[2];
}

u64 rotateRight (u64 x, int n){
    return (x >> n) | (x << (64 - n));
}

void linearDiffusion(u64 x[5]){
    x[0] ^= rotateRight(x[0],19)^rotateRight(x[0],28);
    x[1] ^= rotateRight(x[1],61)^rotateRight(x[1],39);
    x[2] ^= rotateRight(x[2],1)^rotateRight(x[2],6);
    x[3] ^= rotateRight(x[1],10)^rotateRight(x[1],17);
```

```
33      x[4] ^= rotateRight(x[1],7)^rotateRight(x[1],41);
34  }
35
36  // nRounds determina se pA ou pB
37  void permutation((u64 x[5]), int nRounds){
38      for (int i = 0; i < nRounds; i++){
39              additionOfConstants(x, i);
40              sbox(x);
41              linearDiffusion(x);
42      }
43  }
```

Listing 1: Pseudo-código da etapa de permutação

5.4.1 Adição de Constantes

A adição de constantes seleciona o c_r da iteração i, conforme a tabela abaixo. O índice r usado é $r = i$ para p^a e $r = i + a - b$ para p^b:

p^{12}	p^8	p^6	c_r	p^{12}	p^8	p^6	c_r
0			0xF0	6	2	0	0x96
1			0xE1	7	3	1	0x87
2			0xD2	8	4	2	0x78
3			0xC3	9	5	3	0x69
4	0		0xB4	10	6	4	0x5A
5	1		0xA5	11	7	5	0x4B

Em seguida, realiza um XOR entre o valor c_r obtido pela tabela e o x_2:

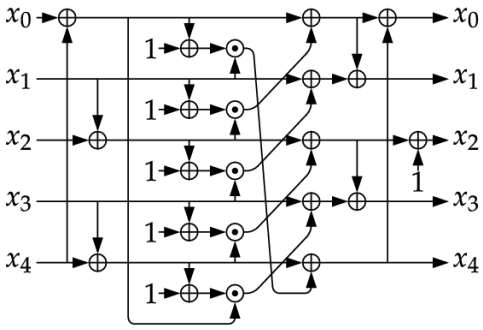
$$x_2 \leftarrow x_2 \oplus c_r$$

5.4.2 Camada de Substituição

A camada de substituição atualiza o estado S com um S-Box 5-bit:

x	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	10	11	12	13	14	15	16	17	18	19	1a	1b	1c	1d	1e	1f
$S(x)$	4	b	1f	14	1a	15	9	2	1b	5	8	12	1d	3	6	1c	1e	13	7	e	0	d	11	18	10	c	1	19	16	a	f	17

Porém, *lookup tables* são vulneráveis à um ataque *side-channel* do tipo *cache timing*, em que é possível recuperar informações do estado interno ou chave medindo o tempo de acesso ao *lookup table*. Para evitar essa vulnerabilidade, são realizadas as operações ilustradas na Figura 11a:



(a) Ascon's 5-bit S-box $\mathcal{S}(x)$

$$x_0 \leftarrow \Sigma_0(x_0) = x_0 \oplus (x_0 \ggg 19) \oplus (x_0 \ggg 28)$$
$$x_1 \leftarrow \Sigma_1(x_1) = x_1 \oplus (x_1 \ggg 61) \oplus (x_1 \ggg 39)$$
$$x_2 \leftarrow \Sigma_2(x_2) = x_2 \oplus (x_2 \ggg 1) \oplus (x_2 \ggg 6)$$
$$x_3 \leftarrow \Sigma_3(x_3) = x_3 \oplus (x_3 \ggg 10) \oplus (x_3 \ggg 17)$$
$$x_4 \leftarrow \Sigma_4(x_4) = x_4 \oplus (x_4 \ggg 7) \oplus (x_4 \ggg 41)$$

(b) Ascon's linear layer with 64-bit functions $\Sigma_i(x_i)$

Figura 11: Camada de substituição e difusão linear[13]

5.4.3 Camada de Difusão Linear

A camada de difusão linear faz uma difusão, com cada bloco de 64 bits, conforme visto na Figura 11b, através de rotações lineares para a direita.

6 Implementação

Os ASCON possui múltiplas implementações e benchmarks disponíveis no website oficial: https://ascon.iaik.tugraz.at/implementations.html, estas implementações já são otimizadas para CPU específico, e paralelizadas quando possível. Vem também incluso um benchmark, sendo apenas necessário o uso da ferramenta CMake, para compilar e executar com os seguintes comandos:

```
# compilar e testar
mkdir build && cd build
cmake ..
cmake --build .
ctest

# compilar benchmark
mkdir build && cd build
cmake .. -DALG_LIST="ascon128;asconhash" -DIMPL_LIST="opt32;opt64" -
    DTEST_LIST="getcycles"
cmake --build .

# obter performance de ciclos para CPU 32 bits
./getcycles_crypto_aead_ascon128v12_opt32
./getcycles_crypto_hash_asconhashv12_opt32

# obter performance de ciclos para CPU 64 bits
./getcycles_crypto_aead_ascon128v12_opt64
./getcycles_crypto_hash_asconhashv12_opt64

# obter tamanho da implementacao
size -t libcrypto_aead_ascon128v12_opt32.a
size -t libcrypto_hash_asconhashv12_opt32.a
size -t libcrypto_aead_ascon128v12_opt64.a
size -t libcrypto_hash_asconhashv12_opt64.a
```

Listing 2: Comandos CMake para compilação e execução

6.1 Benchmarks

Nesta seção, apresentamos os resultados dos testes experimentais com a implementação do algoritmo ASCON. Foram utilizados dois computadores: um *desktop* (AMD® Ryzen™ 5600X @3.7GHz + 16GB DDR4 2133MHz) e um *laptop* (Intel® Core™ i7-8550U @1.8GHz + 8GB DDR4 2133MHz), ambos com o sistema operacional *Ubuntu 22.04.3 LTS*. Também foram realizados testes adicionais no *desktop* com o sistema operacional *Windows* 11 e *Manjaro KDE 23.0.4* para verificar se o sistema operacional utilizado interfere na performance.

Para obter resultados confiáveis, é necessário desabilitar *overclock* automático, a frequência de base pode ser determinada pelo seus respectivos websites ou com o comando lscpu. Após determinar a frequência de base, a frequência pode ser manualmente definida com o comando cpufreq-set ou pela BIOS/UEFI da placa mãe.

Nas Tabelas 1 e 2, apresentamos os *benchmarks* de performance de ciclos por byte, disponíveis no site do ASCON:

Comprimento da mensagem em bytes	1	8	16	32	64	1536	2048
AMD EPYC 7742					8.1	6.6	6.5
AMD Ryzen 9 5950X					11.0	8.2	8.1
Apple M1 (ARMv8)					12.5	9.5	9.3
Cortex-A72 (ARMv8)					13.8	10.7	10.5
Intel Xeon E5-2609 v4					14.9	10.8	10.6
Intel Core i5-6300U	367	58	35	23	17.6	11.9	11.4
Intel Core i5-4200U	521	81	49	32	23.9	16.2	15.8
Cortex-A9 (ARMv7)					51.7	34.1	33.3
Cortex-A7 (NEON)	2182	249	148	97	71.7	47.5	46.5
Cortex-A7 (ARMv7)					69.6	52.0	51.6
ARM1176JZF-S (ARMv6)	1921	277	167	112	83.7	57.2	56.8

Tabela 1: Benchmark para referência Ascon-128 e Ascon-80pq

Comprimento da mensagem em bytes	1	8	16	32	64	1536	2048
AMD EPYC 7742					21.1	13.3	12.4
AMD Ryzen 9 5950X					24.1	16.1	15.8
Apple M1 (ARMv8)					29.2	19.6	18.5
Cortex-A72 (ARMv8)					30.5	20.5	20.0
Intel Xeon E5-2609 v4					31.9	21.4	21.2
Intel Core i5-6300U	747	114	69	46	34.2	23.2	23.1
Intel Core i5-4200U	998	153	92	61	45.5	30.9	30.7
Cortex-A9 (ARMv7)					95.8	55.5	53.9
Cortex-A7 (ARMv7)					138.1	89.9	88.8
ARM1176JZF-S (ARMv6)	3051	462	277	184	137.3	92.6	92.2

Tabela 2: Benchmark para referência Ascon-hash e Ascon-Xof

As Tabelas 3 e 4 representam a performance de ciclos por byte obtida experimentalmente:

Comprimento da mensagem em bytes	1	8	16	32	64	1536	2048
AMD Ryzen 5 5600X (Ubuntu)	318	45	27	18	13.8	9.5	9.4
AMD Ryzen 5 5600X (Windows)	309	45	27	18	13.7	9.4	9.3
AMD Ryzen 5 5600X (Manjaro)	309	45	27	18	13.7	9.4	9.3
Intel Core i7-8550U	378	54	31	20	14.3	9.7	9.3

Tabela 3: Resultados obtidos Ascon-128

Comprimento da mensagem em bytes	1	8	16	32	64	1536	2048
AMD Ryzen 5 5600X (Ubuntu)	615	91	55	36	27	18.2	18.1
AMD Ryzen 5 5600X (Windows)	611	92	55	36	27.1	18.3	18.2
AMD Ryzen 5 5600X (Manjaro)	618	92	55	36	27.1	18.3	18.2
Intel Core i7-8550U	673	98	57	36	27.4	18.7	17.9

Tabela 4: Resultados obtidos Ascon-hash

A partir destas tabelas, geramos os gráficos 12 e 13, que incluem a performance em 2048, 1536 e 64 bytes para os algoritmos AEAD e Hash, respectivamente:

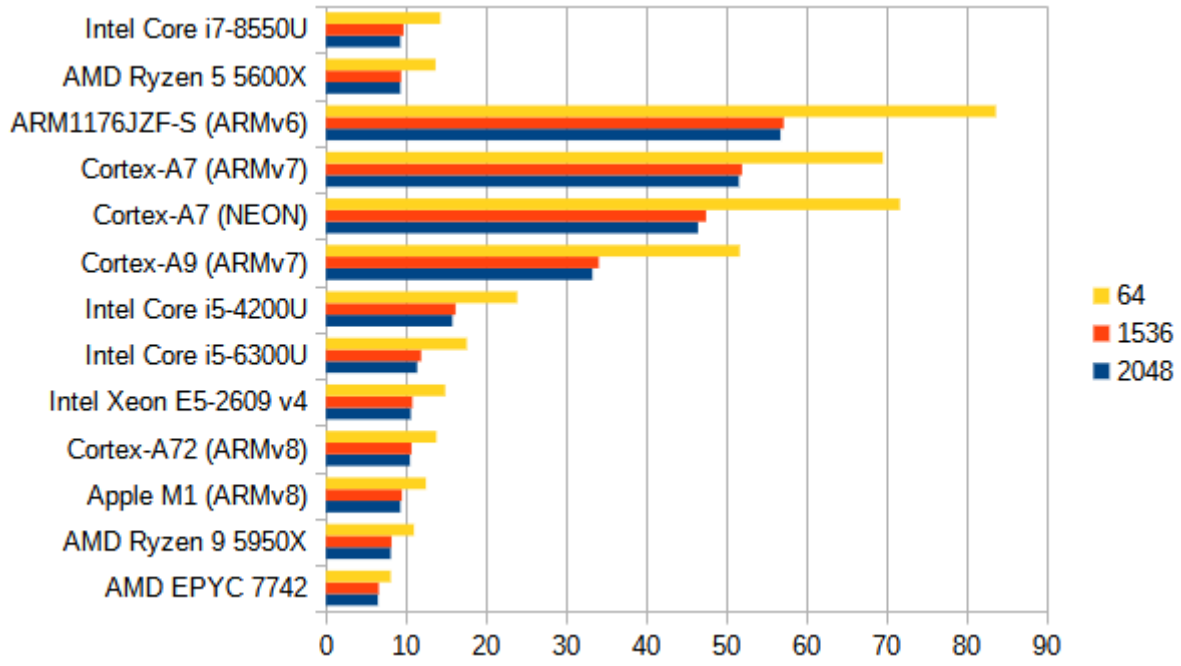


Figura 12: Gráficos para comparação AEAD

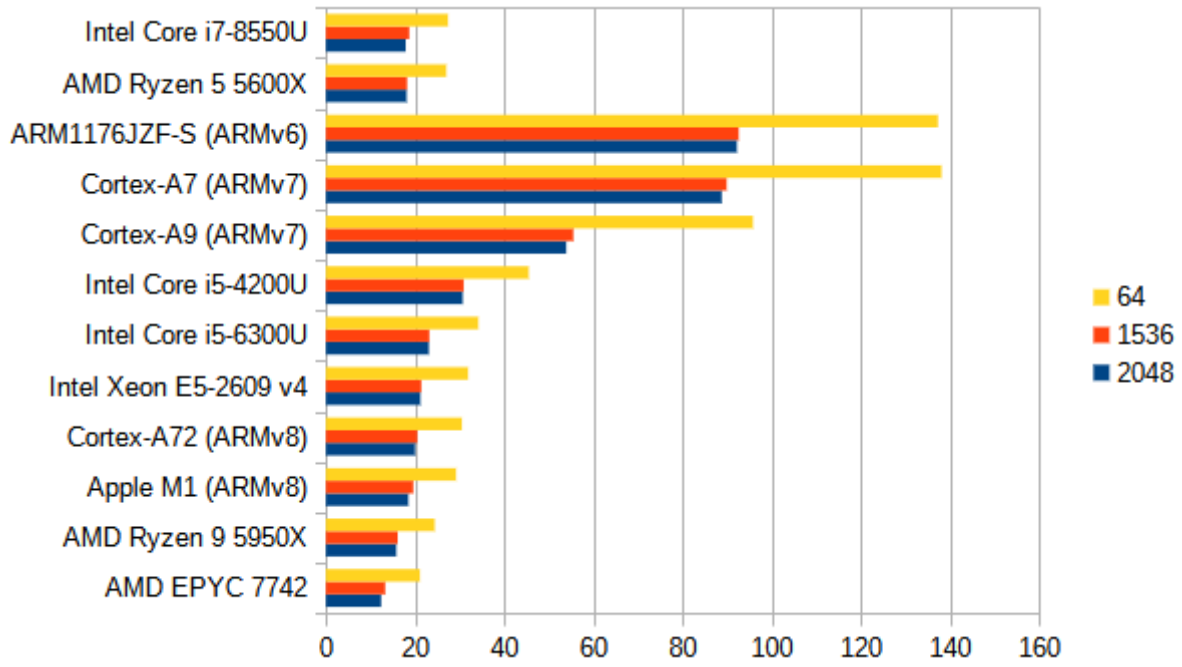


Figura 13: Gráficos para comparação Hash

Para garantir a consistência desses resultados, foram executados os seguintes comandos durante 10 segundos:

```
# aead
while true; do ./getcycles_crypto_aead_ascon128v12_opt64 $factor; done

# hash
while true; do ./getcycles_crypto_hash_asconhashv12_opt64 $factor; done
```

Porém as variações dos resultados eram insignificantes ou nulos, muitas vezes retornando o mesmo resultado inúmeras vezes.

Conforme pode ser observado pelas tabelas, o sistema operacional possui pouca influência na performance, pois os resultados são similares ao comparar a performance dos três sistemas operacionais.

Ao comparar os resultados obtidos com o dados providos para referência, pode se observar que os resultados obtidos são próximos de esperado, pois estão dentro de uma faixa de 20% do CPU equivalente mais próximo. O AMD Ryzen 5 5600X é da mesma geração do AMD Ryzen 9 5950X, que possuem 6 cores e 16 cores, respectivamente, porém os resultados apresentam uma diferença de aproximadamente 15%. Em contrapartida, o Intel Core i7-8550U não possui um CPU na referência da mesma geração, portanto o mais similar para comparação seria o i5-6300U. Novamente, os resultados são suficientemente similares, com uma diferença de aproximadamente 19%, o que é possível concluir que os resultados são consistentes com o esperado.

O teste foi refeito no *desktop*, com a memória RAM em *overclock* para 3600MHz, porém não foram observadas variações na performance de ciclos por byte. Este resultado é esperado, pois algoritmos designados para dispositivos limitados devem requerer pouco uso de memória.

7 Conclusão

A utilização de criptografia leve apresenta-se como um elemento de extrema importância no cenário comtemporâneo, onde a segurança da informação é vital. Este trabalho buscou contextualizar a relevância desse tema, destacando a crescente demanda por soluções eficientes que garantam a confidencialidade e integridade dos dados em ambientes computacionais. A compreensão da importância da criptografia leve revela-se crucial diante dos desafios encontrados no contexto digital, sendo um instrumento essencial para proteger informações sensíveis e promover a confiança em transações online.

O trabalho teve como base o documento do Ascon submetido ao NIST, porém os conceitos de criptografia para compreendê-lo são relativamentes recentes. *Authenticated Encryption* e funções esponja possuem menos de 15 anos e não são estudadas durante a graduação, sendo necessário revisar as bases para entender seu funcionamento. Muitos desses estudos estão documentados em *papers* publicados em jornais científicos, que possuem alto grau de complexidade para compreendê-los, e não possuem tradução. A implementação se deu a partir do código disponibilizado pelo website oficial do Ascon em linguagem C, vale notar que essa implementação é completamente diferente do pseudocódigo descrito na etapa de permutação, uma vez que já vem otimizado e paralelizado.

O desempenho medido nas máquinas foi consistente com o esperado, os resultados variavam apenas com o CPU selecionado, outros fatores como memória e sistema operacional são insignificantes no impacto da performance. Isso indica que o Ascon é simples, leve e flexível em *hardware*, pois usa apenas operações de lógica binária (and, xor, not e rot), permitindo ser utilizada em dispositivos com recursos limitados frequentemente vistos em IoT. Adicionalmente, é possível ser implementado em *hardware* similar ao AES, aumentando sua performance.

A partir da análise conduzida, identificaram-se potenciais direções para trabalhos futuros. Uma dessas direções envolve a otimização dos códigos, especialmente voltada para dispositivos com recursos limitados. Isso implica na definição de critérios para a seleção de algoritmos de cifras leves, passíveis de otimização. Além disso, há a proposta de implementar tais algoritmos em ambientes com restrição de recursos, permitindo uma análise minuciosa de sua execução e resultados em simulações. Essa análise abrangeria aspectos como desempenho, segurança, consumo de memória, processamento e energia.

Referências

[1] *O que é criptografia de dados? Definição e explicação*. Acesso em 10 de outubro de 2023. URL: https://www.kaspersky.com.br/resource-center/definitions/encryption.

[2] José dos Santos Machado et al. *Um Estudo dos Algoritmos de Criptografia Leve para Dispositivos IoT*. Último acesso em 11 de outubro de 2023. 2021. URL: https://periodicos.ifs.edu.br/periodicos/REC/article/view/462/744.

[3] Andrea Corbellini. *Authenticated encryption: why you need it and how it works*. Acesso em 30 de outubro de 2023. 2023. URL: https://andrea.corbellini.name/2023/03/09/authenticated-encryption/.

[4] Mihir Bellare e Phillip Rogaway. *Introduction to Modern Cryptography*. 2005, pp. 93–131, 139–152, 211–236, 237–256.

[5] N. Ferguson, T. Kohno e B. Schneier. *Cryptography Engineering: Design Principles and Practical Applications*. John Wiley Consumer, 2010.

[6] Guido Bertoni et al. *Cryptographic Sponge Functions*. Acesso em 20 de Outubro de 2023. 2011. URL: https://keccak.team/files/CSF-0.1.pdf.

[7] *Post-Quantum Cryptography PQC*. Acesso em 24 de novembro de 2024. 2017. URL: https://csrc.nist.gov/projects/post-quantum-cryptography.

[8] Enrique Martín-Lopez et al. "Experimental realization of Shor's quantum factoring algorithm using qubit recycling". Em: *Nature Photonics* (2012). URL: 10.1038/nphoton.2012.259.

[9] Futurex. *How Will Lightweight Cryptography Impact You?* Acesso em 8 de setembro de 2023. 2019. URL: https://www.futurex.com/blog/how-will-lightweight-cryptography-impact-you/.

[10] Kerry A. Mckay et al. *NISTIR 8114 Report on Lightweight Cryptography*. Acesso em 22 de outubro de 2023. 2017. URL: https://doi.org/10.6028/NIST.IR.8114.

[11] *Lightweight Cryptography Finalists*. Acesso em 28 de outubro de 2023. 2017. URL: https://csrc.nist.gov/projects/lightweight-cryptography/finalists.

[12] Christoph Dobraunig et al. *Ascon v1.2. Submission to NIST*. Último acesso em 24 de novembro de 2023. 2021. URL: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/ascon-spec-final.pdf.

[13] Christoph Dobraunig et al. *Ascon v1.2: Lightweight Authenticated Encryption and Hashing*. Último acesso em 24 de novembro de 2023. 2021. URL: https://link.springer.com/article/10.1007/s00145-021-09398-9.

[14] Guido Bertoni et al. *Sponge Functions*. Acesso em 21 de Outubro de 2023. 2007. URL: https://keccak.team/files/SpongeFunctions.pdf.