UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE TECNOLOGIA E GEOCIÊNCIAS
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

LAÍS MARIA RODRIGUES DE ARAÚJO

BLOCOS DE PERMUTAÇÃO PARA CONSTRUÇÕES ESPONJA:

explorando o uso da transformada do cosseno sobre corpos finitos de característica 2

Recife

2024

LAÍS MARIA RODRIGUES DE ARAÚJO

BLOCOS DE PERMUTAÇÃO PARA CONSTRUÇÕES ESPONJA:

explorando o uso da transformada do cosseno sobre corpos finitos de característica 2

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Pernambuco como requisito parcial para obtenção do título de Mestre em Engenharia Elétrica.

Área de Concentração: Comunicações.

Orientador: Prof. Dr. José Rodrigues de Oliveira Neto.

Coorientador: Prof. Dr. Juliano Bandeira Lima.

Recife

2024

.Catalogação de Publicação na Fonte. UFPE - Biblioteca Central

Araújo, Laís Maria Rodrigues de.
Blocos de permutação para construções esponja: explorando o uso da transformada do cosseno sobre corpos finitos de característica 2 / Laís Maria Rodrigues de Araújo. - Recife, 2024.
86p.: il.

Dissertação (Mestrado) - Universidade Federal de Pernambuco, CTG, Programa de Pós-Graduação em Engenharia Elétrica.
Orientação: José Rodrigues de Oliveira Neto.
Coorientação: Juliano Bandeira Lima.
Inclui Referências.

1. Primitiva criptográfica; 2. Construção esponja; 3. Polinômio de permutação; 4. Transformada do cosseno; 5. Corpos Finitos de Característica 2. I. Oliveira Neto, José Rodrigues de - Orientador. II. Lima, Juliano Bandeira - Coorientador. III. Título.

UFPE-Biblioteca Central CDD 621.3

Dedico esse trabalho à minha mãe, Ana Maria.

AGRADECIMENTOS

Primeiramente, agradeço à minha mãe, Ana Maria, pelo apoio incondicional e por acreditar na minha capacidade. Também agradeço ao meu irmão, Amaro Neto, que me ajudou com tudo o que precisei.

Ao meu namorado, Kelvin Bernardo, sou eternamente grata pelo companheirismo, carinho e extrema paciência, além de ter me ajudado bastante com seu conhecimento. Sua presença ao meu lado tornou essa jornada mais leve e me motivou a continuar mesmo nos momentos de dificuldade.

Aos meus amigos, Cecília e Diego, por todo incentivo e ajuda em momentos difíceis. Em especial, agradeço à Cecília, que me deu forças para dar o primeiro passo no mestrado. A amizade de vocês foi um pilar importante para minha motivação ao longo deste trabalho.

Agradeço ao meu orientador, Professor José Rodrigues, e ao coorientador, Professor Juliano Bandeira, pela orientação, paciência e ensinamentos essenciais para o desenvolvimento deste trabalho. Sou profundamente grata por todas as oportunidades de aprendizado que me proporcionaram.

Agradeço também ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) pelo suporte financeiro a este trabalho.

Por fim, agradeço a todos que contribuíram de forma direta ou indireta, ajudando-me a chegar até aqui. Muito obrigada!

RESUMO

Neste trabalho, é abordada a relevância das primitivas criptográficas na construção de sistemas de segurança e comunicação confiáveis. São investigadas algumas primitivas, incluindo algoritmos de cifragem e decifragem, funções *hash* e geração de números pseudoaleatórios que utilizam a construção esponja em seu projeto. É dada ênfase à flexibilidade e adaptabilidade da construção esponja, devido à sua capacidade de ajustar o tamanho do estado interno para atender aos requisitos de segurança. Neste sentido, são estudados algoritmos que utilizam essa primitiva, tais como as funções *hash* SHA3, Quark, Photon, SPONGENT e o ASCON. Este último é considerado como cifra de bloco, como função *hash* e para geração de números pseudoaleatórios. Também é avaliado, nesse trabalho, o uso de estruturas algébricas finitas em algoritmos de premissas criptográficas. Mais especificamente, são estudados polinômios de permutação e a transformada do cosseno sobre corpos finitos de característica 2. Diante disso, é proposto um novo bloco de permutação de comprimento $(2+n) \times 32$, em que n é um número natural não-nulo, para utilização em construções esponja. Este bloco de permutação emprega um polinômio de permutação juntamente com a transformada do cosseno do tipo 1, ambos definidos sobre o corpo finito $\mathbb{F}_{2^8}$. Quando utilizado nos algoritmos *hash* mencionados anteriormente, o bloco proposto torna possível reduzir o número de rodadas de aplicação do bloco de permutação para dois, ao invés das 12 rodadas do ASCON-*hash*, 24 do SHA3 e 1024 do S-Quark. Além disso, foi verificado o uso do bloco proposto no algoritmo ASCON em sua forma de cifra de bloco e gerador de números pseudoaleatórios. Para validação de todos os sistemas implementados, foram realizados testes de entropia, efeito avalanche, testes estatísticos da suíte de testes do Instituto Nacional de Padrões e Tecnologia (NIST) e teste de resistência a ataques de força bruta. A avaliação dos testes realizados indica que o uso do bloco proposto manteve resultados semelhantes aos gerados pelos algoritmos originais.

Palavras-chave: Primitiva criptográfica. Construção esponja. Polinômio de permutação. Transformada do cosseno. Corpos Finitos de Característica 2.

ABSTRACT

In this work, we address the relevance of cryptographic primitives in constructing reliable security and communication systems. Some primitives are investigated, including encryption and decryption algorithms, hash functions, and the generation of pseudorandom numbers that use the sponge construction in their design. We emphasize the flexibility and adaptability of sponge construction due to its ability to adjust the size of the internal state to meet security requirements. In this sense, algorithms that use this primitive are studied, such as the hash functions SHA3, Quark, Photon, SPONGENT, and ASCON. The latter is a block cipher, a hash function, and a pseudorandom number generator. This work also evaluates using finite algebraic structures in cryptographic premise algorithms. More specifically, permutation polynomials and the cosine transform over finite fields of characteristic two are studied. Therefore, we propose a new permutation block of length $(2 + n) \times 32$, where n is a non-zero natural number, for use in sponge constructions. This permutation block employs a permutation polynomial and the type 1 cosine transform, defined over the finite field $\mathbb{F}_{2^8}$. When used in the previously mentioned hash algorithms, the proposed block makes it possible to reduce the number of permutation block application rounds to two instead of the 12 rounds of ASCON-hash, 24 of SHA3, and 1024 of S-Quark. Furthermore, we verified using the proposed block in the ASCON algorithm as a block cipher and pseudorandom number generator. We conduct entropy tests, avalanche effects, statistical tests from the National Institute of Standards and Technology (NIST) test suite, and resistance tests to brute force attacks to validate all implemented systems. The evaluation of the tests carried out indicates that using the proposed block maintained results similar to those generated by the original algorithms.

Keywords: Cryptographic primitive. Sponge construction. Permutation polynomial. Cosine transform. Finite fields of characteristic 2.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AEAD	*authenticated encryption with associated data* (criptografia autenticada com dados associados)
AES	*advanced encryption standard* (padrão de criptografia avançada)
DCT	*discrete cosine transform* (transformada discreta do cossen)
DCFSA	*deterministic chaotic finite state automata* (estado finito caótico determinístico autômato)
DNA	*deoxyribonucleic acid* (ácido desoxirribonucleico)
DPA	*differential power analysis* (ataques de análise de potência diferencial)
FFCT	*finite field cosine transform* (transformada do cosseno de corpo finito)
FFDCT	*finite field discrete cosine transform* (transformada discreta do cosseno sobre corpos finitos)
IoT	*internt of things* (internet das coisas)
LFSR	*linear-feedback shift register* (registrador de deslocamento com realimentação linear)
MAC	*message authentication code* (código de autenticação de mensagem)
NFSR	*nonlinear-feedback shift register* (registradores de deslocamento com realimentação não-linear)
NIST	*national institute of standards and technology* (instituto nacional de padrões e tecnologia)
PRBG	*pseudo-random-bit generator* (gerador de bits pseudo-aleatórios)
PRF	*pseudorandom function* (função pseudoaleatória)
PRNG	*pseudorandom number generator* (gerador de número pseudoaleatório)
RAM	*random access memory* (memória de acesso aleatório)
RFID	*radio frequency identification* (tags de identificação de rádio frequência)
SPN	*substitution–permutation network* (rede de substituição-permutação)
XOF	*extendable-output function* (funções de saída extensível)

LISTA DE SÍMBOLOS

AND	Operação lógica E, denotado por $\cdot$
$\mathbb{C}$	Conjunto dos números complexos
$\mathbb{F}_q$	Corpo finito com q elementos
$\mathbb{F}_q^*$	Conjunto de todos os elementos não nulos de $\mathbb{F}_q$.
$\mathbb{I}_q$	Conjunto de inteiros Gaussianos sobre $\mathbb{F}_q$
$(\text{mod}\,)$	Função que retorna o módulo, ou seja, o resto de uma divisão inteira
$\mathbb{N}$	Conjunto dos números naturais
$\mathbb{N}^*$	Conjunto dos números naturais excluindo o zero
O(.)	Notação Grande-O
$P(.)$	Função de Permutação utilizada nas funções *hash* ou de cifragem
$\mathbb{Q}$	Conjunto dos números racionais
$\mathbb{R}$	Conjunto dos números reais
XOR	Operação lógica Ou Exclusivo, denotada por $\oplus$
$\mathbb{Z}$	Conjunto dos números inteiros
Z_d	Coeficientes da matriz A_b
$x\|\|y$	Concatenação entre os *strings* de bits x e y
$\lceil x \rceil$	Função teto, retorna o menor número inteiro que seja maior ou igual a x.
$x \ggg i$	Deslocamento circular para a direita por i bits

SUMÁRIO

1 INTRODUÇÃO

Primitivas criptográficas são blocos fundamentais na construção de sistemas criptográficos. A partir dessas primitivas, surgem diversos sistemas de segurança e comunicação segura (EASTTOM, 2015). Alguns exemplos notáveis incluem algoritmos de cifragem e decifragem (ABOUSHOSHA et al., 2020; DOBRAUNIG et al., 2021b; GUO; LI; LIU, 2021; CHENG; GUO; HE, 2022), funções *hash* (ALAWIDA et al., 2020; MAETOUQ; DAUD, 2020; RAJSKI et al., 2024), geradores de bits pseudo-aleatórios (PRBGs, do inglês *pseudo-random-bit generators*) (SHARMA; RANJAN; BHARTI, 2022; NGUYEN et al., 2022; AL-MHADAWI; ALBAHRANI; LAFTA, 2023) e códigos de autenticação de mensagem (MAC, do inglês *message authentication code*) (CUI et al., 2019; DOBRAUNIG et al., 2021a; YAN et al., 2022; WU et al., 2016). A combinação dessas primitivas contribui para a construção de sistemas criptográficos robustos e eficazes, proporcionando níveis mais elevados de segurança em comparação com a utilização isolada de cada uma delas. Isso se reflete nos aspectos de confidencialidade, integridade, autenticidade e não repúdio (DOULIGERIS; SERPANOS, 2007).

Ao estudar as primitivas criptográficas, é importante selecionar aquelas apropriadas para uso na criptografia. Nesse contexto, uma primitiva que tem recebido destaque é a construção esponja, revelando-se como uma estrutura versátil, capaz de se adaptar à necessidade de requisitos de segurança (BERTONI et al., 2007). Nos últimos anos, construções esponja vem sendo aplicadas em funções *hash* (ALTAWY et al., 2018; ALAWIDA et al., 2021; JIMALE et al., 2022; WINDARTA et al., 2023), como por exemplo, os algoritmos SHA-3 (DWORKIN, 2015) e Ascon (DOBRAUNIG et al., 2021b), selecionados como vencedores de competições do Instituto Nacional de Padrões e Tecnologia (NIST, do inglês *national institute of standards and technology*), para padronização (DWORKIN, 2017) e proteção da comunicação entre dispositivos com baixo poder de processamento (BOUTIN, 2023), respectivamente.

A construção esponja emprega um estado interno, que contém dados que serão processados e utilizados durante a execução do algoritmo. O tamanho deste estado é determinado pela soma da capacidade c, e da taxa de bits r, em que a capacidade representa a parte do estado interno que não absorve os dados de entrada e está ligada à segurança da função, enquanto a taxa de bits define a quantidade de bits que são atualizados a cada bloco de dados processado. Essa definição confere à construção esponja a habilidade de ajustar tanto a capacidade quanto a taxa de bits. É importante destacar que o tamanho do estado interno deve ser pelo menos duas vezes maior que a saída gerada para garantir uma segurança adequada (ESGIN; KARA, 2016; BERTONI et al., 2007). Logo, ter essa flexibilidade é fundamental para adequar a construção esponja a diferentes requisitos de segurança e desempenho. Essa abordagem ressalta a importância de utilizar um estado interno com um grande número de bits, uma vez que quanto maior o estado interno, maior a segurança contra certos tipos de ataques (ESGIN; KARA, 2016).

Estruturas algébricas desempenham um papel fundamental na definição de primitivas criptográficas, fornecendo a base matemática necessária para assegurar a segurança e a robustez dos sistemas de segurança da informação (OLADIPUPO et al., 2023; GAO et al., 2020; SHAH et al., 2022). Entre as estruturas mais notáveis, destacam-se os corpos finitos, os quais se mostram úteis em algoritmos de cifragem e decifragem. Por exemplo, cada byte de dados pode ser representado como um vetor de bits, que por sua vez pode ser interpretado como um elemento de um corpo finito, facilitando operações criptográficas. Além disso, eles também minimizam os erros de arredondamento e são aplicados à teoria dos números (DWORKIN et al., 2001; BRUCE, 1996).

Entre as ferramentas matemáticas sobre corpos finitos encontram-se a transformada discreta do cosseno sobre corpos finitos (FFDCT, do inglês *finite field discrete cosine transform*) e os polinômios de permutação. A FFDCT emprega a trigonometria sobre corpos finitos para realizar um mapeamento entre vetores com componentes sobre essas estruturas, possuindo aplicações em processamento de sinais e comunicações (CAMPELLO DE SOUZA et al., 2004). Já os polinômios de permutação, especialmente os que são involuções, ou seja, tem uma inversa composicional idêntica, introduzem complexidade durante a permutação em algoritmos de cifragem (YOUSSEF; TAVARES; HEYS, 1996; NIU et al., 2020).

Diante da complexidade e diversidade das primitivas criptográficas, esta dissertação tem como foco a utilização da construção esponja, que é adaptável a requisitos de segurança e flexível no tamanho do estado interno. Dessa forma, neste trabalho é proposto um novo bloco de permutação destinado a ser utilizado em construções esponja. Neste bloco, serão empregadas a FFDCT e polinômios de permutação. A partir desta abordagem, é possível reduzir o número de rodadas de permutação nos algoritmos testados para apenas 2.

1.1 OBJETIVOS

Objetivo geral:

O objetivo geral deste trabalho é a proposição de um bloco de permutação que empregue em seu projeto a FFDCT e polinômios de permutação, integrando-o na estrutura da construção esponja, visando avaliar a segurança do novo bloco proposto e o desempenho dos testes realizados em relação à aleatoriedade.

Objetivos específicos:

Os objetivos específicos desta proposta são:

1. Estudar blocos de permutação utilizados em construções esponja encontradas em algoritmos propostos na literatura;
2. Estudar as estruturas algébricas FFDCT e polinômios de permutação;

3. Aplicar as estruturas algébricas FFDCT e polinômios de permutação para propor um bloco de permutação para construções esponja;
4. Desenvolver e integrar o código do bloco de permutação proposto dentro de algoritmos de cifragem, funções *hash* e geração de números pseudoaleatórios que utilizem construções esponja;
5. Realizar testes de segurança para validar a eficácia do bloco de permutação integrado à construção esponja.

1.2 METODOLOGIA

A fim de alcançar os objetivos mencionados na Seção 1.1, o roteiro de execução do trabalho se divide nas etapas sumarizadas a seguir.

- ETAPA 1- Na fase inicial do projeto, é realizado um estudo do estado da arte da construção esponja e estruturas algébricas. São utilizadas bases de periódicos como IEEE, ScienceDirect e Springer para a revisão bibliográfica e a análise de trabalhos relacionados.
- ETAPA 2 - Durante esta etapa, é proposto um novo bloco de permutação, utilizando polinômios de permutação e transformadas do cosseno sobre corpos finitos. É empregado o *software* Visual Studio Code para criar o algoritmo do novo bloco de permutação utilizando a linguagem Python e a biblioteca SageMath para a manipulação de corpos finitos.
- ETAPA 3 - Nesta etapa, o novo bloco de permutação é integrado à estrutura esponja de algoritmos de função *hash*. Os algoritmos selecionados são SHA3, Quark, SPONGENT e ASCON. São feitas alterações necessárias para que o bloco proposto se integre de forma adequada em cada um desses algoritmos.
- ETAPA 4 - São realizados testes de aleatoriedade e segurança do bloco de permutação, utilizando testes que medem o efeito avalanche, entropia e a resistência contra ataques de força bruta.

1.3 ESTRUTURA DO DOCUMENTO E CONTRIBUIÇÕES

O restante do documento está estruturado da seguinte forma:

- No Capítulo 2, é fornecida uma revisão de conceitos matemáticos essenciais para compreender as aplicações da pesquisa, incluindo álgebra abstrata e polinômios de permutação.
- No Capítulo 3, é introduzida a construção esponja, com ênfase em sua definição e aplicações. Também será fornecida uma breve explicação sobre cifra, funções *hash* e algoritmos que utilizam a construção esponja em sua estrutura.

- No Capítulo 4, é proposto um novo bloco de permutação, analisando algoritmos baseados na construção esponja.
- No Capítulo 5, são apresentados os resultados da integração do novo bloco de permutação aos algoritmos selecionados. Também são realizados testes para avaliar sua aleatoriedade e segurança.
- No Capítulo 6, são revisitadas de forma sumária as principais contribuições do trabalho, elencados possíveis desdobramentos desta dissertação em trabalhos futuros.

2 ARITMÉTICA SOBRE ESTRUTURAS ALGÉBRICAS FINITAS

Neste capítulo, é feita uma revisão de conceitos matemáticos necessários para a compreensão das aplicações feitas na presente pesquisa. O Capítulo 1 do livro Masuda e Panario (2007) e os Capítulos 2 a 5 do livro de Herstein (1970) são as principais referências utilizadas na revisão sobre álgebra abstrata. Já na parte sobre polinômios de permutação, as principais referências são o Capítulo 8 do livro Mullen e Panario (2013) e Niu et al. (2020).

2.1 GRUPOS

Definição 2.1. (Grupo) Um grupo é uma estrutura algébrica que consiste em um conjunto G não vazio, juntamente com uma operação binária, geralmente denotada por $*$, que satisfaz quatro axiomas:

1. **Fechamento:** para quaisquer elementos a e b pertencentes ao grupo G, tem-se que $a * b$ também pertence a G.
2. **Associatividade:** para quaisquer elementos a, b e c pertencentes ao grupo G, tem-se que $a * (b * c) = (a * b) * c$.
3. **Identidade:** para todo elemento a pertencente ao grupo G, há uma elemento identidade, também podendo ser chamado de unidade, denominado por e, que satisfaz $a*e = e*a = a$.
4. **Inverso:** para todo elemento a pertencente ao conjunto G, existe um elemento a^{-1} pertencente à G tal que $a * a^{-1} = a^{-1} * a = e$.

Em um conjunto S, uma operação binária é uma aplicação que associa pares ordenados de elementos de S a um elemento pertencente a S. Essa operação é denotada como $\tau : S \times S \to S$, indicando que, para elementos arbitrários a e b pertencentes a S, a operação τ produz um resultado c que também pertence a S, representado como $c = \tau(a, b)$. Esse resultado pode ser interpretado como uma combinação dos elementos a e b sob τ. Utilizando uma analogia, isso pode ser expresso como $a * b = c$, em que a operação τ é responsável pela combinação e c o resultado da aplicação da operação τ em a e b. A operação τ é conhecida como projeção de $S \times S$ sobre S, pois ela projeta pares ordenados de $S \times S$ de volta para S.

É importante notar que o símbolo '$*$', comumente utilizado para indicar uma multiplicação, pode diferir da multiplicação tradicional. Ele é empregado de forma mais abrangente para denotar uma operação binária que combina dois elementos, resultando em um terceiro.

Definição 2.2. (Grupo Abeliano) Um grupo G é chamado abeliano ou comutativo quando, para quaisquer elementos a e b em G, a propriedade comutativa $a * b = b * a$ é satisfeita. Quando ela não é satisfeita, o grupo é classificado como não abeliano.

Uma característica importante de um grupo é a sua cardinalidade, denotada pelo símbolo $\#$, que corresponde à quantidade de elementos que o grupo contém, denominada como ordem de G. Quando um grupo possui um número finito de elementos, é chamado de grupo finito. Se o número de elementos for infinito, é chamado de grupo infinito. Um exemplo de grupo é o conjunto dos números inteiros $\mathbb{Z}$.

Definição 2.3. (Subgrupo) Dado um grupo G, um subconjunto não vazio H de G é considerado um subgrupo de G se, em relação à operação definida para os elementos de G, o subconjunto H também forma um grupo. E satisfaz às seguintes condições:

1. $a, b \in H$ resulta em $a * b \in H$.
2. $a \in H$ resulta em $a^{-1} \in H$.

Além disso, é importante notar que se H é um subgrupo de G e, adicionalmente, H possuir um subgrupo K, então K é também um subgrupo de G.

Definição 2.4. (Grupo Cíclico) Seja a um elemento pertencente a um grupo G. O subgrupo cíclico gerado por a, denotado por $\langle a \rangle$, é definido como $\langle a \rangle = \{a^i \mid i = 1, 2, ..., n\}$. Este subgrupo, $\langle a \rangle$, é reconhecido como um subgrupo de G. Se a é capaz de gerar todo o grupo G, ou seja, $G = \langle a \rangle$, tem-se que G é um grupo cíclico.

Dado $\mathbb{Z}_n$, um conjunto dos números inteiros $\mathbb{Z}$ módulo n, e dada $+_n$, a operação de adição definida no conjunto dos inteiros módulo n, um exemplo de grupo cíclico é o grupo aditivo dos inteiros, $\langle \mathbb{Z}_n, +_n \rangle$. Ao escolher o elemento 1 como gerador e realizar sucessivas adições módulo n, é obtido o grupo cíclico gerado por 1, isto é, $\langle 1 \rangle = \{1, 1+1, 1+1+1, .., 1+(n-1)\}$.

Antes do próximo grupo ser definido, é importante compreender o conceito de mapeamento e permutação. Uma função ou mapeamento $f : A \longrightarrow B$ é uma regra que atribui para cada elemento a em A precisamente um elemento b em B, em que o conjunto A é chamado de domínio de f e o conjunto B é chamado de contradomínio de f. Se $a \in A$ é mapeado para $b \in B$, então b é chamado de imagem de a, a é chamado de pré-imagem de b e será escrito da forma $f(a) = b$ (MENEZES, 1997). Já, uma permutação ϕ de um conjunto S é uma aplicação bijetora que mapeia S em si mesmo. Em outras palavras, para cada elemento s pertencente a S, existe um único elemento $\phi(s)$ que também pertence a S. Isso pode ser representado por $(x_1, x_2, x_3...x_n) \rightarrow (x_{i_1}, x_{i_2}, x_{i_3}...x_{i_n})$, em que x_{ik} é a imagem de x_i sob ϕ.

Definição 2.5. (Grupo de Permutação) Seja um conjunto S composto por $n!$ elementos. Um grupo de permutação S_n consiste em todas as permutações de um conjunto finito S com n elementos.

Um grupo de permutação, também conhecido como grupo simétrico, é o conjunto de todas as bijeções de S em si mesmo. Em outras palavras, é composto por todas as funções injetoras que associam cada elemento de S a um elemento distinto de S. Assim, S_n inclui todas as bijeções possíveis de S para si mesmo, sendo chamado de grupo de permutação de grau n.

Para exemplificar, o grupo de permutação S_3 representa as permutações de um conjunto com 3 elementos. Considerando o conjunto $S = \{1, 2, 3\}$, O grupo S_3 consiste em todas as bijeções possíveis de S para si mesmo. O grupo de permutação S_3 é composto por $3! = 6$ conjuntos que representam diferentes maneiras pelas quais os seus elementos podem ser rearranjados:

1. $(1, 2, 3) \rightarrow (1, 2, 3)$
2. $(1, 2, 3) \rightarrow (2, 1, 3)$
3. $(1, 2, 3) \rightarrow (1, 3, 2)$
4. $(1, 2, 3) \rightarrow (3, 2, 1)$
5. $(1, 2, 3) \rightarrow (2, 3, 1)$
6. $(1, 2, 3) \rightarrow (3, 1, 2)$

2.2 ANÉIS

Definição 2.6. (Anel) Um anel R é um conjunto não vazio munido de duas operações, a adição e a multiplicação. Para ser considerado um anel, ele deve satisfazer os seguintes axiomas:

1. R é um grupo abeliano sob a operação de adição.
2. **Fechamento:** para quaisquer elementos a e b em R, o produto $a * b$ também pertence a R.
3. **Associatividade:** para todos elementos a, b e c em R, tem-se que $a * (b * c) = (a * b) * c$.
4. **Distributividade:** para todos elementos a, b e c em R, vale $a * (b + c) = a * b + a * c$ e $(b + c) * a = b * a + c * a$.

Um anel R é denominado anel com elemento unidade se existe um elemento 1 em R tal que $a * 1 = 1 * a = a$, para todo a em R. Se a operação de multiplicação em R é comutativa, ou seja, a igualdade $a * b = b * a$ valer para todos a e b em R, então R é chamado de anel comutativo.

O elemento nulo em um anel, denotado por 0, é o elemento identidade da operação de adição, o que significa que para qualquer elemento $a \in R$, tem-se que $a + 0 = 0 + a = a$. Além disso, um anel R é chamado de anel com divisão se os elementos não nulos de R formam um grupo sob a multiplicação.

Exemplos comuns de anéis incluem os conjuntos dos números inteiros $\mathbb{Z}$, racionais $\mathbb{Q}$, reais $\mathbb{R}$ e complexos $\mathbb{C}$ junto as operações de adição e multiplicação.

Definição 2.7. (Característica de um Anel) Dado um anel R, para todo inteiro positivo a pertencente a R e um inteiro positivo n, a característica de R é o menor inteiro positivo n tal que

$$n * a = \underbrace{a + a + ... + a}_{n\ vezes} = 0. \tag{2.1}$$

Se n é positivo, diz-se que R tem característica positiva. Caso não exista tal n, diz-se que R tem característica zero. Além disso, se n é um número primo, R é denominado ter característica prima.

Os conjuntos dos números inteiros $\mathbb{Z}$, racionais $\mathbb{Q}$, reais $\mathbb{R}$ e complexos $\mathbb{C}$, mencionados anteriormente, são anéis com característica zero. Já um exemplo de característica prima é o conjunto $\mathbb{Z}_n$, em que $n \geq 2$.

2.3 CORPOS

Definição 2.8. (Corpo) Um corpo F é caracterizado como um anel com divisão comutativo que possui um elemento unidade, e cada elemento não nulo apresenta um inverso multiplicativo. No corpo F, verificam-se:

1. F é um grupo abeliano sob a operação de adição.
2. Os elementos não nulos formam um grupo abeliano sob a operação de multiplicação.
3. **Fechamento:** para quaisquer elementos a e b em F, o produto $a * b$ também pertence a F.
4. **Distributividade:** para todos elementos a, b e c em F, vale $a * (b + c) = a * b + a * c$ e $(b + c) * a = b * a + c * a$.

Como dito anteriormente, quando os elementos não nulos de um anel formam um grupo sob a operação de multiplicação, o anel é chamado de anel com divisão. Se esse anel com divisão é também comutativo, ele recebe a denominação de corpo. Em outras palavras, em um corpo, todos os elementos não nulos são inversíveis. Exemplos de corpos incluem os conjunto dos números racionais $\mathbb{Q}$, reais $\mathbb{R}$ e complexos $\mathbb{C}$, todos com um número infinito de elementos.

Um corpo K é considerado uma extensão de F se F está contido em K ou, equivalentemente, se F é um subcorpo de K. Essa relação implica que K é um espaço vetorial sobre F, e a dimensão desse espaço, chamada de grau de K sobre F, é denotada por $n = [K : F]$.

Tem-se como exemplo o conjunto dos números racionais $\mathbb{Q}$, que é um subcorpo tanto do conjunto dos reais $\mathbb{R}$ quanto dos complexos $\mathbb{C}$. Da mesma forma, o conjunto dos números reais $\mathbb{R}$ é um subcorpo dos números complexos $\mathbb{C}$.

2.3.1 Anéis de Polinômios

Dado um corpo F, o anel de polinômios, denotado por $F[x]$ é o conjunto de todos os polinômios da forma $a_0 + a_1x + ... + a_nx^n$, em que x é uma variável, n é qualquer inteiro não negativo e os coeficientes $a_0 + a_1...a_n$ pertencem ao corpo F. Assim, esse polinômio é expresso da seguinte maneira:

$$\begin{aligned} f(x) &= a_0 + a_1x + \cdots + a_{n-1}x^{n-1} + a_nx^n, \\ &= \sum^{n} a_ix^i. \end{aligned} \tag{2.2}$$

O grau de um polinômio $f(x)$ é determinado pelo maior inteiro i para o qual o coeficiente associado à potência x^i em $f(x)$ não é nulo. No caso do polinômio ser uma constante, seu grau é zero.

Os axiomas enunciados para um anel também são aplicáveis a um anel de polinômios, uma vez que é possível realizar o reconhecimento de igualdade entre dois polinômios e pela execução das operações de adição e multiplicação entre eles.

2.4 ESPAÇO VETORIAL

Definição 2.9. (Espaço Vetorial) Um espaço vetorial V é definido como um conjunto não vazio sobre um corpo F se atende aos seguintes axiomas:

1. V forma um grupo abeliano em relação à operação de adição.
2. **Distributividade à esquerda:** para quaisquer vetores v e w em V e para qualquer escalar α em F, tem-se que $\alpha * (v + w) = \alpha * v + \alpha * w$.
3. **Distributividade à direita:** para qualquer vetor v em V e para quaisquer escalares α e β em F, tem-se que $(\alpha + \beta) * v = \alpha * v + \beta * v$.
4. **Associatividade:** para quaisquer escalares α e β em F e para qualquer vetor v em V, tem-se que, $\alpha * (\beta * v) = (\alpha * \beta) * v$.
5. **Identidade:** Para qualquer vetor v em V existe um elemento identidade 1 em F com relação à multiplicação, tal que $1 * v = v$.

É interessante observar que, durante uma multiplicação, se o elemento escalar pertencente à F ou o vetor pertencente a V for zero, o resultado será um vetor nulo também pertencente ao espaço vetorial V.

Definição 2.10. (Combinação Linear) Se V é um espaço vetorial sobre o corpo F, e se $v_1, ..., v_n$ são vetores de V, então qualquer expressão da forma $\alpha_1 v_1 + ... + \alpha_n v_n$, em que α_i pertence à F, é chamada de combinação linear sobre F.

Além disso, os vetores $v_1, ..., v_n$ são considerados linearmente dependentes sobre F se existirem elementos não nulos $\alpha_1, ..., \alpha_n$ pertencentes a F tal que $\alpha_1 v_1 + \alpha_2 v_2 +, ..., \alpha_n v_n = 0$. Caso contrário, eles são ditos linearmente independentes.

Definição 2.11. (Base) Um subconjunto SV de um espaço vetorial V é denominado uma base de V se for linearmente independentes e se ele gerar V, significando que qualquer vetor em V pode ser expresso como uma combinações linear dos elementos pertencentes a SV.

A dimensão de um espaço vetorial V sobre um corpo F é determinada pelo número de elementos em qualquer base de V sobre F. Ou seja, é a quantidade mínima de vetores linearmente independentes necessários para gerar todo o espaço vetorial V. Um exemplo prático é a base padrão $(1, 0, 0)$, $(0, 1, 0)$ $(0, 0, 1)$, que possui dimensão 3. A partir desses três vetores, todo o espaço vetorial V é abrangido.

2.5 CORPOS FINITOS

Definição 2.12. (Corpo Finito) Um corpo F é finito se possuir um número finito de elementos, denotado por q. Nesse contexto, ele é referido como um corpo finito de ordem q, denotado por $\mathbb{F}_q$.

Quando um corpo finito K é expandido para formar um corpo maior F, o número total de elementos em F é dado por q^n, em que q é a ordem de K e n é o grau de extensão. Se o corpo finito F tem ordem q e característica prima p, ou seja, é um número primo, então a relação entre q e p é dada por $q = p^n$, em que n é o grau da extensão de F sobre o corpo finito $\mathbb{F}_q$. Isso significa que F é um corpo de extensão de $\mathbb{F}_q$ e tem exatamente p^n elementos.

Definição 2.13. (Elemento Primitivo) Dentro de um corpo finito $\mathbb{F}_q$, um elemento primitivo α é definido como um elemento de ordem ord$(\alpha) = q - 1$, isto é, $q - 1 = min_{i \in \mathbb{N}}\{i : \alpha^i = 1\}$. Este elemento gera o grupo multiplicativo que consiste em todos os elementos não nulos de $\mathbb{F}_q$, denotado por $\mathbb{F}_q^*$.

Seja α um elemento primitivo de $\mathbb{F}_q$, tem-se que $\mathbb{F}_q^* = \{1, \alpha, \alpha^2, ..., \alpha^{q-2}\}$ é o grupo multiplicativo gerado por esse elemento. Ao encontrar um elemento $\alpha \in F_q$ de ordem $q - 1$, todos os elementos primitivos deste corpo são identificados. Isso ocorre devido que qualquer

potência α^i, em que i e $q-1$ são coprimos (ou seja, o único número positivo que os divide é 1), está incluída no conjunto dos elementos primitivos. A quantidade de elementos primitivos de um corpo finito é determinada pela função de Euler, $\phi(q-1)$, conforme descrito em (HARDY; WRIGHT, 1979).

Como exemplo, em $\mathbb{F}_5$, as potências do elemento 3 são $3^1 = 3$, $3^2 = 4$, $3^3 = 2$, $3^4 = 1$. Isso significa que 3 gera todos os elementos do corpo finito $\mathbb{F}_5$ quando elevado à potências diferentes. Logo, 3 é um elemento primitivo de $\mathbb{F}_5$.

Em um corpo finito, cada elemento pertencente a $\mathbb{F}_{q^n}$ pode ser representado por um polinômio. Em outras palavras, cada elemento de $\mathbb{F}_{q^n}$ é um polinômio com coeficientes em $\mathbb{F}_q$. Essa associação é expressa por $\mathbb{F}_{q^n} \cong \mathbb{F}_q[x]/(f)$ em que f é um polinômio irredutível de grau n sobre $\mathbb{F}_q$. Ou seja, f é um polinômio que não pode ser fatorado em polinômios de grau menor dentro o corpo $\mathbb{F}_q$. Para exemplificar, seja um corpo finito $\mathbb{F}_2$, que consiste nos elementos {0,1} com as operações de adição e multiplicação módulo 2. Uma extensão finita $\mathbb{F}_{2^2}$ pode ser construída utilizando o polinômio irredutível $f = x^2 + x + 1$. Essa extensão $\mathbb{F}_{2^2}$ é expressa da forma $\mathbb{F}_2[x]/(x^2 + x + 1)$ e seus elementos são representados por um polinômio de no máximo grau 1, isto é, $a_0x + a_1$, em que $a_0, a_1 \in 0, 1$. Dessa forma, os elementos de $\mathbb{F}_{2^2}$ são $\{0, 1, x, x + 1\}$, os quais tem coeficientes em $\mathbb{F}_{2^2}$.

2.5.1 Polinômios de Permutação

Definição 2.14. (Polinômio de Permutação) Seja q uma potência de um primo, $\mathbb{F}_q$ um corpo finito com q elementos e $\mathbb{F}_q^*$ o conjunto de todos os elementos não nulos de $\mathbb{F}_q$. Um polinômio $f \in \mathbb{F}_q[x]$ é denominado polinômio de permutação de $\mathbb{F}_q$ se a correspondente função polinomial $f : c \to f(c)$ mapeia elementos de $\mathbb{F}_q$ em si mesmo, resultando em uma permutação desses elementos.

Lema 2.1. *(MASUDA; PANARIO, 2007) Seja F um corpo de ordem q e característica p e a um elemento pertencente a F. O polinômio $x^q - x$ é fatorado em $\mathbb{F}_q[x]$ como*

$$x^q - x = \prod_{a \in F} (x - a). \tag{2.3}$$

Essa fatoração revela que o polinômio $x^q - x$ possui, no máximo, q raízes distintas em F. Cada elemento $a \in F$ representa uma raiz de $x^q - x$, refletindo-se na expressão fatorada. É relevante ressaltar que essa fatoração é exclusiva para o corpo F e não se aplica a outros subcorpos de F.

Seja uma operação de composição de polinômios, uma operação que combina dois polinômios de permutação $f(x)$ e $g(x)$, formando um terceiro, $h(x) = f(g(x))$, com a redução do resultado módulo $x^q - x$. O conjunto de todos os polinômios de permutação em um corpo

finito $\mathbb{F}_q$ forma um grupo sob a operação de composição de polinômios módulo $x^q - x$. Esse grupo é isomorfo ao grupo de permutação S_q de ordem $q!$. Em outras palavras, isso significa que existe uma correspondência bijetiva entre os elementos dos polinômios de permutação e as permutações de q elementos, preservando as operações de composição. Para $q > 2$, o grupo de permutação S_q pode ser gerado pelo polinômio x^{q-2} e todos os polinômios lineares $a_0x + a_1$, em que a_0 e $a_1 \in \mathbb{F}_q$.

Dessa forma, afirma-se que um polinômio $f \in \mathbb{F}_q[x]$ é classificado como um polinômio de permutação se, e somente se, atende às seguintes condições (MULLEN; PANARIO, 2013):

1. o polinômio f possui exatamente uma raiz em $\mathbb{F}_q$;
2. para cada inteiro t no intervalo $1 \leq t \leq q-2$ e $t \not\equiv 0 \pmod p$, a redução de $(f(x))^t \pmod{x^q - x}$ tem no máximo grau $q - 2$.

Para um polinômio de permutação, existe um único polinômio $f^{-1}(x)$ sobre o corpo finito $\mathbb{F}_q$ chamado de inverso composicional de $f(x)$ tal que $f(f^{-1}(x)) \equiv f^{-1}(f(x))$. Quando o seu inverso composicional for ele mesmo, o polinômio de permutação é chamado de involução.

Há um crescente interesse na construção de polinômios de permutação em corpos finitos $\mathbb{F}_q$, devido às aplicações nas áreas de criptografia, teoria da codificação e combinatória. Nesse contexto, o Lema 2.2 apresenta um resultado significativo e generalizado de critérios de outros autores que ficou conhecido como o critério Akbary–Ghioca–Wang (AGW), o qual teve um impacto notável no desenvolvimento de outros resultados para construção de polinômios de permutação:

Lema 2.2. *(AKBARY; GHIOCA; WANG, 2011) Sejam A, S e $\overline{S}$ conjuntos finitos com $\#S = \#\overline{S}$. Considere os mapeamentos $f : A \to A$, $\overline{f} : S \to \overline{S}$, $\lambda : A \to S$ e $\overline{\lambda} : A \to \overline{S}$ tais que $\overline{\lambda} \circ f = \overline{f} \circ \lambda$. Se λ e $\overline{\lambda}$ são sobrejetoras, logo as seguintes afirmações são equivalentes:*

1. *f é uma bijeção (uma permutação de A) e*
2. *$\overline{f}$ é uma bijeção de S para $\overline{S}$ e f é injetora em $\lambda^{-1}(s)$ para cada $s \in S$.*

Além disso, (AKBARY; GHIOCA; WANG, 2011) trazem um critério apresentado em (ZIEVE, 2008), o qual desempenha um importante papel na construção de polinômios de permutação de $\mathbb{F}_q$. Os autores destacam que que a relevância deste critério consiste em simplificar o problema de determinar se um dado polinômio é um polinômio de permutação, transformando-o em uma questão de verificar se um outro polinômio permuta um subconjunto menor. Dados d e r, inteiros positivos, e μ_d, o conjunto de números que são raízes d-ésimas da unidade em um corpo finito $\mathbb{F}_q$, o critério é:

Lema 2.3. *(ZIEVE, 2008) Seja* $d, r > 0$ *com* $d|(q-1)$*, e seja* $h \in \mathbb{F}_q[x]$*. Logo* $f(x) := x^r h(x^{(q-1)/d})$ *permuta* $\mathbb{F}_q$ *se e somente se*

1. $\gcd(r, (q-1)/d) = 1$ *e*
2. $x^r h(x^{(q-1)/d})$ *permuta* μ_d

As raízes d-ésimas da unidade são raízes cuja ordem é um divisor de d. Essas raízes são elementos que quando elevados à potência d, resultam em 1.

A partir dos resultados apresentados em (AKBARY; GHIOCA; WANG, 2011), os autores de (NIU et al., 2020) apresentam casos específicos de classes de polinômios de permutação da forma $f = x^r h(x^s)$, que também são involuções, sobre um corpo finito $\mathbb{F}_{q^2}$. Seja γ o elemento primitivo do corpo finito $\mathbb{F}_{q^2}^*$ e μ_{q+1} um subgrupo de $\mathbb{F}_{q^2}^*$ tal que $\mu_{q+1} = \{x \in \mathbb{F}_{q^2}^* : x^{q+1} = 1\}$, uma dessas involuções é:

Corolário 2.1. *(NIU et al., 2020) Seja* m *um inteiro positivo,* $q = 2^{2m}$ *e um inteiro* $1 \leq k \leq q/2$*. Seja* $f(x) = x^{q^2-2} h(x^{q-1})$*, em que* $h(x) = \gamma(1 + \beta x^k + \beta^{-1} x^{-k})$*,* $\gamma \in \mathbb{F}_{q^2}^*$ *e* $\beta \in \mu_{q+1}$*. Então* $f(x)$ *é uma involução em* $\mathbb{F}_{q^2}$ *se e somente se* $\beta^2 \gamma^{(q-1)k} = 1$*.*

Então, a involução é dada por:

$$f(x) = x^{-1}\gamma(1 + \beta x^{kq-k} + \beta^{-1} x^{-kq+k}). \tag{2.4}$$

Ao analisar involuções, é necessário considerar a quantidade de pontos fixos que essas funções possuem, especialmente no contexto da criptografia, em que tal característica desempenha um papel crucial na segurança do algoritmo. Um ponto fixo, representado por $f(a) = a$, é um elemento que permanece inalterado sob a aplicação de uma função. Uma permutação aleatória em $\mathbb{F}_{2^q}$ tem, em média, $O(1)$ ponto fixo. Isso significa que, à medida que o tamanho da entrada q cresce, a quantidade de pontos fixos não aumenta proporcionalmente com q. Enquanto que uma involução aleatória em $\mathbb{F}_{2^q}$ apresenta $2^{q/2} + O(1)$ pontos fixos (FLAJOLET; SEDGEWICK, 2009; CHARPIN; MESNAGER; SARKAR, 2015). Essas características têm implicações práticas, uma vez que uma involução ou permutação sem pontos fixos ou com mais do que $O(1)$ pontos fixos pode ser distinguida de uma permutação aleatória, tornando-se vulnerável a ataques (CHARPIN; MESNAGER; SARKAR, 2015).

Existe uma correlação entre as propriedades criptográficas de algoritmos e a quantidade de pontos fixos, sugerindo que uma S-box deva conter poucos ou nenhum pontos fixos (YOUSSEF; TAVARES; HEYS, 1996). As involuções apresentam, em média, um alto número de pontos fixos, dessa forma, deve ser feita uma escolha adequada da seleção da involução para garantir uma maior robustez e resistência contra possíveis ataques (CHARPIN; MESNAGER; SARKAR,

2015). Em (NIU et al., 2020), é feito um estudo de pontos fixos nas involuções construídas por eles e como resultado, a involução da Equação (2.4) apresenta 2 pontos fixos.

2.6 TRANSFORMADA DO COSSENO DEFINIDA SOBRE UM CORPO FINITO DE CARACTERÍSTICA 2

A transformada discreta do cosseno (DCT, do inglês *discrete cosine transform*) é uma ferramenta matemática aplicada no processamento de sinais e imagens, introduzida em (AHMED; NATARAJAN; RAO, 1974). Uma propriedade distintiva da DCT é sua capacidade de, ao ser aplicada a um vetor de valores reais, gerar outro vetor composto exclusivamente por valores reais (BLAHUT, 2010). A DCT apresenta vários tipos, sendo a DCT do tipo 2 (DCT-II) a mais comumente empregada (GUPTA; GARG, 2012). Neste trabalho será adotada a notação de que o vetor **v**, cuja n-ésima componente é v_n, é denotado por $\mathbf{v} = (v_n)$ e sua transformada é o vetor $\mathbf{V} = (V_k)$. Assim, considerando um comprimento N, a DCT-I é definida por

$$V_k = \sum_{n=0}^{N-1} v_n \cos\left(\frac{nk\pi}{N-1}\right), \tag{2.5}$$

em que $k = 0, 2, ..., N-1$.

E a DCT-II é definida da seguinte forma

$$V_k = \sum_{n=0}^{N-1} v_n \cos\left(\frac{(2n+1)k\pi}{2N}\right), \tag{2.6}$$

em que $k = 0, 2, ..., N-1$.

A partir das Equações (2.5) e (2.6), surgiram diversas contribuições de outros autores no campo de pesquisa de processamento de imagem. Entre essas contribuições, está a dos autores de (CAMPELLO DE SOUZA et al., 2004), que introduziram a transformada discreta do cosseno em corpo finito, (FFDCT, do inglês *finite field discrete cosine transform*). A FFDCT é um transformada discreta definida para sinais sobre o corpo finito $\mathbb{F}_q$, em que $q \equiv 3 \pmod 4$, e seu espectro apresenta componentes sobre $\mathbb{F}_q$.

Para a construção dessa transformada, são utilizados inteiros gaussianos, denotados por $\mathbb{I}_q$, os quais são números da forma $a + bj$, em que $a, b \in \mathbb{F}_q$ e j é um número imaginário que satisfaz $j^2 = -1$.

Definição 2.15. (**Função k-trigonométrica do cosseno** (CAMPELLO DE SOUZA et al., 2004)) Seja ζ um elemento não negativo pertencente ao conjunto de inteiros Gaussianos, $\mathbb{I}_q$, sobre $\mathbb{F}_q$, em que $q \equiv 3 \pmod 4$. A função cosseno sobre corpo finito é definida como

$$\cos_k(n) := (2^{-1} \pmod q)(\zeta^{nk} + \zeta^{-nk}), \tag{2.7}$$

com $n, k = 0, 1, ..., N-1$, em que k é o subscrito da função k-trigonométrica e ζ tem ordem N.

Considerando que $\mathbf{v} = (v_n)$ é um vetor de comprimento N sobre $\mathbb{F}_q$, a FFDCT é definida como:

Definição 2.16. (**FFDCT** (CAMPELLO DE SOUZA et al., 2004)) Se $\zeta \in \mathbb{I}_q$ possui ordem multiplicativa $2N$, a transformada discreta do cosseno em corpo finito da sequência $\mathbf{v} = (v_n)$, em que $n = 0, 1, ..., N-1$ e $v_n \in \mathbb{F}_q$, resulta na sequência $\mathbf{V} = (V_k)$, em que $k = 0, 1, ..., N-1$ e $V_k \in \mathbb{I}_q$, composta pelos elementos

$$V_k := \sum_{n=0}^{N-1} 2v_n \cos_k \left(\frac{2n+1}{2} \right). \tag{2.8}$$

Transformadas definidas sobre corpos finitos são discretas tanto no domínio da variável quanto no domínio da transformada. Isso significa que, além de operarem em sinais ou dados discretos no domínio da variável, os coeficientes transformados também pertencem a um conjunto finito de elementos.

Posteriormente, (LIMA; BARONE; CAMPELLO DE SOUZA, 2016) introduziram as transformadas de cosseno sobre corpos finitos de característica 2, isto é, em $\mathbb{F}_{2^r}$, em que r pode ser qualquer inteiro positivo.

Definição 2.17. (**Função cosseno sobre corpo finito** (LIMA; BARONE; CAMPELLO DE SOUZA, 2016)) Seja $\zeta \in \mathbb{F}_{2^r}$ um elemento com ordem multiplicativa denotada por ord(ζ). A função cosseno sobre corpo finito relacionada à ζ é definida como

$$\cos_\zeta (n) := \zeta^n + \zeta^{-n}, \tag{2.9}$$

com $n \in \mathbb{Z}$.

A função em (2.9) possui a propriedade que todos os valores possíveis da função são obtidos para $n \in 0, 1, ..., \text{ord}(\zeta) - 1$ (LIMA; BARONE; CAMPELLO DE SOUZA, 2016).

Os autores apresentam 4 variantes da transformada de cosseno sobre corpos finitos de característica 2, denotadas por transformada do cosseno de corpo finito (FFCT, do inglês *finite field cosine transform*). No contexto deste estudo, será utilizada é a FFCT do tipo 1, denotada por FFCT-I, definina na sequência.

Definição 2.18. (**FFCT-I** (LIMA; BARONE; CAMPELLO DE SOUZA, 2016)) Seja $\zeta \in \mathbb{F}_{2^r}$ um elemento tal que ord$(\zeta) = 2N - 1$. A transformada do cosseno de corpo finito de tipo I do vetor $\mathbf{v} = (v_n), v_n \in \mathbb{F}_{2^r}, n = 1, 2, ..., N-1$ é o vetor $\mathbf{V} = (V_k), V_k \in \mathbb{F}_{2^r}$ cujos componentes são definidos como

$$V_k := \sum_{n=1}^{N-1} v_n \cos_\zeta (kn), \tag{2.10}$$

$k = 1, 2, ..., N-1$.

E sua inversa é dada por:

Definição 2.19. (FFCT-I^{-1} (LIMA; BARONE; CAMPELLO DE SOUZA, 2016)) Seja $\zeta \in \mathbb{F}_{2^r}$ um elemento tal que ord$(\zeta) = 2N - 1$. A transformada do cosseno de corpo finito inversa de tipo I do vetor $\mathbf{V} = (V_k), V_k \in \mathbb{F}_{2^r}, k = 1, 2, ..., N-1$ é o vetor $\mathbf{v} = (v_n), v_n \in \mathbb{F}_{2^r}$ cujos componentes são definidos como

$$v_n := \sum_{n=1}^{N-1} V_k \cos_\zeta (kn), \tag{2.11}$$

$n = 1, 2, ..., N-1$.

Com isso, é visto que a relação entre os vetores **v** e **V** pode ser representada como **V = Cv**, em que **C** é uma matriz de transformação de tamanho $(N-1) \times (N-1)$, cujo elemento $C_{k,n}$ localizado na k-ésima linha e na n-ésima coluna, é determinado com base no respectivo núcleo cosseno. Além disso, como a FFCT de tipo 1 é um involução, implica que a matriz **C** é igual a sua inversa, ou seja, $\mathbf{C} = \mathbf{C^{-1}}$.

Ao definir uma FFCT em um corpo de característica 2, é possível estabelecer uma relação direta entre palavras binárias de comprimento específico e elementos correspondentes no corpo de característica 2, ou seja, oferece uma forma direta de mapear dados digitais para elemento do corpo. Essa relação estabelecida pode ser utilizada para realizar operações de processamento no domínio da transformada (LIMA; SILVA; CAMPELLO DE SOUZA, 2017). Alguns exemplos de aplicação da FFCT incluem cifragem de imagem, comunicação multiusuário e códigos de correção de erros (LIMA; CAMPELLO DE SOUZA; PANARIO, 2011; LIMA; BARONE; CAMPELLO DE SOUZA, 2016; LIMA; SILVA; CAMPELLO DE SOUZA, 2017).

2.7 CONSIDERAÇÕES

No presente capítulo, foi discutida a aritmética sobre estruturas algébricas finitas, incluindo grupos, anéis e corpos. Além disso, também foram discutidas funções e transformações em corpos finitos que podem ser aplicadas em conjunto como primitivas criptográficas na construção de sistemas criptográficos.

3 CONSTRUÇÃO ESPONJA

As primitivas criptográficas são algoritmos fundamentais na construção de sistemas de segurança da informação (EASTTOM, 2015). Funções *hash* e algoritmos de cifragem são exemplos que podem ser citados, e a combinação dessas ou outras primitivas resulta em novos algoritmos criptográficos (EASTTOM, 2015; MUHAMMAD; ÖZKAYNAK, 2020; WINDARTA et al., 2023). Dentre essas primitivas, a construção esponja vem sendo bastante utilizada como bloco construtivo de outros algoritmos como funções *hash* e algoritmos de cifragem (DOBRAUNIG et al., 2021b; WINDARTA et al., 2023). Sua estrutura apresenta flexibilidade e pode ser adaptada a diferentes requisitos de segurança e aplicações. Além disso, ela mostra sua versatilidade ao poder processar mensagens de tamanhos diversos e gerar saídas com comprimentos variáveis (BERTONI et al., 2007). Este capítulo introduz a construção esponja, trazendo uma revisão bibliográfica e abordando sua definição suas e aplicações. Além disso, será dada uma breve explicação sobre cifras e funções *hash*, além de algoritmos que empregam a construção esponja em sua estrutura.

3.1 REVISÃO BIBLIOGRÁFICA SOBRE CONSTRUÇÃO ESPONJA

A crescente demanda por funções *hash* leves impulsionou o surgimento de várias opções, especialmente para dispositivos com recursos limitados (WINDARTA et al., 2022). Entre as mais destacadas estão aquelas cuja estrutura é baseada na construção esponja, como Quark, Photon, SPONGENT e ASCON (WINDARTA et al., 2022; AUMASSON et al., 2013; GUO; PEYRIN; POSCHMANN, 2011; BOGDANOV et al., 2011; DOBRAUNIG et al., 2021b).

Em (WINDARTA et al., 2022), é feita uma revisão de funções *hash* adequadas para o uso em dispositivos altamente restritos, que requerem baixo custo computacional. Com isso, são investigadas 34 funções *hash* leves, das quais 24 empregam a construção esponja como primitiva criptográfica, citada como uma das mais utilizadas. Além de análises de segurança dessas funções, é conduzida uma análise de suas implementações em hardware e software, abordando métricas como consumo de energia, consumo de memória de acesso aleatório (RAM, do inglês, *random access memory*), tamanho de código, latência, taxa de transferência e área ocupada (*gate equivalent*). No entanto, essas análises são conduzidas apenas para fins informativos e não para fins comparativos diretos.

Em (AL-SHATARI et al., 2020), é realizada a implementação de uma estrutura iterativa da função *hash* Photon em dispositivos FPGA, com o objetivo de otimizar o desempenho da versão Photon-$80/20/16$, melhorando sua relação entre área ocupada e taxa de transferência. Como resultado, observou-se uma significativa melhoria na taxa de transferência, variando de $10{,}26\%$ a $51{,}04\%$, e uma redução na área utilizada, variando de $7{,}55\%$ a $60{,}64\%$, quando

comparadas com outras implementações do Photon.

Já em (PISTONO; BELLAFQIRA; COATRIEUX, 2021), a função *hash* Quark é utilizada juntamente com a substituição do bit menos significativo para verificar a integridade dos dados de IoT cifrados do gerador linear congruente combinado (CLCG do inglês *combined linear congruential generator*) sem decifrá-lo.

Em (RASHIDI, 2021), são apresentadas duas estruturas eficientes de baixo custo e alto rendimento da função *hash* SPONGENT. A primeira é a estrutura serializada de 4 bits, que vai apresentar implementação de área pequena com características de tempo aceitáveis para o SPONGENT. A segunda é uma estrutura paralelizada completa com largura do tamanho do caminho de dados do processador, de por exemplo 32 bits ou 64 bits, ideal para aplicações criptográficas de alta velocidade e alta taxa de transferência.

Em (ALAWIDA et al., 2021), é proposta uma nova função *hash* caótica baseada na construção esponja, na computação de ácido desoxirribonucleico (DNA, do inglês *deoxyribonucleic acid*) e no estado finito caótico determinístico autômato (DCFSA, do inglês *deterministic chaotic finite state automata*). A construção esponja foi utilizada para aumentar a segurança da função *hash* e a computação de DNA para reduzir os blocos de mensagem e assim também melhorar a segurança e eficiência. Como resultado, a função *hash* proposta apresentou propriedades estatísticas superiores às de outras funções *hash* caóticas.

Em (GROSS et al., 2017), são apresentadas implementações em hardware do ASCON apropriadas para tags de identificação de rádio frequência (RFID, do inglês *radio frequency identification*), nós de sensores sem fio, sistemas embarcados e aplicativos que necessitam operar com alta capacidade de processamento, velocidade e eficiência para lidar com grandes volumes de dados em tempo real. Também é mostrado que, além de pequeno e rápido, o ASCON é seguro contra ataques de análise de potência diferencial (DPA, do inglês *differential power analysis*).

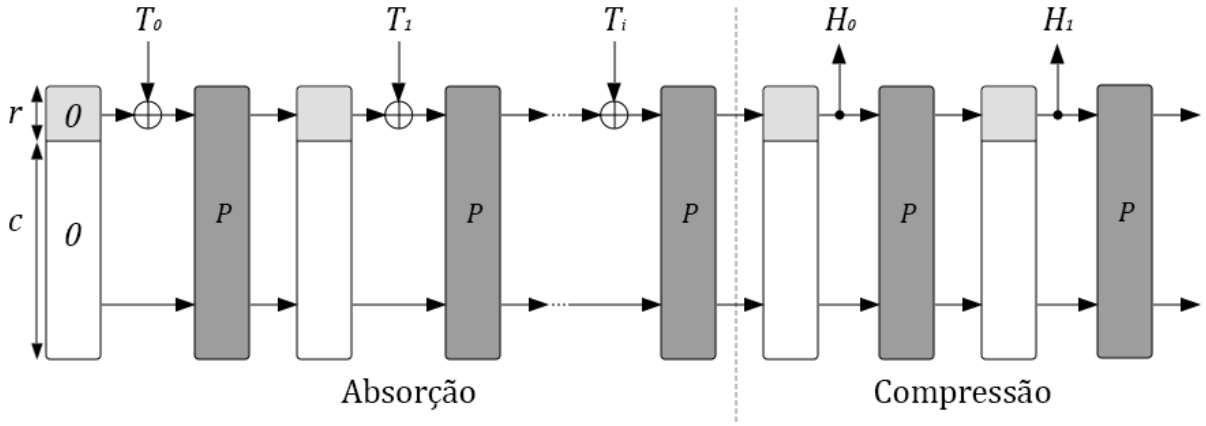
3.2 CONSTRUÇÃO ESPONJA

A construção esponja é um modelo iterativo simples que executa uma função de permutação $P(.)$, operando com um número fixo de b bits. Essa função aceita uma entrada de comprimento variável e produz uma saída de comprimento arbitrário, sendo definida por quatro parâmetros fundamentais: o tamanho do estado b, a taxa de bits r, que é a quantidade de bits que são atualizados a cada bloco de dados processado, a capacidade c, que é a parte do estado interno que não absorve os dados de entrada e está ligada à segurança, e o tamanho da saída n. Dessa forma, seu estado interno obedece a relação $b = r + c$ (BERTONI et al., 2007; ALFRHAN; MOULAHI; ALABDULATIF, 2021; NABEEL; HABAEBI; ISLAM, 2021).

A Figura 1 esquematiza a construção esponja, destacando suas duas fases distintas: absorção e compressão. Primeiramente, antes de iniciar a fase de absorção, todos os bits do estado são inicializados com valor zero, e a mensagem de entrada T é particionada em blocos de r bits.

Seguindo para a fase de absorção, ocorre uma operação XOR nos blocos de mensagem de entrada de r bits, intercalada com a aplicação da função de permutação $P(.)$. Após o processamento de todos os blocos de mensagens, a construção esponja inicia a fase de compressão. Nessa fase, os primeiros r bits do estado são retornados como blocos de saída H, intercalados com aplicações da permutação $P(.)$, resultando no comprimento desejado (BERTONI et al., 2012).

Figura 1 – Esquemático da construção esponja.



Fonte: Adaptado de (NABEEL; HABAEBI; ISLAM, 2021).

A construção esponja se revela uma ferramenta versátil, empregada na criação de funções *hash*, MAC e cifras de fluxo (BERTONI et al., 2007). Operando com uma transformação, ou permutação, ela embaralha a entrada com seu estado interno, resultando em uma saída. Ao construir uma função *hash*, a entrada é misturada durante a absorção e parte do estado interno é extraído para gerar a saída. Para o MAC, a entrada consiste na concatenação de uma chave e a mensagem. A saída da construção esponja é então truncada para atingir o tamanho desejado. No caso da construção de uma cifra de fluxo, a esponja recebe como entrada a concatenação de uma chave secreta e um diversificador (valor adicional). O resultado gerado é uma sequência de bits denominada “fluxo de chaves”, que é uma sequência pseudoaleatória ou chaves temporárias. Independentemente do cenário, o estado interno permanece oculto do adversário.

Ao ser empregada como primitiva criptográfica a construção esponja pode ser utilizada como referência para reivindicações de segurança quando uma transformação ou permutação aleatória é adotada (BERTONI et al., 2008). Foi provado que a construção esponja é indistinguível de um oráculo aleatório. Isso significa que a construção esponja oferece um alto nível de segurança comparável ao de um oráculo aleatório, o qual é considerado um sistema ideal (BERTONI et al., 2008; AUMASSON et al., 2013). Dessa forma, para uma construção esponja com capacidade c, taxa de bits r e uma saída de n bits, seus valores de segurança a ataques de resistência são (GUIDO et al., 2011; BERTONI et al., 2008; BERTONI et al., 2010; GUO; PEYRIN; POSCHMANN, 2011):

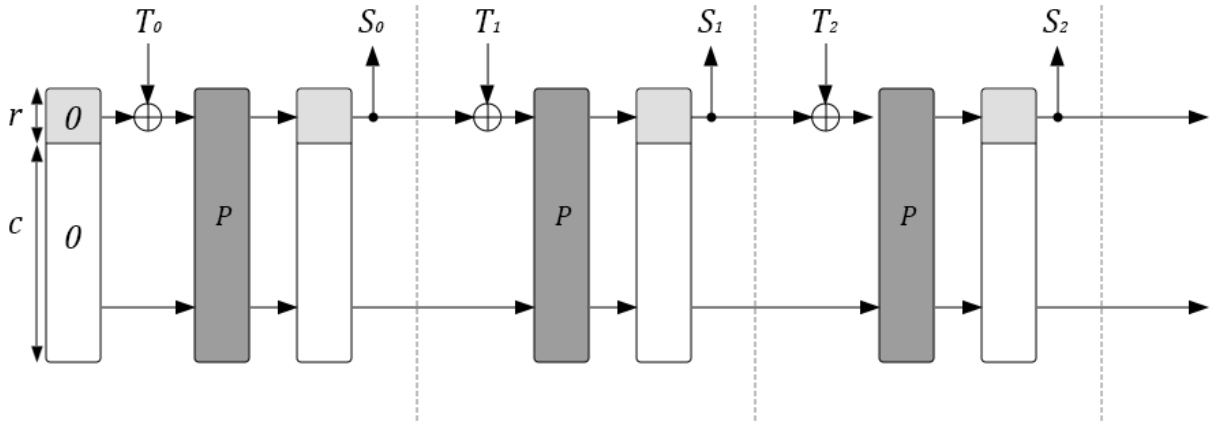
- **Colisão:** $\min\{2^{n/2},\ 2^{c/2}\}$
- **Primeira pré-imagem:** $\min\{2^{n},\ 2^{c},\ \max\{2^{n-r},\ 2^{c/2}\}\}$
- **Segunda pré-imagem:** $\min\{2^{n},\ 2^{c/2}\ \}$

Esses valores de segurança representam a quantidade de operações ou tentativas necessárias para executar com sucesso o ataque criptográfico. Entretanto, ataques contra uma função esponja implicam que sua permutação pode ser distinguida de uma permutação aleatória. Dessa forma, foi criada a estratégia da esponja hermética, que consiste em adotar a construção esponja com uma permutação que não possui vulnerabilidades estruturais (GUIDO et al., 2011).

Posteriormente, em (BERTONI et al., 2012) é apresentada uma construção baseada na construção esponja, chamada de construção *duplex*. Da mesma forma que na construção esponja, a construção *duplex* utiliza uma permutação ou transformação fixa e um estado interno de b bits, composto pela taxa de bits r e a capacidade c. Além disso, ela apresenta o mesmo nível de segurança que a construção esponja.

Na Figura 2, está ilustrado o funcionamento da construção *duplex*. Primeiramente, os bits do estado interno são inicializados como zero. Após a inicialização, é recebida uma sequência de bits como mensagem de entrada, T; ela é integrada aos primeiros r bits do estado interno por meio de uma operação XOR. Em seguida, a permutação $P(.)$ é aplicada ao estado e por fim são extraídos do estado, no máximo, r bits gerando a saída S.

Figura 2 – Construção *duplex*.



Fonte: Adaptado de (BERTONI et al., 2012).

Essa construção foi criada com o intuito principal de ser aplicada em autenticação de mensagens, no entanto, ela também pode ser aplicada em um gerador de bits pseudoaleatório (BERTONI et al., 2012).

3.3 CIFRA

Uma cifra é um algoritmo usado para cifrar informações, transformando-as em uma forma codificada utilizando uma chave. A partir dela, a mensagem original torna-se ilegível para aqueles sem a chave de decifragem correspondente. A codificação de um texto claro é alcançada por meio de uma função ou processo de cifragem. Ao aplicar essa função, a mensagem se transforma em um texto cifrado. Para reverter um texto cifrado, trazendo-o de volta à sua forma original, é empregada a função de decifragem (MENEZES, 1997).

O principal propósito de uma cifra é garantir a confidencialidade das informações, seja durante a transmissão ou o armazenamento. Dessa forma, para que a cifragem das informações seja feita de forma segura, é essencial que a chave seja secreta, ou seja, não seja de conhecimento público. Existem dois tipos principais de cifras no que diz respeito ao compartilhamento de chaves: cifras simétricas e cifras assimétricas. Na cifra simétrica, a mesma chave é utilizada tanto para cifrar quanto para decifrar. Por outro lado, na cifra assimétrica, um par de chaves é utilizado, composto por uma chave pública, que pode ser compartilhada, e uma chave privada, mantida em segredo, uma vez que a decifragem é feita usando esta chave.

Sistemas de chave simétrica dividem-se em duas classes principais: cifra de bloco e cifra de fluxo. Na cifra de bloco, as mensagens de texto claro são divididas em blocos de tamanho fixo, sendo cada bloco cifrado individualmente. Um exemplo clássico desta cifra é o padrão de criptografia avançada (AES, do inglês *advanced encryption standard*) (DWORKIN et al., 2001). Enquanto a cifra de fluxo opera usando uma corrente pseudoaleatória de bits conhecida como fluxo de chave (ou *keystream*) para cifrar continuamente a mensagem em formato de *string*, um exemplo clássico é a Cifra de Vernam (MENEZES, 1997).

As cifras de bloco tendem a ser utilizadas como bloco de construção em outras primitivas criptográficas, como gerador de números pseudoaleatórios (AL-MHADAWI; ALBAHRANI; LAFTA, 2023), funções *hash* (GUO; PEYRIN; POSCHMANN, 2011; BOGDANOV et al., 2011), MACs (WU et al., 2016), assinaturas digitais (NAGAMANI; MONISHA, 2022) e até mesmo outras cifras de fluxo (CANNIERE, 2006). Em contraste, a cifra de fluxo encontra sua aplicação mais proeminente em hardware, onde seus circuitos menos complexos conferem velocidade superior em comparação com cifras de bloco (SREELAJA; PAI, 2012; NOURA et al., 2023). Ela também é ideal para aplicações em comunicação segura, especialmente quando não se possui muito armazenamento e quando os bits precisam ser processados individualmente conforme são recebidos (PUDI; CHATTOPADHYAY; LAM, 2018).

3.4 FUNÇÃO *HASH*

A função *hash* é uma função computacionalmente eficiente que mapeia *strings* binárias de comprimento arbitrário para *strings* binárias de comprimento fixo (MENEZES, 1997). Para que uma função *hash* seja utilizada na criptografia, ela deve possuir algumas propriedades

importantes, as quais são (GUO; YU, 2022; MENEZES, 1997):

i. **Tamanho fixo:** ao receber uma entrada qualquer é criada uma saída com um tamanho fixo. Dessa forma reduzindo mensagens para assinaturas digitais.

ii. **Resistência à pré-imagem:** dado um *hash* de saída, é computacionalmente inviável fazer a engenharia reversa para obter a entrada original, ou seja, em termos computacionais, é difícil encontrar uma pré-imagem x' tal que $h(x') = y$, dado um y quando não se conhece a entrada.

iii. **Segunda resistência à pré-imagem:** é computacionalmente inviável obter a mesma saída de *hash* para uma entrada diferente mesmo sabendo a entrada do *hash* original. Isto é, dada uma entrada x, em termos computacionais, é difícil encontrar uma entrada x', em que $x \neq x'$, tal que $h(x) = h(x')$. Essa resistência também é conhecida como fraca resistência a colisão.

iv. **Resistência à colisão:** é computacionalmente inviável produzir o mesmo *hash* de saída para quaisquer duas entradas distintas, x e x', isto é, $h(x) = h(x')$. Ela também é conhecida como forte resistência à colisão.

v. **Grande mudança:** caso haja qualquer alteração no bit da entrada, será produzida uma saída de *hash* significativamente diferente.

Na criptografia, a função *hash* é utilizada em assinaturas digitais, autenticação de mensagem (integridade de dados), na construção de uma função pseudoaleatória (PRF, do inglês *pseudorandom function*) ou gerador de número pseudoaleatório (PRNG, do inglês *pseudorandom number generator*), no gerador de chaves, na derivação de chave criptográfica, na segurança de senhas e em *Blockchain* (MENEZES, 1997; STALLINGS, 2023; WINDARTA et al., 2022). No caso de dispositivos com recursos limitados, deve ser utilizada uma função *hash* leve (WINDARTA et al., 2022), uma vez que ela é uma função que apresenta um algoritmo criptográfico que faz demandas mínimas de recursos ao sistema para funcionar (STALLINGS, 2023). Nesta seção serão apresentadas funções *hash* da família de algoritmo de *hash* seguro e funções leves que utilizam a construção esponja em sua estrutura.

3.5 SHA-3

A família SHA-3 é uma família de funções *hash* baseada no KECCAK, o qual foi o algoritmo vencedor da Competição de Algoritmo de *Hash* Criptográfico SHA-3 realizada pelo NIST, a agência do governo dos Estados Unidos responsável por estabelecer padrões e diretrizes em várias áreas, incluindo criptografia e segurança da informação. Essa competição tinha como objetivo desenvolver uma nova função *hash* para complementar o SHA-1 e SHA-2. A diferença dessa família é que o SHA-3 tem a sua estrutura baseada na construção esponja, e ele é composto

por quatro funções *hash* criptográficas, denominadas SHA3-224, SHA3-256, SHA3-384 e SHA3-512 . Além disso, também apresenta duas funções de saída extensível (XOF, do inglês *extendable-output function*), denominadas SHAKE128 e SHAKE256 (DWORKIN, 2015). Na Tabela 1, são listadas as versões que compõem a família SHA-3, contendo o tamanho da saída n, a taxa de bits r e a capacidade c. Para as funções com saída extensível, d representa o valor de um comprimento desejado como saída.

Tabela 1 – Família SHA-3.

Versão	Saída (n)	Taxa de bits (r)	Capacidade (c)
SHA3-224	224	1152	448
SHA3-256	256	1088	512
SHA3-384	384	832	768
SHA3-512	512	576	1024
SHAKE128	d	1344	256
SHAKE256	d	1088	512

Fonte: (DWORKIN, 2015).

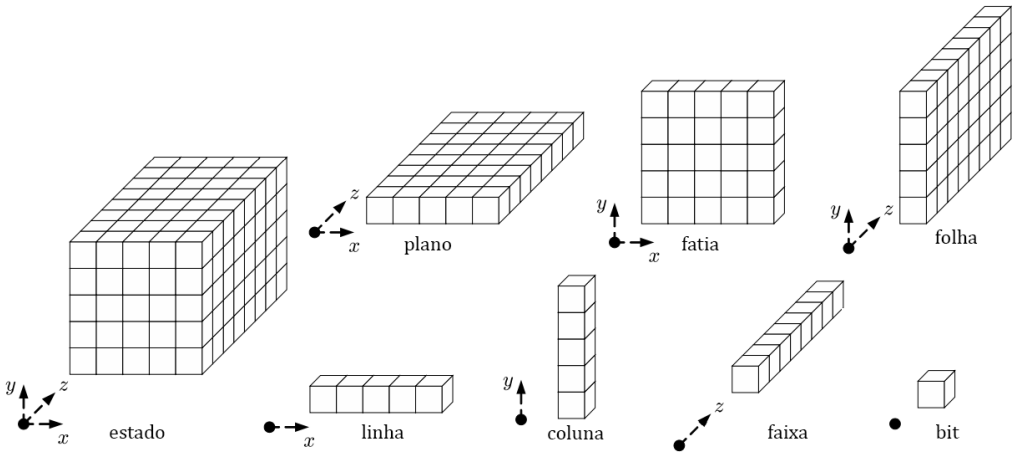
Os tamanhos das saídas de *hash* nas funções SHA-3 são especificados nos seus sufixos, correspondendo aos mesmos valores em bits que os das funções SHA-2. Dessa forma, é possível implementar as funções SHA-3 em substituição às funções do SHA-2, e vice-versa. No caso das XOFs, que são funções cujas saídas podem ser estendidas para qualquer comprimento desejado, o sufixo não representa o tamanho da saída do *hash* e sim o nível de segurança que essas funções geralmente podem suportar, em outras palavras, é a sua resistência a ataques criptográficos.

O estado interno do SHA-3, é conhecido como matriz de estados, e é representado por uma matriz tridimensional $A[x, y, z]$, em que $0 \leq x < 5$, $0 \leq y < 5$ e $0 \leq z < w$. Este estado pode ser dividido em plano, fatia, folha, linha, coluna, faixa e bit, como ilustrado na Figura 3. O SHA-3 utiliza uma permutação expressa por $P_{SHA3}(b, N_r)$, em que b é o tamanho do estado interno e N_r o número de rodadas da transformação interna. O estado interno para essa permutação é composto por b bits, em que $b \in \{25, 50, 100, 200, 400, 800, 1600\}$ está relacionado aos valores $b/25$ e $log(b/25)$, denotados respectivamente por w e l. Aqui, w representa o tamanho, em bits, de cada faixa da matriz de estados e l é o logaritmo binário de w, que serve para definir o número de rodadas da permutação e determina a quantidade de bits de uma constante de rodada gerados para cada rodada da permutação.

A matriz de estados, juntamente com um inteiro chamado índice da rodada (*round index*), compõem as rodadas de permutação. Uma rodada da permutação $P_{SHA3}(.)$, denotada por Rnd, consiste de uma sequência de cinco transformações, θ, ρ, π, χ e ι, chamadas de mapeamentos de etapas. Essas transformações são aplicadas da seguinte forma:

$$Rnd(A, i_r) = \iota(\chi(\pi(\rho(\theta(A)))), i_r), \tag{3.1}$$

Figura 3 – Matriz de estados.



Fonte: Adaptada de (DWORKIN, 2015).

em que:

- θ: realiza uma operação XOR, somando todos os bits na coluna à esquerda do bit que será processado. A mesma operação é realizada com todos os bits da coluna localizada na sua frontal direita. Por fim, é feito um XOR desses valores com o do bit processado. Isto é, para todos os pares (x, z) tais que $0 \leq x < 5$ e $0 \leq z < w$,

$$C[x, z] = A[x, 0, z] \oplus A[x, 1, z] \oplus A[x, 2, z] \oplus A[x, 3, z] \oplus A[x, 4, z]. \tag{3.2}$$

  Para todos os pares (x, z) tais que $0 \leq x < 5$ e $0 \leq z < w$,

$$D[x, z] = C[(x - 1) \,(\text{mod}\, 5), z] \oplus C[(x + 1) \,(\text{mod}\, 5), (z - 1) \,(\text{mod}\, w)]. \tag{3.3}$$

  Para todos os ternos (x, y, z) tais que $0 \leq x, y < 5$ e $0 \leq z < w$,

$$A'[x, y, z] = A[x, y, z] \oplus D[x, z]. \tag{3.4}$$

- ρ: rotaciona os bits de cada faixa à esquerda por um comprimento chamado de *offset*. O *offset* é calculado com base nas coordenadas x e y de cada faixa. De modo equivalente, cada bit do eixo z é modificado ao adicionar o *offset* e o módulo do tamanho da faixa. Logo, para todo z, tal que $0 \leq z < w$, seja $A'[0, 0, z] = A[0, 0, z]$ e seja $(x, y) = (1, 0)$. E para t que assume valores de 0 à 23 e para todo z, tal que $0 \leq z < w$, o *offset* é determinado por

$$A'[z, y, z] = A[x, y, (z - (t + 1)(t + 2)/2) \,(\text{mod}\, w)]. \tag{3.5}$$

  Para o mesmo intervalo de t, os valores de x e y são atualizados por

$$(x, y) = (y, (2x + 2y) \,(\text{mod}\, 5)). \tag{3.6}$$

- π: reorganiza todas as posições das faixas, exceto a faixa que contém as coordenadas x, y = 0. A reorganização ocorre da forma: para todos os ternos (x, y, z) tais que $0 \leq x, y < 5$ e $0 \leq z < w$,

$$A'[x, y, z] = A[(x + 3y) \pmod 5), x, z]. \tag{3.7}$$

- χ: atualiza os valores de todos os bits de cada faixa, usando as operações XOR, NOT e AND. Nessa etapa, o valor de um bit é atualizado ao aplicar uma função não-linear com outros dois bits que estão na sua mesma linha do eixo x. Dito isto, para todos os ternos (x, y, z) tais que $0 \leq x, y < 5$ e $0 \leq z < w$,

$$A'[x, y, z] = A[x, y, z] \oplus ((A[(x+1) \pmod 5), y, z] \oplus 1) \cdot A[(x+2) \pmod 5), y, z]). \tag{3.8}$$

- ι: modifica os bits da faixa (0,0) fazendo uma operação XOR dos mesmos com a constante de rodada RC. Essa etapa é parametrizada por uma constante chamada de índice da rodada i_r. A partir da constante i_r, é calculada a constante de rodada RC para cada rodada. Daí, para todo j que assume valores de 0 à l,

$$RC[2^j - 1] = rc(j + 7i_r). \tag{3.9}$$

  Nesse caso, l determina quantos bits da constante de rodada são gerados na função RC. Com o RC calculado, é feito o XOR com a faixa z em que x e y tem valor 0. Para todo z, tal que $0 \leq z < w$,

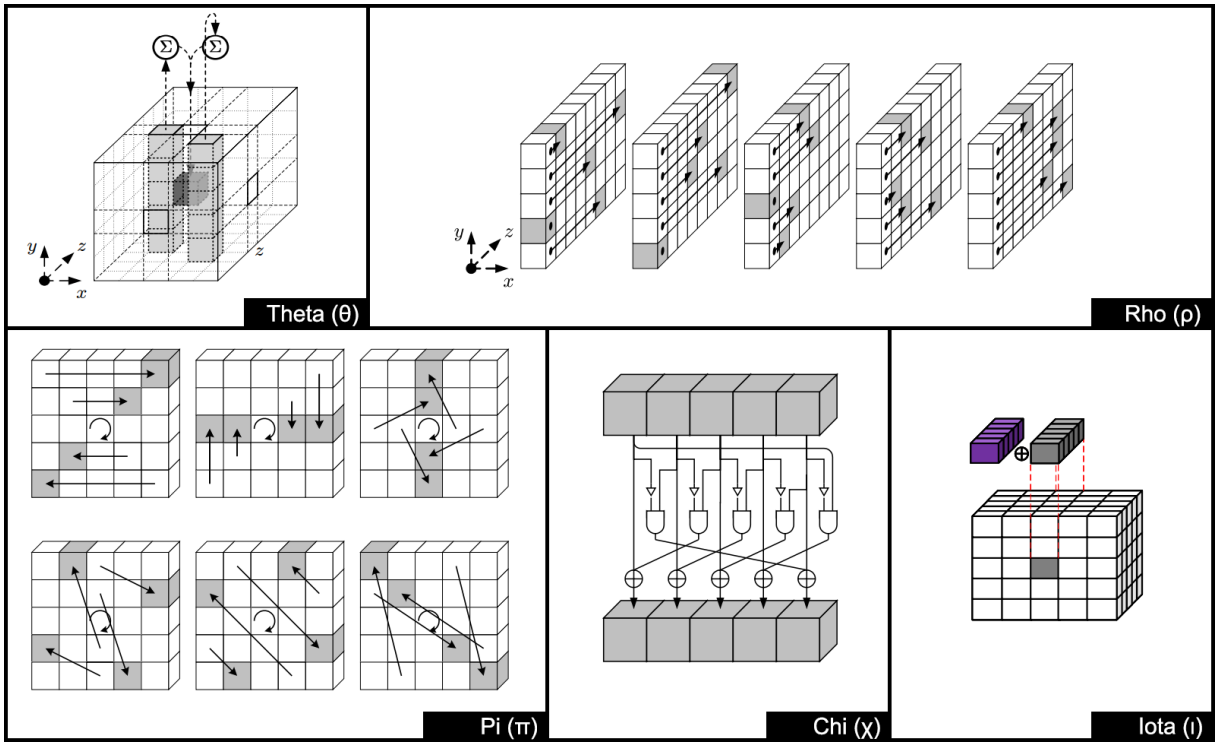
$$A'[0, 0, z] = A[0, 0, z] \oplus RC[z]. \tag{3.10}$$

Na Figura 4 são ilustradas as transformações que ocorrem em uma rodada de permutação do SHA-3.

Em relação à segurança, as quatro funções *hash* do SHA-3 foram desenvolvidas para serem iguais ou superar as resistências à colisão, pré-imagem e segunda pré-imagem fornecidas pelo SHA-2 (DWORKIN, 2015). Além disso, elas demonstram que são resistentes a outros ataques, como por exemplo o ataque de extensão de comprimento, também chamado de ataque de preenchimento (*padding attack*). Esse ataque é um tipo de ataque criptográfico no qual um intruso pode adicionar dados extras a uma mensagem autêntica, seja no início ou no final dela, e fazer com que a mensagem resultante seja aceita como genuína (TSUDIK, 1992). No caso das duas funções XOF, elas também apresentam resistência contra ataques genéricos contra funções *hash*, bem como outros ataques que possam ser defendidos por uma função qualquer com comprimento de saída desejado, oferecendo um nível de segurança de 128 ou 256 bits, dependendo da função utilizada. Na Tabela 2 são apresentados os níveis de segurança da família SHA-3, em que d representa o valor de um comprimento desejado como saída de *hash*.

Assim como o SHA-2, o SHA-3 possui implementações em hardware, software, firmware, ou uma combinação deles (DWORKIN, 2015; MOUMNI; FETTACH; TRAGHA, 2019; CHOI; SEO, 2021).

Figura 4 – Mapeamentos de etapas do SHA-3.



Fonte: Adaptado de (DWORKIN, 2015; CHOI; SEO, 2021).

Tabela 2 – Nível de segurança da família SHA-3.

Resistência (bits)	SHA3-224	SHA3-256	SHA3-384	SHA3-512	SHAKE128	SHAKE256
Colisão	112	128	192	256	min(d/2,128)	min(d/2,256)
Pré-imagem	224	256	384	512	≥min(d, 128)	≥min(d, 256)
Segunda pré-imagem	224	256	384	512	min(d, 128)	min(d, 128)

Fonte: (DWORKIN, 2015).

3.6 QUARK

Quark é uma família de funções *hash* leves proposta por (AUMASSON et al., 2013). Inicialmente, foram propostas três versões da função Quark (AUMASSON et al., 2013), em que cada versão possui diferentes taxas de bits, capacidades, larguras, comprimentos de saída e funções booleanas não-lineares. No entanto, elas possuem a mesma função booleana linear. Uma função booleana $f(x)$ é uma função que tem como domínio o espaço vetorial $\mathbb{F}_2^n$ de tuplas binárias de tamanho n, $(x_1, x_2, ..., x_n)$, cujos elementos assumem os valores 0 e 1 no corpo $\mathbb{F}_2$, isto é $\mathbb{F}_2 = \{0, 1\}$ (PRENEEL et al., 1991; TILBORG; JAJODIA, 2011). Posteriormente, foi apresentada uma versão mais pesada do Quark, a qual é baseada na construção *duplex* e que possui uma segurança mais robusta que as demais versões (AUMASSON; KNELLWOLF; MEIER, 2012). Na Tabela 3, estão apresentadas as versões do Quark, contendo os valores em

bits do tamanho da saída n, taxa de bits r e capacidade c.

Tabela 3 – Versões da função *hash* Quark

Versão	Saída (n)	Taxa de bits (r)	Capacidade (c)
U-Quark	136	8	128
D-Quark	176	16	160
S-Quark	256	32	224
C-Quark	384	64	320

Fonte: (AUMASSON et al., 2013).

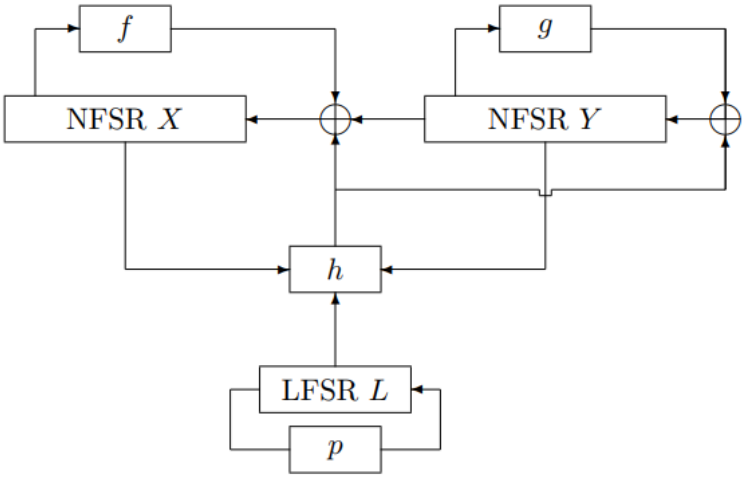
A permutação $P_{Quark}(.)$ utilizada no Quark é baseada em uma combinação das cifras de fluxo Grain e cifra de bloco KATAN. Após analisar as características de ambas, os autores (AUMASSON et al., 2013) escolheram as funcionalidades mais adequadas de cada uma para a criação da permutação. Dessa forma, foi feita a permutação $P_{Quark(.)}$ que conta com três funções booleanas não-lineares, f, g e h, e conta também com uma função booleana linear p. O estado interno dessa permutação, em um determinado momento durante a execução do algoritmo, é representado por uma época $t \geq 0$ e é composto por dois registradores de deslocamento com realimentação não-linear (NFSR, do inglês *nonlinear-feedback shift register*), X e Y, e um registrador de deslocamento com realimentação linear (LFSR, do inglês *linear-feedback shift register*), L.

Em (AUMASSON et al., 2013), foram escolhidas as funções f e g semelhantes à função não linear do Grain-v1, uma vez que elas alcançam uma boa não linearidade e resistência, produzindo resultados variáveis e imprevisíveis, tornando-as robustas contra ataques criptográficos. Além disso, essas funções são polinômios de grau seis e incluem termos de todos os graus abaixo de seis. Já a função h, será de um grau menor comparado com o grau das funções f e g, mas terá mais termos lineares para aumentar a difusão cruzada entre os dois registradores. Essa escolha promove mais interação entre os dois registradores, melhorando assim a dispersão de dados entre eles.

O diagrama da permutação do Quark é apresentado na Figura 5. Os dois registradores não-lineares X e Y têm $b/2$ bits de tamanho e são denotados por $X^t = (X_0^t, ..., X_{b/2-1}^t)$ e $Y^t = (Y_0^t, ..., Y_{b/2-1}^t)$, respectivamente. Enquanto que o registrador linear L tem $\lceil \log 4b \rceil$ bits de tamanho e é denotado por $L^t = (L_0^t, ..., L_{\lceil \log 4b \rceil - 1}^t)$. Assim, o processamento de uma entrada s com b bits de tamanho feito pela permutação $P_{Quark(.)}$ ocorre em três etapas.

- **Inicialização**: dada uma entrada $S = (S_0, ..., S_{b-1})$, a permutação $P_{Quark(.)}$ irá inicializar seu estado interno dividindo os b bits da entrada entre os dois registradores X e Y. X é inicializado com os primeiros $b/2$ bits de entrada; Y é inicializado com os últimos $b/2$ bits

Figura 5 – Diagrama da permutação do Quark.



Fonte: (AUMASSON et al., 2013).

de entrada; e L é inicializado com uma *string* preenchida com 1. Logo, os valores de X, Y e L inicializados são

$$\begin{aligned}(X_0^0, ..., X_{b/2-1}^0) &:= (S_0, ..., S_{b/2-1}),\\ (Y_0^0, ..., Y_{b/2-1}^0) &:= (S_{b/2}, ..., S_{b-1}),\\ (L_0^0, ..., L_{\lceil \log 4b \rceil}^0) &:= (1, 1, ..., 1).\end{aligned} \tag{3.11}$$

- **Atualização de estado**: a partir de um estado interno (X^t, Y^t, L^t), ao ser realizado um *clock* do mecanismo interno, determina-se o próximo estado $(X^{t+1}, Y^{t+1}, L^{t+1})$. Isto é, atualiza o valor dos registradores para o próximo. Primeiramente, h^t é determinado por $h(X^t, Y^t, L^t)$; a seguir são realizados os *clockings* de: X usando Y_0^t, a função f e h^t; Y usando a função f e h^t; e L usando a função p. Nessa atualização, os registradores são realimentados com suas respectivas funções booleanas e seu último bit é substituído, se tornando

$$\begin{aligned}(X_0^{t+1}, ..., X_{b/2-1}^{t+1}) &:= (X_1^t, ..., X_{b/2-1}^{t+1}, Y_t^0 + f(X^t) + h^t),\\ (Y_0^{t+1}, ..., Y_{b/2-1}^{t+1}) &:= (Y_1^t, ..., Y_{b/2-1}^{t+1}, g(X^t) + h^t),\\ (L_0^{t+1}, ..., L_{\lceil \log 4b \rceil}^{t+1}) &:= (L_1^t, ..., L_{\lceil \log 4b \rceil - 1}^t, p(L^t).\end{aligned} \tag{3.12}$$

- **Cálculo da saída**: depois do estado do Quark ser atualizado $4b$ vezes a saída será o valor final dos dois registradores não-lineares, X e Y. A ordem dos bits é a mesma que foi seguida na inicialização, em que X ocupa os primeiros $b/2$ bits e Y os últimos $b/2$ bits. Como resultado, S tem a seguinte configuração,

$$S \quad = \quad (S_0, ..., S_{b-1}) = (X_0^{4b}, X_1^{4b}, ..., Y_{b/2-2}^{4b}, Y_{b/2-1}^{4b}). \tag{3.13}$$

Em termos de segurança, por usar a construção esponja em sua estrutura, a função *hash* Quark oferece uma resistência a ataques de pré-imagem de 2^c. Além disso, ela oferece segurança contra ataques de colisão genéricos e ataques genéricos de segunda pré-imagem de aproximadamente $2^{\frac{c}{2}}$. Análises de segurança foram conduzidas, submetendo sua função Quark contra alguns tipos de ataques, como por exemplo, ataque genérico de segunda pré-imagem, ataques de cubo e testadores de cubo, ataques diferenciais e ressincronização de slides, se mostrando resistente a esses tipos de ataques (AUMASSON et al., 2013).

A função Quark foi criada para ser implementada em hardware de dispositivos com recursos limitados, como por exemplo RFID. Portanto, ela não é otimizada para ser utilizada em software. Ela pode ser implementada de duas formas, a serial ou a paralela. A implementação serial usa apenas um módulo de permutação e é uma estrutura mais compacta. Já a paralela utiliza mais módulos de permutação, resultando em uma execução mais rápida. A função Quark apresenta características vantajosas para seu uso como uma função *hash* leve. Devido a utilização de registradores, que são facilmente implementáveis, juntamente com a capacidade de realizar vários *trade-offs* entre espaço e tempo, que contribuem para a sua versatilidade e eficiência.

3.7 PHOTON

A família de funções *hash* Photon, apresentada em (GUO; PEYRIN; POSCHMANN, 2011), é uma classe de algoritmos orientadas a hardware. Da mesma forma que no Quark, essa família também emprega a construção esponja em sua estrutura, com o intuito de manter o tamanho de sua memória interna o mais baixo possível e, como resultado, oferecer flexibilidade nas relações entre velocidade, ocupação de espaço e níveis de segurança.

Em (GUO; PEYRIN; POSCHMANN, 2011), são propostas cinco versões do Photon, destinadas a uma ampla gama de aplicações. Suas funções são denotadas por Photon-n/r/r', em que r é a taxa bits de entrada, r' a taxa de bits de saída e n o tamanho da saída que assume valores de 64 a 256 bis. O seu estado interno, denotado por interno $b = (c + r)$, varia de acordo com o tamanho de saída do *hash* e pode assumir os seguintes valores: 100, 144, 196, 256 e 288. Para cada valor de b, são utilizadas permutações internas específicas, denotadas como $P_{Photon_b}(.)$:= {$P_{Photon_{100}}(.)$, $P_{Photon_{144}}(.)$, $P_{Photon_{196}}(.)$, $P_{Photon_{256}}(.)$ e $P_{Photon_{288}}(.)$}, resultando nas suas versões apresentadas na Tabela 4. Na tabela são incluídas as permutações internas de cada versão e os parâmetros de tamanho de saída n, taxa de bits r e capacidade c, em bits. Vale ressaltar que o Photon-80/20/16 é a única versão que apresenta uma taxa de bits de saída diferente da taxa de bits da entrada, em que $r' = 16$ e $r = 20$, respectivamente.

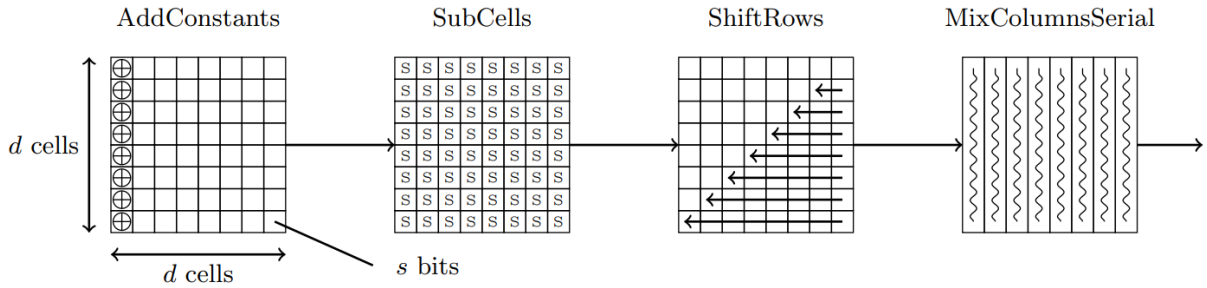
Tabela 4 – Versões da função *hash* Photon.

Permutação	Versão	Saída (n)	Taxa de bits (r)	Capacidade (c)
$P_{Photon_{100}}(.)$	Photon-80/20/16	80	20	60
$P_{Photon_{144}}(.)$	Photon-128/16/16	128	16	112
$P_{Photon_{196}}(.)$	Photon-160/36/36	160	36	124
$P_{Photon_{256}}(.)$	Photon-224/32/32	224	32	192
$P_{Photon_{288}}(.)$	Photon-256/32/32	256	32	224

Fonte: (GUO; PEYRIN; POSCHMANN, 2011).

Sua permutação interna $P_{Photon_b}(.)$ é projetada com características semelhantes ao AES e é aplicada a um estado interno de d^2 elementos de s bits cada, representado por uma matriz $d \times d$. Diferentemente do AES, nessa permutação não são permitidas chaves secretas como parâmetro de entrada. Isso resulta em uma redução de eficiência do algoritmo em alguns aspectos, como velocidade ou flexibilidade. Dessa forma, são evitados ataques que procuram fraquezas no escalonamento da chave (GUO; PEYRIN; POSCHMANN, 2011). Mais especificamente, a permutação $P_{Photon_b}(.)$ é composta por quatro fases que são aplicadas em N_r rodadas a um estado interno S, e está apresentada na Figura 6.

Figura 6 – Etapas de uma rodada da permutação do Photon.



Fonte: (GUO; PEYRIN; POSCHMANN, 2011).

- ***AddConstants* (AC)**: nesta etapa, é feita uma operação XOR das constantes da rodada $RC(v)$ e das constantes internas $IC_d(i)$, na rodada v, aplicada a cada célula da primeira coluna do estado interno. Essas constantes internas dependem da quantidade de células d e da posição da linha i. Daí, para cada rodada v tem-se

$$S'[i,0] = S[i,0] \oplus RC(v) \oplus IC_d(i), \quad \text{para} \quad 0 \leq i < d. \tag{3.14}$$

- ***SubCells* (SC)**: é aplicada uma S-box a cada célula do estado interno. A S-box aplicada vai depender do tamanho s em bits de cada célula que pode ser de 4 bits ou de 8 bits. Para células de tamanho 4 bits é utilizada a S-box da cifra de bloco PRESENT (BOGDANOV et al., 2007). E para as células de 8 bits é utilizada a S-box da cifra de bloco AES (DWORKIN

et al., 2001). Essas escolhas permitem que possam ser feitas implementações de software mais simples e rápidas. A aplicação é feita da forma

$$S'[i,j] = SBOX(S[i,j]), \quad \text{para} \quad 0 \leq i,j < d. \tag{3.15}$$

- ***ShiftRows*** **(ShR)**: as posições das células são rotacionadas para a esquerda por i posições de colunas para cada linha i. Dessa forma, essa etapa é descrita como

$$S'[i,j] = S[i,(j+i)\,(\text{mod}\,d)] \quad \text{para} \quad 0 \leq i,j < d. \tag{3.16}$$

- ***MixColumnsSerial*** **(MCS)**: esta etapa mistura todas as colunas de forma linear independentemente. Para cada vetor de entrada da coluna j, o qual é da forma $(S[0,j],...,S[d-1,j])^T$, uma matriz $A_b = Serial(Z_0,...,Z_{d-1})$ é aplicada d vezes. Dessa forma, tem-se que

$$(S'[0,j],...,S[d-1,j])^T = A_b^d \times (S[0,j],...,S[d-1,j])^T. \tag{3.17}$$

A_b também é uma matriz $d \times d$, da forma:

$$A_b = \begin{bmatrix} 0 & 1 & 0 & 0 & ... & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & ... & 0 & 0 & 0 & 0 \\ & \vdots & & & & & & \vdots & \\ 0 & 0 & 0 & 0 & ... & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & ... & 0 & 0 & 0 & 1 \\ Z_0 & Z_1 & Z_2 & Z_3 & ... & Z_{d-4} & Z_{d-3} & Z_{d-2}) & Z_{d-1} \end{bmatrix}$$

em que $(Z_0,...,Z_{d-1})$ são coeficientes escolhidos pelos autores.

Da mesma forma que no Quark, a capacidade do sistema é equivalente ao tamanho da saída ($c = n$), enquanto a taxa de bits r permanece muito pequena. Esse equilíbrio resulta em uma melhor relação entre segurança e área, permitindo visar requisitos de área extremamente baixos, simultaneamente obtendo um nível de segurança próximo ao ideal (GUO; PEYRIN; POSCHMANN, 2011). Dito isso, em questão de segurança, o Photon pode ter a resistência ideal contra colisão, mas não para segunda pré-imagem. Ele apresenta uma resistência a colisão de $min(2^{n/2}, 2^{c/2})$, resistência a pré-imagem de $min(2^n, 2^c, max(2^{n-r'}, 2^{c/2}))$ e resistência a segunda pré-imagem de $min(2^n, 2^c)$.

O Photon utiliza a estratégia da esponja hermética, logo, enquanto a permutação interna $P_{Photon_b}(.)$ não apresentar falhas estruturais, ele garantirá uma segurança confiável. Entretanto,

mesmo que ele apresente uma falha, a segurança não será comprometida, pois um atacante não teria muitas variáveis de entrada disponíveis para realizar um ataque (GUO; PEYRIN; POSCHMANN, 2011). Adicionalmente, foram realizadas análises de segurança por meio de testes de criptoanálise diferencial/linear, ataques de rebote, super-Sbox, testadores de cubo e ataques algébricos e outras criptoanálises, resultando que o Photon possui uma estrutura segura (GUO; PEYRIN; POSCHMANN, 2011).

A versatilidade do Photon é evidente em suas implementações potenciais, tanto em hardware quanto em software. Testes realizados por (GUO; PEYRIN; POSCHMANN, 2011) indicam que implementações simples em software apresentam desempenho bastante aceitável quando comparado às versões do Quark. A versões U-Quark, D-Quark e S-Quark apresentaram, respectivamente, 8.000, 30.000 e 22.000 ciclos por bytes. Os ciclos por bytes representam as quantidades de ciclos de *clock* necessários para processar 1 byte. Enquanto que no Photon, suas versões Photon-$80/20/16$, Photon-$128/16/16$, Photon-$160/36/36$, Photon-$224/32/32$ e Photon-$256/32/32$ apresentam , respectivamente, 95, 156, 116, 227 e 157 ciclos por bytes. Além disso, as versões do Photon se mostram mais rápidas que o Quark, quando implementado em hardware, apresentando uma área pequena em todas as suas versões. Em termos de latência, a maioria das versões do Photon apresentou desempenho superior, com exceção do Photon-$224/32/32$, que registrou 12.012 ciclos, tornando-se mais lento do que o S-Quark, que alcançou 8.192 ciclos. Quanto à taxa de transferência de dados, o Photon geralmente apresentou um desempenho superior ao do Quark, com exceção do Photon-$224/32/32$, que registrou uma taxa de $0{,}56$ kbps ao processar uma mensagem de entrada de 96 bits, sendo inferior ao desempenho do S-Quark, que alcançou $0{,}85$ kbps.

3.8 SPONGENT

SPONGENT é uma família de funções *hash* apresentada em (BOGDANOV et al., 2011). Ela também possui a sua estrutura baseada na construção esponja e tem um *footprint* menor do que a maioria das funções *hash* leves dedicadas existentes, como por exemplo, Quark. Além disso, sua área se compara à do Photon (BOGDANOV et al., 2013).

Em (BOGDANOV et al., 2013), são introduzidas treze versões do SPONGENT, abrangendo cinco níveis diferentes de segurança para atender às diversas aplicações. Seus parâmetros também são iguais aos da construção esponja e suas versões são denotadas por SPONGENT-$n/c/r$, com os tamanhos do *hash* $n \in (88, 128, 160, 224, 256)$, que resulta nas versões apresentadas na Tabela 5, que também inclui a saída n, a taxa de bits r e a capacidade c, todas em bits. Nota-se que $n = c$ ou $n = \frac{c}{2}$ para todas as versões, exceto em SPONGENT-$88/80/8$.

A permutação $P_{Spongent_b}(.)$ utilizada no SPONGENT é baseada no PRESENT, uma cifra de bloco ultra leve apresentada por (BOGDANOV et al., 2007). Essa permutação é uma transformação de N_r rodadas que opera em um estado interno de b bits de tamanho, conforme

Tabela 5 – Versões da função *hash* Spongent.

Versão	Saída (n)	Taxa de bits (r)	Capacidade (c)
SPONGENT-88/80/8	88	8	80
SPONGENT-88/176/88	88	88	176
SPONGENT-128/128/8	128	8	128
SPONGENT-128/256/128	128	128	256
SPONGENT-160/160/16	160	16	160
SPONGENT-160/160/80	160	80	160
SPONGENT-160/320/160	160	160	240
SPONGENT-224/224/16	224	16	224
SPONGENT-224/224/112	224	112	224
SPONGENT-224/448/224	224	224	448
SPONGENT-256/256/16	256	16	256
SPONGENT-256/256/128	256	128	256
SPONGENT-256/512/256	256	256	512

Fonte: (BOGDANOV et al., 2013).

definido no Algoritmo 1.

Algoritmo 1: Permutação $P_{Spongent_b}(.)$

Input : N_r, STATE
Output : STATE
for $i = 1$ **to** N_r **do**
  $state \leftarrow \text{reverseLC}_b(i) \oplus STATE \oplus lCounter_b(i)$;
  $state \leftarrow sBoxLayer_b(STATE)$;
  $state \leftarrow pLayer_b(STATE)$;
end

O $\mathrm{lCounter_b(i)}$ é o valor de um LFSR dependente de b no tempo i que contém a constante de rodada na rodada i. Esse valor é adicionado aos bits mais à direita do estado interno por meio de uma operação XOR. Enquanto que $\text{reverseLC}_b(i)$ é o valor de $\mathrm{lCounter_b(i)}$ com seus bits em ordem reversa. E ele é adicionado ao bits mais à esquerda do estado interno. Já $sBoxLayer_b$ e $pLayer_b$ descrevem como o estado interno, STATE, é modificado.

A seguir, as etapas da estrutura do PRESENT para larguras b bits são apresentadas por (BOGDANOV et al., 2013):

- $sBoxLayer_b$: nessa etapa, é utilizada uma S-box de 4 bits, que vai ser aplicadas $b/4$ vezes de forma paralela.
- $pLayer_b$: essa etapa é uma extensão da permutação de bits do PRESENT que vai deslocar um bit j do estado interno para a posição do bit $P_b(j)$, em que

$$P_b(j) = \begin{cases} j \cdot b/4 \pmod{b-1}, & \text{se } j \in \{0, \ldots, b-2\}, \\ b-1, & \text{se } j = b-1. \end{cases} \tag{3.18}$$

- $lCounter_b(i)$: é um dos quatro LFSR's de $\lceil \log_2 R \rceil$ bits de tamanho, que é incrementado sempre que for utilizado e tem o valor final configurado como todos os bits sendo 1.

No SPONGENT, os requisitos de segurança padrão são de $2^{n/2}$ e 2^n para resistência à colisão e à pré-imagem e segunda pré-imagem, respectivamente. Em alguns casos, a resistência é reduzida para 2^{n-r} para pré-imagem e reduz para $2^{c/2}$ para segunda pré-imagem e colisão. Já em outros casos, a resistência à pré-imagem e segunda pré-imagem são de $2^{n/2}$, enquanto a resistência à colisão se mantém em $2^{n/2}$. Em (BOGDANOV et al., 2013), são realizadas análises de segurança nos quesitos de resistência contra criptoanálises diferenciais, ataques de colisão, resistência a pré-imagem e ataques lineares. Como a estrutura da permutação é baseada no PRESENT, há garantia de segurança no que diz respeito aos ataques mais importantes, e dependendo da versão do SPONGENT, pode haver uma resistência mais elevada.

Suas implementações são destinadas a hardware, que, dependendo da versão do SPONGENT, podem ser aplicadas em situações extremamente restritas, em situações que os requisitos de segurança são mais flexíveis e podem ser compatíveis com interfaces padrão em aplicações embarcadas leves (RASHIDI, 2021; BOGDANOV et al., 2013). Análises sobre a implementação do SPONGENT em hardware são feitas e comparadas com os outros tipos de função *hash* leve, Quark e Photon (BOGDANOV et al., 2013). Algumas variantes do SPONGENT são similares ao Quark e Photon ao apresentarem vantagens de ter uma área pequena, devido ao nível reduzido de segurança de segunda pré-imagem, enquanto mantém a resistência de colisão de nível padrão. No entanto, SPONGENT apresenta a menor taxa de transferência entre eles, pois o Quark e o Photon possuem um menor número de rodadas ou uma taxa de bits maior.

3.9 ASCON

ASCON é um conjunto de cifras de criptografia autenticada com dados associados (AEAD, do inglês *authenticated encryption with associated data*) e funções *hash*, que proporciona 128 bits de segurança (DOBRAUNIG et al., 2021b). O ASCON foi escolhido pelo NIST como padrão para proteger dispositivos leves, com o propósito de garantir a segurança dos dados gerados e transmitidos pela IoT e outros dispositivos eletrônicos de tamanho reduzido (DOBRAUNIG et al., 2021b; BOUTIN, 2023). Ele é composto pelas cifras ASCON-128, ASCON-128a, pelas funções *hash* ASCON-*hash* e ASCON-*hasha*, e pelas funções de saída extensíveis ASCON-Xof e ASCON-Xofa, que produzem uma saída de tamanho variável. Todas as versões do ASCON compartilham os mesmos parâmetros de taxa de bits e capacidade, sendo eles $r = 64$ e $c = 256$, respectivamente. No entanto, em relação ao tamanho da saída n,

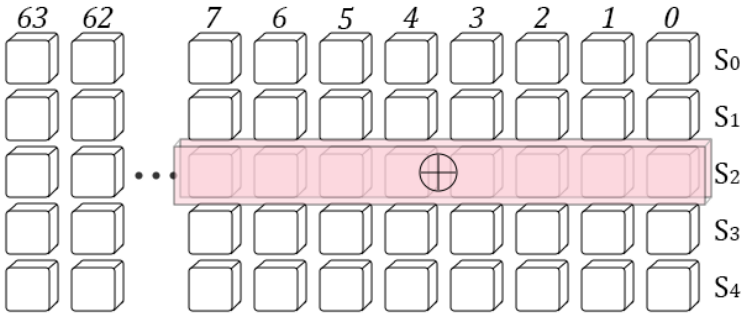
enquanto as funções ASCON-*hash* e ASCON-*hasha* têm $n = 256$ bits, as cifras ASCON-128 e ASCON-128a podem ter um tamanho variável dependendo da entrada, e para as funções de saída extensível, ASCON-Xof e ASCON-Xofa, o tamanho da saída pode ser escolhido conforme desejado.

3.9.1 Permutação no ASCON

O ASCON possui uma permutação $P_{ASCON(.)}$ de 320 bits com um número de rodadas N_r. Essas permutações aplicam, de forma iterativa, transformações $P_{ASCON(.)}$ baseadas em rede de substituição-permutação (SPN, do inglês *substitution–permutation network*) (FEISTEL, 1973), que consistem em três etapas (DOBRAUNIG et al., 2021b; KHAN; LEE; HWANG, 2021):

- **Etapa p_C - Adição de Constante**: a cada rodada de permutação, é adicionada uma constante de rodada RC à palavra de registro de 64 bits S_2 do estado S. É feita uma operação XOR da constante com os 8 bits menos significativos de S_2. Ou seja, $S_2 \leftarrow$: $S_2 \oplus RC$. A constante de rodada é adicionada nessa posição para permitir *pipelining* com as próximas operações ou com as anteriores. Essa etapa está ilustrada na Figura 7.

Figura 7 – Adição de Constante (p_C).

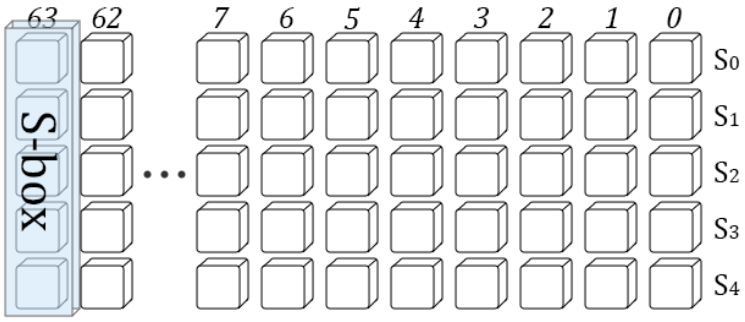


Fonte: Adaptado de (DOBRAUNIG et al., 2021b).

- **Etapa p_S - Camada de Substituição**: essa etapa é não-linear e atualiza o estado S com 64 aplicações paralelas da S-box $S(x)$ de 5 bits para cada *bit-slice* das cinco palavras de registro S_0, S_1, S_2, S_3 e S_4, em que S_0 tem os bits mais significativos e S_4 os bits menos significativos. O estado S é dividido em 64 fatias de 5 bits, embaralhando os bits de cada fatia, cada um na mesma posição de bit em todas as 5 palavras de estado, como ilustrado na Figura 8.

- **Etapa p_L - Camada de Difusão Linear**: essa etapa fornece uma difusão dentro de cada palavra de registro S_i de 64 bits, ao aplicar em cada uma dessas palavras uma função linear $\Sigma_i(S_i)$, descrita em (3.19). Dentro dessa função, é realizada uma operação XOR de S_i com

Figura 8 – Camada de Substituição (p_S).



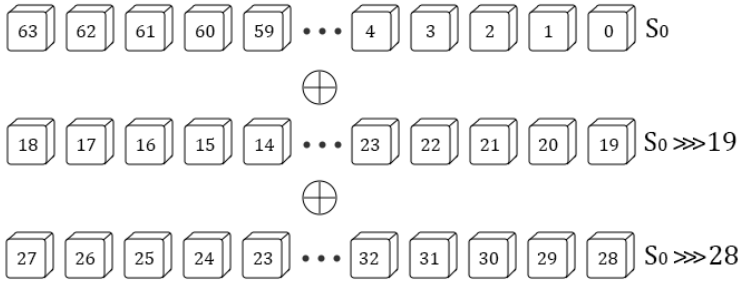
Fonte: Adaptado de (DOBRAUNIG et al., 2021b).

duas rotações à direita do seu valor. Estas rotações são rotações circulares para a direita por i bits, denotada por $\ggg$.

$$
\begin{aligned}
S_0 &\leftarrow \Sigma_0(S_0) = S_0 \oplus (S_0 \ggg 19) \oplus (S_0 \ggg 28) \\
S_1 &\leftarrow \Sigma_1(S_1) = S_1 \oplus (S_1 \ggg 61) \oplus (S_1 \ggg 39) \\
S_2 &\leftarrow \Sigma_2(S_2) = S_2 \oplus (S_2 \ggg 1) \oplus (S_2 \ggg 6) \\
S_3 &\leftarrow \Sigma_3(S_3) = S_3 \oplus (S_3 \ggg 10) \oplus (S_3 \ggg 17) \\
S_4 &\leftarrow \Sigma_4(S_4) = S_4 \oplus (S_4 \ggg 7) \oplus (S_4 \ggg 41).
\end{aligned}
\tag{3.19}
$$

Um exemplo de como essa etapa é realizada é ilustrado na Figura 9, na palavra de registro S_0.

Figura 9 – Camada de Difusão Linear (p_L) em S_0.



Fonte: Adaptado de (DOBRAUNIG et al., 2021b).

3.9.2 ASCON-128

O ASCON-128 é uma cifra baseada no modo de operação esponja duplex. Ele possui dois parâmetros recomendados, o ASCON-128 e o ASCON-128a, sendo o ASCON-128 mais recomendado pelos projetistas. O ASCON-128 trabalha com blocos de dados de 64 bits e $N_r = 6$ rodadas consecutivas de permutação $P_{ASCON(.)}$, e o ASCON-128a com blocos de dados de 128 bits e $N_r = 8$ (DOBRAUNIG et al., 2021b).

O processamento do ASCON-128, representado na Figura 10, é composto por quatro fases (ADOMNICAI; FOURNIER; MASSON, 2018; GROSS et al., 2017; KHAN; LEE; HWANG, 2021):

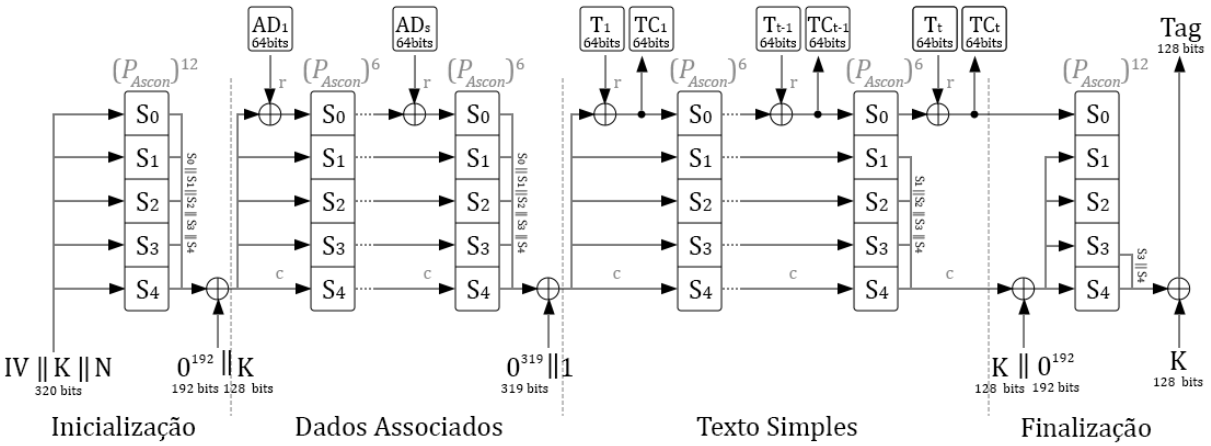
i. **Inicialização**: é construído um estado inicial de 320 bits a partir de uma chave de cifragem K de 128 bits, de um vetor de inicialização IV de 128 bits e de um número público de uso único, chamado de *nonce* N, também de 128 bits. O estado interno de 320 bits é representado por cinco palavras de registro de 64 bits, definidos como S_0, S_1, S_2, S_3 e S_4. Após a construção do estado, são aplicadas $N_r = 12$ rodadas de permutação $P_{ASCON(.)}$ e é feita uma operação XOR na chave K com os últimos 128 bits do estado S_3 e S_4. E com as palavras de registro restantes fazem um XOR com zeros.

ii. **Processamento de dados associados**: nessa fase, os dados associados DA_i são processados em blocos de $r = 64$ bits. Nos dados, é feito um *padding*, ou seja, é anexado um único 1 seguido pelo mínimo de zeros possíveis para obter um múltiplo de $r = 64$ bits para que esses dados sejam divididos em s blocos de 64 bits. Cada bloco é absorvido no estado a partir de uma operação XOR e em seguida são aplicadas $N_r = 6$ rodadas de permutação $P_{ASCON(.)}$ de forma iterada. Ao final do processamento do último bloco, é feita uma operação XOR com um único 1 seguido de 319 zeros com o novo valor do estado interno. Os dados associados são informações que não precisam ser confidenciais, mas de qualquer forma, não devem ser alteradas por um invasor. Caso não existam dados associados para serem processados, toda a etapa de dados associados pode ser omitida.

iii. **Processamento do texto claro**: da mesma forma que no processamento de dados associados, no texto claro, é feito um *padding* para que seu comprimento seja um múltiplo de $r = 64$ bits. O texto claro é então dividido em t blocos de 64 bits e cada bloco T_i é inserido no estado interno através de uma operação XOR com os primeiros 64 bits do estado. Após essa operação, são extraídos do estado interno um bloco 64 bits de texto cifrado TC_i e o estado é atualizado por 6 rodadas consecutivas de permutação. Esse processo é repetido até que T_{t-1} blocos de textos sejam processados. No último bloco, T_t, que não será feita essa operação. Ele tem seu tamanho truncado para que o tamanho total do texto cifrado seja o mesmo do texto claro sem o *padding*.

vi. **Finalização**: é feita uma operação XOR na chave de cifragem que está concatenada com 128 bits de zeros para o estado interno e em seguida são aplicadas 12 rodadas de permutação. Por fim, tem-se como resultado uma marcação chamada de *tag* de autenticação Tag de 128 bits. Essa *tag* é o resultado de uma operação XOR feita com os últimos 128 bits menos significativos do estado e a chave K.

Figura 10 – Processamento do ASCON.



Fonte: Adaptado de (DOBRAUNIG et al., 2021b).

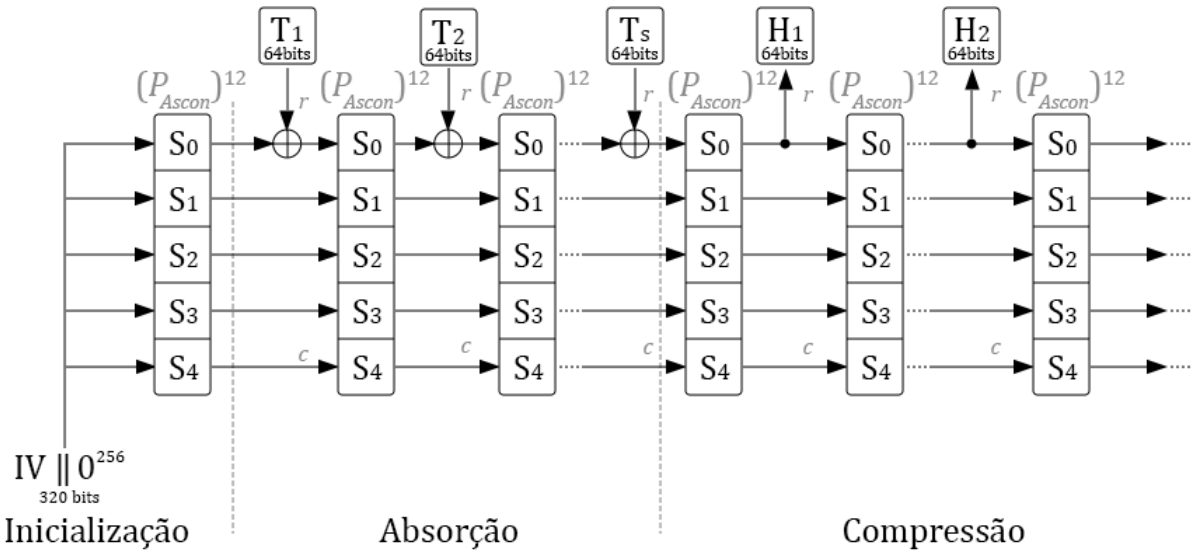
3.9.3 ASCON-*hash*

ASCON-*hash*, é uma função *hash* leve que também é baseada na construção esponja. Assim como na construção esponja, o ASCON-*hash* é composto pelas fases de inicialização, absorção e compressão, como está ilustrado na Figura 11 (KHAN; LEE; HWANG, 2022). Primeiramente, na fase de inicialização, é formado um estado inicial de 320 bits ao concatenar o vetor de inicialização IV de 64 bits com 256 bits de zeros. E em seguida são feitas 12 rodadas de permutação $P_{ASCON(.)}$.

Após esse processo, inicia-se a fase de absorção da mensagem, em que a mensagem T que computa o *hash* é processada, sendo convertida em uma *string* de bits que é um múltiplo de 64, e por fim é dividida em blocos de 64 bits, que serão processados um de cada vez. É então realizada uma operação XOR com o primeiro bloco de 64 bits da mensagem e os primeiros 64 bits do estado, seguida por 12 rodadas de permutação. Esse processo se repete até que todo o comprimento da mensagem seja processado.

Por fim, é iniciada a fase de compressão, em que será computado o *hash* de saída de 256 bits. O *hash* é extraído dos primeiros 64 bits do estado e em seguida são feitas 12 rodadas de permutação. A fase de compressão é concluída após serem feitas 4 extrações do *hash* de saída, cada uma delas com 64 bits de comprimento.

Figura 11 – Funcionamento do ASCON-*hash*.



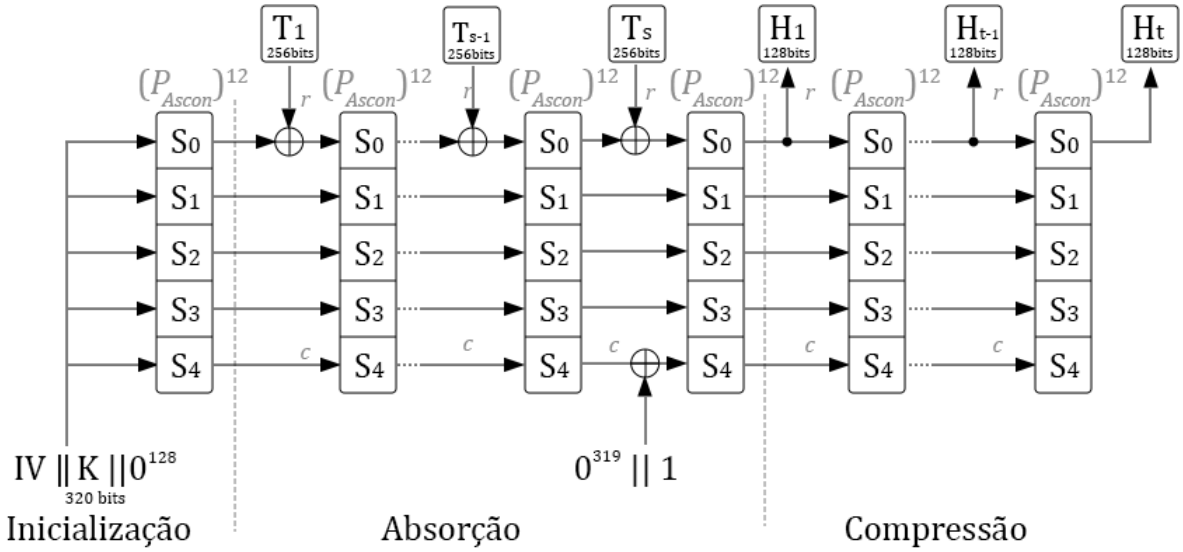
Fonte: Adaptado de (DOBRAUNIG et al., 2021b).

3.9.4 ASCON-PRF

Além das funcionalidades de cifragem e *hashing*, o ASCON também oferece uma função pseudoaleatória denominada ASCON-PRF (DOBRAUNIG et al., 2021a). Seu modo de operação semelhante ao do ASCON-*hash*, utiliza a construção esponja, como ilustrado na Figura 12. No entanto, na fase de inicialização, além do vetor de inicialização IV de 64 bits, seu estado interno também é formado por uma chave K de 128 bits e 128 bits de zeros. Assim como no ASCON-*hash*, o ASCON-PRF também apresenta um estado interno com 320 bits e em cada rodada a permutação $P_{ASCON(.)}$ é aplicada 12 vezes. No entanto, ele possui os parâmetros $r = 128$ e $c = 192$, e o tamanho da sua saída pode ser escolhido conforme desejado.

Na fase de absorção, a mensagem é processada e convertida em uma *string* que é um múltiplo de 256 bits. Essa *string* é então dividida em blocos de 256 bits. Cada bloco é integrado ao estado interno por meio de uma operação XOR, seguido por 12 rodadas de permutação. Após o processamento de todos os blocos, é iniciada a fase de compressão. Nesta fase, é gerada uma saída denominada *tag* (Tag). Isso é feito pela extração de blocos de 128 bits, que são seguidos por 12 rodadas de permutação. Exceto pelo último bloco, esse processo continua até que o comprimento de saída desejado seja alcançado.

Figura 12 – Funcionamento do ASCON-PRF.



Fonte: Adaptado de (DOBRAUNIG et al., 2021a).

Com o objetivo de fornecer uma visão abrangente dos parâmetros de todos os algoritmos analisados, todas as informações serão resumidas e organizadas na Tabela 6. Nela são exibidos os de tamanho de saída n, taxa de bits r e capacidade c. Em que d representa o comprimento de saída variável.

Tabela 6 – Parâmetros dos algoritmos SHA3, Quark, Photon, SPONGENT e ASCON.

Versão	Saída (n)	Taxa de bits (r)	Capacidade (c)
SHA3-224	224	1152	448
SHA3-256	256	1088	512
SHA3-384	384	832	768
SHA3-512	512	576	1024
SHAKE128	d	1344	256
SHAKE256	d	1088	512
U-Quark	136	8	128
D-Quark	176	16	160
S-Quark	256	32	224
C-Quark	384	64	320
Photon-80/20/16	80	20	60
Photon-128/16/16	128	16	112
Photon-160/36/36	160	36	124
Photon-224/32/32	224	32	192
Photon-256/32/32	256	32	224
SPONGENT-$88/80/8$	88	8	80
SPONGENT-$88/176/88$	88	88	176
SPONGENT-$128/128/8$	128	8	128
SPONGENT-$128/256/128$	128	128	256
SPONGENT-$160/160/16$	160	16	160
SPONGENT-$160/160/80$	160	80	160
SPONGENT-$160/320/160$	160	160	240
SPONGENT-$224/224/16$	224	16	224
SPONGENT-$224/224/112$	224	112	224
SPONGENT-$224/448/224$	224	224	448
SPONGENT-$256/256/16$	256	16	256
SPONGENT-$256/256/128$	256	128	256
SPONGENT-$256/512/256$	256	256	512
ASCON-128	d	64	256
ASCON-*hash*	256	64	256
ASCON-PRF	d	128	192

Fonte: (DWORKIN, 2015; AUMASSON et al., 2013; GUO; PEYRIN; POSCHMANN, 2011; BOGDANOV et al., 2011; DOBRAUNIG et al., 2021a; DOBRAUNIG et al., 2021b).

3.10 CONSIDERAÇÕES

Em síntese, neste capítulo, foi explorada a primitiva criptográfica construção esponja juntamente com sua revisão bibliográfica. Além disso, foram estudadas as premissas criptográficas que a utilizam, discutindo como são estruturadas, seus níveis de segurança e possíveis cenários de implementação.

4 CONSTRUÇÃO DE UM BLOCO DE PERMUTAÇÃO A PARTIR DA TRANSFORMADA DO COSSENO SOBRE CORPOS DE CARACTERÍSTICA DOIS

Neste capítulo, são apresentadas as metodologias desenvolvidas para criar e implementar um bloco de permutação. Este bloco foi projetado com o propósito de ser empregado em algoritmos criptográficos, de cifragem e *hash*, que utilizam a construção esponja, visando fornecer uma segurança apropriada para esses sistemas. Para alcançar os objetivos estabelecidos nesta pesquisa, foram analisados alguns algoritmos de criptografia, todos baseados na construção esponja. Entre eles, destacam-se o SHA3, Quark, SPONGENT e ASCON.

Inicialmente, será apresentada a estrutura e o funcionamento do bloco de permutação proposto, construído utilizando a transformada do cosseno sobre corpos de característica 2 do tipo 1 (LIMA; BARONE; CAMPELLO DE SOUZA, 2016), juntamente com um polinômio de permutação (NIU et al., 2020). Além disso, será discutido como é feita a seleção dos parâmetros β's, bem como os testes iniciais realizados para avaliar a aleatoriedade e a eficácia do bloco de permutação em cada algoritmo. Especificamente, é explorado o uso do efeito avalanche como métrica para determinar se o novo bloco de permutação é capaz de proporcionar o nível adequado de aleatoriedade e segurança exigidos pelos algoritmos criptográficos.

4.1 EFEITO AVALANCHE

O efeito avalanche é uma propriedade fundamental em algoritmos criptográficos, essencial para garantir a segurança e aleatoriedade dos dados (UPADHYAY et al., 2022). Essa propriedade refere-se à capacidade de um algoritmo criptográfico produzir um resultado significativamente diferente para entradas levemente diferentes. Em outras palavras, uma pequena alteração na entrada, como a mudança de um único bit, deve resultar em uma mudança significativa no valor de saída do algoritmo. É consenso que aproximadamente metade dos bits de saída são alterados quando há uma mudança mínima na entrada (WEBSTER; TAVARES, 1985; UPADHYAY et al., 2022). Matematicamente, o efeito avalanche é representado por:

$$\text{Efeito Avalanche} = \frac{\text{Hamming}(H(\mathbf{x}), H(\mathbf{x}'))}{l}\,, \tag{4.1}$$

em que x e x' representam as mensagens de entrada, Hamming() é a distância de Hamming, que conta os pares de bits diferentes entre duas *strings*, H é a função *hash* e l é o tamanho da saída do *hash*. Na Equação (4.1), uma taxa de alteração de $0{,}50$ indica um efeito de avalanche adequado. No entanto, de acordo com (ASTUTI; ARFIANI; ARIBOWO, 2019), uma faixa de $0{,}45 - 0{,}60$ ainda é considerada um bom resultado.

4.2 CONSTRUÇÃO DO BLOCO DE PERMUTAÇÃO

Para criar uma permutação, foram empregadas dois métodos distintos: uso de polinômios de permutação (NIU et al., 2020) e a transformada do cosseno sobre corpos de característica 2 do tipo 1 (LIMA; BARONE; CAMPELLO DE SOUZA, 2016). A combinação dessas funções é apresentada no Algoritmo 2, em que ambas são definidas sobre $\mathbb{F}_{2^8}$. Essas funções foram escolhidas por serem involuções, ou seja, são iguais às suas próprias inversas. Isso simplifica sua implementação em algoritmos criptográficos, pois elimina a necessidade de calcular funções inversas. Além disso, o corpo finito $\mathbb{F}_{2^8}$ foi selecionado por possuir 256 possíveis valores, incluindo o zero, e cada elemento pode ser representado por um byte (8 bits), o que se alinha com a arquitetura dos computadores atuais.

Nesse algoritmo, $f(.)$ denota a função de permutação que emprega o polinômio de permutação de (2.4), enquanto $\mathbf{C}$ representa a matriz de transformação da FFCT-I conforme definida em (2.9). No início do processo, o bloco recebe como entrada um vetor binário $\mathbf{I}$ de comprimento $b = (2 + n) \times 32$, em que $n \in \mathbb{N}^*$. O comprimento b foi definido dessa forma para garantir que o estado interno tenha, no mínimo, 3×32 bits, assegurando que seja um múltiplo de 32. Isso permite que a matriz de transformação $\mathbf{C}$ seja aplicada a cada bloco de 32 bits em conjunto com outros dois blocos distintos. Em seguida, a função de permutação $f(.)$ é aplicada para reorganizar os elementos do vetor, transformando cada bloco de 8 bits em outro bloco de 8 bits. Essa aplicação é feita para introduzir confusão e não-linearidade no sistema. Após a aplicação do polinômio de permutação, a matriz $\mathbf{C}$ transforma um bloco de $8 \times 8 = 64$ bits, realizando uma sobreposição de 32 bits em cada aplicação para gerar difusão.

A difusão é o processo de reorganização do texto claro em texto cifrado, garantindo que mesmo pequenas mudanças no texto claro causem grandes mudanças no texto cifrado. Já a confusão visa tornar a relação entre o texto claro e o texto cifrado mais complexa, dificultando a previsão do texto claro a partir do texto cifrado. Um exemplo comum de confusão é a substituição de caracteres em um texto.

Algoritmo 2: Bloco de Permutação $P(.)$ Proposto

Input : $\mathbf{I} \rightarrow$ vetor binário de comprimento b.
Output : $\mathbf{I}' \rightarrow$ vetor binário de comprimento b.
begin
  $\mathbf{I}' \leftarrow \mathbf{I}$
  for $n \leftarrow 0$ **to** $(b/8 - 1)$ **do**
    $\mathbf{I}'[8n : 8n + 7] \leftarrow f(\mathbf{I}'[8n : 8n + 7])$
  end
  for $n \leftarrow 0$ **to** $(b/32 - 1)$ **do**
    $\mathbf{I}'[32n : 32n + 63 (\text{mod } b)] \leftarrow \mathbf{C} \times (\mathbf{I}'[32n : 32n + 63 (\text{mod } b)])$
  end
end

O estado interno foi estruturado de forma que fosse representado por um vetor unidimensional $(M \times 1)$, em que M representa o número de linhas. Cada elemento desse vetor possui 8 bits. Seguindo para o início da criação do bloco de permutação, foi necessário criar o corpo finito $\mathbb{F}_{2^8}$. A partir desse corpo, foram desenvolvidas uma tabela que mapeia números de 0 a 255 em polinômios pertencentes a esse corpo finito, e outra que realiza a conversão inversa, de polinômio para número natural.

A partir desse corpo finito, também foram criados os β's e seus respectivos k's, obedecendo ao Corolário 2.1. Para aplicar a equação de polinômios de permutação (2.4), é essencial selecionar o parâmetro β pertencente ao conjunto $\mu_{17} \in \mathbb{F}_{2^8}^*$, ou seja, elementos de ordem 17 pertencentes a esse corpo finito, juntamente com seu respectivo k, em que $1 \leq k \leq 8$. Essa escolha deve assegurar que a relação $\beta^2\gamma^{(q-1)k} = 1$ seja satisfeita, assegurando que a equação seja uma involução. Na Tabela 7, estão listados os β's e seus respectivos k's que satisfazem essa relação, sendo $\gamma = \alpha$ um elemento gerador do corpo $\mathbb{F}_{2^8}$.

Tabela 7 – Lista de β e k que satisfazem a relação $\beta^2\gamma^{(q-1)k} = 1$.

nº	k	β
0	7	$\alpha^3 + \alpha^2 + \alpha + 1$
1	5	$\alpha^7 + \alpha^6 + \alpha^4 + \alpha^3 + \alpha^2 + \alpha + 1$
2	3	$\alpha^4 + \alpha^3 + \alpha$
3	1	$\alpha^5 + \alpha^4 + \alpha^3 + \alpha + 1$
4	6	$\alpha^6 + \alpha^4 + \alpha^3 + 1$
5	4	$\alpha^5 + \alpha^2$
6	2	$\alpha^5 + \alpha^3 + \alpha^2$

Fonte: Autora.

Cada β foi utilizado individualmente em (2.4) para gerar tabelas que contêm todos os elementos do corpo finito com a aplicação dessa equação. Isso foi feito de forma iterativa, aplicando a equação para cada combinação de β e k, um de cada vez, nos 256 elementos do corpo finito $\mathbb{F}_{2^8}$.

Após a aplicação do polinômio de permutação em cada um dos elementos, cria-se a matriz de transformação **C** a partir do corpo finito $\mathbb{F}_{2^8}$, em que sua ordem é ord(α) = 255. Os elementos da matriz C são calculados usando (2.10) em que $\zeta = \alpha^{15}$, com ordem ord(ζ) = ord(α^{15}) $= 2N - 1 = 17$. Dessa forma, é gerada uma matriz de tamanho $N - 1 = 8$.

Dando continuidade, a matriz de transformação **C** é aplicada nos novos elementos obtidos por (2.4). Ao longo do comprimento do vetor, é realizado um *overlapping* para promover uma difusão mais eficaz dos bits. No *overlapping*, cada bloco contêm 64 bits e a transformada vai sendo aplicada a um intervalo de 32 bits. A transformada do cosseno é então aplicada em grupos de 8 elementos por vez, multiplicando a matriz de transformação **C** (8×8) por um vetor v_n (8×1), resultando em um vetor V_n (8×1). O próximo conjunto de elementos a ser transformado

é composto pelos 4 últimos elementos do conjunto anterior e os 4 elementos subsequentes. Na última aplicação do *overlapping* o conjunto é formado pelos 4 últimos elementos do vetor seguidos pelos 4 primeiros. Esse processo de aplicação da transformada **C** ocorre $b/32$ vezes. Dada a sobreposição proposta, é esperado que após a aplicação o bloco de permutação por duas vezes a informação contida em cada bloco tenha interferido em todos os demais blocos.

4.3 IMPLEMENTAÇÃO DO BLOCO DE PERMUTAÇÃO

A implementação do bloco de permutação envolveu a seleção das versões SHA3-224, SHA3-256, SHA3-384, SHA3-512, S-Quark, SPONGENT-$128/256/128$, SPONGENT-$160/320/160$, SPONGENT-$224/448/224$, SPONGENT-$256/256/128$, SPONGENT-$256/512/256$, ASCON-*hash* e ASCON-128. Esses algoritmos foram selecionados considerando que o tamanho do estado interno b precisa ser múltiplo de 32, para garantir a aplicação do bloco proposto.

Para ajustar os algoritmos selecionados ao novo bloco de permutação, substituindo seus blocos de permutação originais, foi essencial converter seus estados internos em uma representação adequada. Isso implicou na conversão de todos os estados internos em um vetor $(M \times 1)$, em que cada elemento é obtido a partir da divisão do estado interno em blocos de 8 bits.

O algoritmo SHA3 emprega um estado interno de 1600 bits, organizado em uma estrutura tridimensional de dimensões $5 \times 5 \times 64$ bits. Este estado é então transformado em 25 sequências de 64 bits, conhecidas como faixas da matriz de estados. Para a integração do novo bloco de permutação no algoritmo SHA3, é necessário converter essas faixas em uma matriz de 200 elementos de 8 bits. Para o algoritmo Quark, que utiliza um *array* de bits para representar seu estado interno, cada bloco de 8 bits é convertido em um inteiro e organizado no vetor correspondente. No caso do SPONGENT, em que o estado interno é representado por um único inteiro, que pode ter tamanhos 384, 480, 672 ou 768 bits, esse inteiro é dividido em M partes de 8 bits, e então armazenado no vetor. Já para o ASCON, os 64 bits de cada uma das 5 partes que compõem o estado interno são convertidos em 8 elementos de 8 bits cada, totalizando 40 elementos de 8 bits. Para todos os algoritmos citados, os elementos do estado interno são transformados em polinômios pertencentes ao corpo finito $\mathbb{F}_{2^8}$ e organizados em um vetor coluna de dimensão $M \times 1$. Essa adaptação permite a integração bem-sucedida do bloco de permutação aos algoritmos.

O vetor $(M \times 1)$ resultante é a nova representação do estado interno, no qual são aplicadas as funções que compõem o bloco de permutação. Após cada rodada de aplicação dessas funções, o vetor é transformado para restaurar a forma original do estado interno de cada algoritmo. De maneira concisa, no bloco de permutação, os elementos do vetor são convertidos em polinômios, nos quais a função de permutação polinomial é aplicada. E por fim, a transformada do cosseno é aplicada em blocos de 8 elementos/polinômios por vez, com um *overlapping* ao longo do seu comprimento.

4.4 ESCOLHA DO β

Cada β foi submetido a um teste individual, aplicando-o na equação do polinômio de permutação, que, em conjunto com a transformada do cosseno, compõe o bloco de permutação. Ao implementar esse bloco nos algoritmos SHA3, Quark, SPONGENT e ASCON, foi realizado o cálculo do *hash* para cada um dos β's.

Para calcular o *hash* de cada algoritmo, foi utilizada a mensagem de entrada "EMBARCADOS", que possui 80 bits na codificação ASCII. Em seguida, invertendo um bit por vez, o *hash* é calculado novamente, totalizando em 80 *hashes* diferentes para cada algoritmo. Durante esses testes, foi aplicada a função de efeito avalanche da mensagem original a cada *hash* resultante das alterações de um único bit. O critério de seleção do β mais adequado foi baseado na análise do efeito que cada β produziu nos *hashes* calculados. Assim foi selecionado o β que proporcionasse uma dispersão significativa nos bits do *hash*, mantendo-o dentro de uma faixa de variação entre 45% e 60%, abrangendo tanto o maior mínimo quanto o menor máximo possível. Nas Tabelas 8 a 11 apresentadas a seguir, são exibidos os testes realizados com todos os β's nas versões selecionadas dos algoritmos, apresentando valor médio, desvio padrão σ, valor máximo e mínimo ao calcular o efeito avalanche.

Tabela 8 – Cálculo do efeito avalanche para todos os β's implementados em versões do SHA3.

Versão	SHA3-224				SHA3-256			
β	Média	σ	Máx	Mín	Média	σ	Máx	Mín
0	0,4993	0,0322	0,5670	0,4196	0,5060	0,0335	0,5898	0,4336
1	0,4998	0,0331	0,5714	0,4286	0,5012	0,0341	0,5938	0,4180
2	0,5038	0,0358	0,5982	0,4152	0,4993	0,0353	0,5781	0,4258
3	0,4978	0,0359	0,5670	0,4107	0,4960	0,0332	0,5781	0,4102
4	0,4998	0,0352	0,5714	0,4241	0,5002	0,0315	0,5664	0,4297
5	0,5017	0,0356	0,5982	0,4107	0,5020	0,0295	0,5859	0,4336
6	0,4993	0,0350	0,5848	0,4330	0,5030	0,0350	0,5977	0,4141
Versão	SHA3-384				SHA3-512			
β	Média	σ	Máx	Mín	Média	σ	Máx	Mín
0	0,4988	0,0274	0,5573	0,4375	0,5038	0,0247	0,5684	0,4434
1	0,5009	0,0297	0,5651	0,4089	0,5038	0,0250	0,5645	0,4414
2	0,5013	0,0310	0,5651	0,4219	0,5042	0,0215	0,5469	0,4453
3	0,5024	0,0292	0,5833	0,4323	0,5011	0,0265	0,5762	0,4277
4	0,4997	0,0292	0,5599	0,4375	0,5007	0,0243	0,5762	0,4492
5	0,5011	0,0289	0,5703	0,4323	0,4991	0,0221	0,5469	0,4531
6	0,4970	0,0316	0,5729	0,4141	0,5011	0,0249	0,5508	0,4199

Fonte: Autora.

Tabela 9 – Cálculo do efeito avalanche para todos os β's implementados em versões do SPONGENT.

Versão	SPONGENT-128/256/128				SPONGENT-160/320/160			
β	Média	σ	Máx	Mín	Média	σ	Máx	Mín
0	0,5004	0,0423	0,5780	0,4063	0,5030	0,0346	0,5813	0,4188
1	0,5055	0,0496	0,6094	0,3984	0,4959	0,0418	0,6125	0,4063
2	0,4949	0,0399	0,6094	0,3828	0,4973	0,0359	0,5750	0,3813
3	0,4977	0,0408	0,6016	0,4141	0,5026	0,0405	0,5938	0,3813
4	0,4881	0,0428	0,5938	0,3828	0,5042	0,0393	0,5938	0,4125
5	0,5009	0,0479	0,6328	0,3516	0,4945	0,0377	0,5813	0,4063
6	0,4926	0,0431	0,6172	0,3906	0,5001	0,0386	0,6063	0,4188
Versão	SPONGENT-224/448/224				SPONGENT-256/256/128			
β	Média	σ	Máx	Mín	Média	σ	Máx	Mín
0	0,5056	0,0369	0,6205	0,4196	0,4990	0,0360	0,5977	0,4219
1	0,5040	0,0365	0,5893	0,4063	0,5037	0,0334	0,5625	0,4102
2	0,5032	0,0305	0,5893	0,4420	0,4964	0,0273	0,5508	0,4297
3	0,5016	0,0384	0,5759	0,4107	0,4979	0,0310	0,5586	0,4258
4	0,5064	0,0367	0,6071	0,4107	0,4945	0,0322	0,5820	0,4258
5	0,4982	0,0370	0,6295	0,4107	0,5035	0,0367	0,6016	0,4219
6	0,4960	0,0339	0,5580	0,3929	0,4999	0,0310	0,5664	0,4258
Versão	SPONGENT-256/512/256							
β	Média	σ	Máx	Mín				
0	0,5022	0,0310	0,5630	0,4258				
1	0,5026	0,0364	0,5898	0,4141				
2	0,5003	0,0289	0,5547	0,4102				
3	0,4984	0,0322	0,5625	0,4258				
4	0,4989	0,0376	0,5859	0,4063				
5	0,5023	0,0269	0,5742	0,4297				
6	0,5000	0,0300	0,5742	0,4414				

Fonte: Autora.

Após analisar os valores de efeito avalanche nas tabelas fornecidas, e considerando o β que melhor se adequa à faixa de $0,45 - 0,60$, os β's selecionados para cada versão estão listados na Tabela 12.

Os β's escolhidos serão integrados ao bloco de permutação de cada algoritmo. Posteriormente, testes serão conduzidos para avaliar a segurança e a eficácia do bloco de permutação com os β's selecionados para cada algoritmo de função *hash*.

Tabela 10 – Cálculo do efeito avalanche para todos os β's implementados no algoritmo S-Quark.

β	Média	σ	Máximo	Mínimo
0	0,5047	0,0305	0,5781	0,4258
1	0,5016	0,0320	0,5781	0,4375
2	0,4990	0,0317	0,5664	0,3945
3	0,4955	0,0329	0,5703	0,4219
4	0,4973	0,0296	0,5898	0,4336
5	0,5063	0,0296	0,5781	0,4414
6	0,4944	0,0291	0,5664	0,4180

Fonte: Autora.

Tabela 11 – Cálculo do efeito avalanche para todos os β's na versão ASCON-*hash*.

β	Média	σ	Máximo	Mínimo
0	0,4958	0,0349	0,5781	0,4219
1	0,5079	0,0350	0,6055	0,4219
2	0,5024	0,0287	0,5664	0,4336
3	0,5019	0,0340	0,5859	0,4023
4	0,5025	0,0326	0,5781	0,4297
5	0,5074	0,0294	0,5977	0,4492
6	0,5003	0,0304	0,5703	0,4375

Fonte: Autora.

Tabela 12 – β's escolhidos para cada algoritmo.

n^o	k	β	**Algoritmos**
0	7	$\alpha^3+\alpha^2+\alpha+1$	SHA3-384, SPONGENT-$160/320/160$
1	5	$\alpha^7+\alpha^6+\alpha^4+\alpha^3+\alpha^2+\alpha+1$	
2	3	$\alpha^4+\alpha^3+\alpha$	SPONGENT-$224/448/224$, SPONGENT-$256/256/128$
3	1	$\alpha^5+\alpha^4+\alpha^3+\alpha+1$	SPONGENT-$128/256/128$
4	6	$\alpha^6+\alpha^4+\alpha^3+1$	
5	4	$\alpha^5+\alpha^2$	SHA3-256, SHA3-512, S-Quark, ASCON-*hash*
6	2	$\alpha^5+\alpha^3+\alpha^2$	SHA3-224, SPONGENT-$256/512/256$

Fonte: Autora.

4.5 CONSIDERAÇÕES

Neste capítulo, foi apresentado o bloco de permutação proposto e sua implementação nos algoritmos SHA3, Quark, SPONGENT e ASCON, além dos valores de β possíveis para a função do polinômio de permutação. Além disso, foram realizados testes para verificar a propriedade do efeito avalanche do novo bloco de permutação, quando substituído nas funções *hashes* testadas.

5 IMPLEMENTAÇÕES E RESULTADOS

Neste capítulo, serão apresentados os resultados obtidos ao integrar o bloco de permutação nos algoritmos criptográficos SHA3, Quark, SPONGENT e ASCON, para avaliar a segurança de *hash* e cifragem. Foram conduzidos testes para avaliar o efeito avalanche em cada versão dos algoritmos e cálculo de entropia para a cifra, a fim de fornecer uma compreensão inicial da aleatoriedade e segurança oferecidas pelo bloco de permutação proposto quando em substituição ao utilizado em cada um dos algoritmos mencionados. Também são explorados os conceitos de difusão e confusão, propriedades de segurança e tipos de ataques, utilizando o capítulo 11 e Apêndice E do livro (STALLINGS, 2023).

Os experimentos foram realizados em uma máquina com as seguintes especificações: sistema operacional Windows 10, processador Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz 2.90 GHz e 8GB de RAM. O desenvolvimento da solução foi realizado no programa Visual Studio Code, versão 1.86.2, utilizando a linguagem de programação Python 3.10.12 e a biblioteca Sagemath, versão 9.5, para uma manipulação mais eficiente da álgebra sobre corpos finitos. Os seguintes códigos foram utilizados como base: SHA3 (ASSCHE, 2024), SPONGET (RIJNEVELD, 2014), S-Quark (SOUSSI, 2020) e ASCON (EICHLSEDER; VISSOULTCHEV; FAZ, 2023).

5.1 ATAQUES

A segurança em sistemas criptográficos é crucial para proteger informações confidenciais contra acessos não autorizados. Para garantir essa segurança, os algoritmos criptográficos devem empregar duas técnicas principais: difusão e confusão (NABEEL; HABAEBI; ISLAM, 2021). Como mencionado em 4.2, a difusão é o processo de espalhar a influência de cada bit do texto claro por todo o texto cifrado. Isso significa que cada bit do texto cifrado depende de muitos bits do texto claro, dificultando a análise do texto cifrado para descobrir o texto claro original. Enquanto que na confusão, caracteres no texto claro são substituídos por outros caracteres de uma maneira que parece aleatória, de modo que conhecer o texto cifrado não fornece informações diretas sobre o texto claro.

Para serem consideradas seguras, as funções *hash* criptográficas devem atender a requisitos específicos. Esses requisitos incluem tamanho de entrada variável, tamanho de saída fixo, eficiência, resistência à pré-imagem, segunda pré-imagem e colisão, além de pseudoaleatoriedade (STALLINGS, 2023). As duas primeiras propriedades garantem que a função *hash* possa processar entradas de diferentes tamanhos, sempre produzindo uma saída de tamanho fixo. A eficiência assegura que a função seja prática em várias aplicações, enquanto as propriedades de resistência são cruciais para prevenir ataques onde a mensagem é interceptada, decifrada

e/ou alterada. A pseudoaleatoriedade é fundamental para gerar saídas aleatórias e imprevisíveis, sendo também importante na geração de chaves criptográficas e números pseudoaleatórios, bem como na garantia da integridade dos dados.

Os ataques à função *hash* são divididos em duas categorias: ataques de força bruta e criptoanálise. Ataque de força bruta é um ataque criptoanalítico que depende apenas do tamanho da saída do *hash* em bits. Ele envolve um procedimento exaustivo para testar todas as possibilidades, uma por uma. Exemplos de ataque de força bruta são ataque de colisão e ataque de pré-imagem e segunda pré-imagem. Já a criptoanálise é um ataque que se baseia em pontos fracos de um algoritmo criptográfico específico (STALLINGS, 2023). A criptoanálise é o estudo de técnicas matemáticas utilizadas para tentar derrotar técnicas criptográficas. Em outras palavras, são técnicas usadas para tentar decifrar um texto cifrado (MENEZES, 1997). Como exemplos existem a criptoanálise diferencial (CHAN et al., 2023), criptoanálise linear (XU et al., 2024), ataque de cubo (HE et al., 2020), entre outros (DING et al., 2019; GAO; ZHOU, 2021; DING et al., 2024).

5.1.1 Ataque de Colisão

Um ataque de colisão se baseia na probabilidade de encontrar resultados idênticos ao selecionar um elemento de forma aleatória de um conjunto finito S. Com base no paradoxo do aniversário, essas repetições ocorrem depois de aproximadamente $\sqrt{|S|}$, em que $|S|$ denota o tamanho do conjunto S. Isso significa que, à medida que o conjunto de elementos aumenta em tamanho, as chances de encontrar repetições também aumentam. Como o próprio nome sugere, este ataque é utilizado para encontrar colisões em funções *hash* criptográficas (TILBORG; JAJODIA, 2011). Para uma função *hash* com uma saída resultante em n bits, necessita-se de aproximadamente $2^{n/2}$ execuções da função *hash*. A seguir será apresentado como que se chega a esse resultado, com uma explicação mais completa descrita em (STALLINGS, 2023).

O paradoxo do aniversário afirma que, em um grupo de apenas 23 pessoas selecionadas aleatoriamente, a probabilidade de pelo menos duas delas compartilharem o mesmo aniversário é superior a 50%. Segue de:

$$P(365, k) = 1 - \frac{365!}{(365-k)!(365)^k}. \tag{5.1}$$

Substituindo k por 23, tem-se que P(365,23) = 0,5073, indicando que a probabilidade de pelo menos duas pessoas compartilharem o mesmo aniversário em um grupo de 23 pessoas é de cerca de 50,73$\%$. Além disso, o paradoxo de aniversário pode ser generalizado da seguinte forma: Ao considerar uma variável aleatória que pode assumir qualquer valor inteiro de 1 a n com igual probabilidade, e ao selecionar k valores distintos dessa variável aleatória, em que $k \leq n$, surge o interesse em determinar a probabilidade $P(n, k)$ de haver pelo menos uma coincidência entre os k valores selecionados. Dito isso, tem-se

$$P(n, k) = 1 - \frac{n!}{(n-k)!(n)^k}. \tag{5.2}$$

que calcula essa probabilidade de forma mais geral. (5.2) pode ser reescrita da forma

$$\begin{aligned} P(n,k) &= 1 - \frac{n \times (n-1) \times ... \times (n-k+1)}{n^k}, \\ &= 1 - \left[\frac{n-1}{n} \times \frac{n-2}{n} \times ... \times \frac{n-k+1}{n}\right], \\ &= 1 - \left[\left(1-\frac{1}{n}\right) \times \left(1-\frac{2}{n}\right) \times ... \times \left(1-\frac{k-1}{n}\right)\right]. \end{aligned} \tag{5.3}$$

Dando continuidade, usando a inequação $(1-x) \leq e^{-x}$, para todo $x \geq 0$, tem-se que,

$$\begin{aligned} P(n,k) &> 1 - [(e^{-1/n}) \times (e^{-2/n}) \times ... \times (e^{-(k-1)/n})], \\ &> 1 - e^{-[(1/n)+(2/n)+...+((k-1)/n)]}, \\ &> 1 - e^{-(k\times(k-1))/2n}. \end{aligned} \tag{5.4}$$

Disso, é desejado saber qual valor de k é necessário para que $P(n,k) > 0{,}5$. Daí, substituindo esse valor em (5.4), tem-se que,

$$\begin{aligned} 1/2 &= 1 - e^{-(k\times(k-1))/2n}, \\ 2 &= e^{(k\times(k-1))/2n}, \\ ln(2) &= \tfrac{k\times(k-1)}{2n}. \end{aligned} \tag{5.5}$$

A expressão $k \times (k-1)$ pode ser substituída por k^2, quando k for grande. Logo, (5.5) resulta em

$$k = \sqrt{2(ln2)n} = 1.18\sqrt{n} \approx \sqrt{n}. \tag{5.6}$$

Para entender a base de um ataque de aniversário, considere uma função H, com 2^m saídas possíveis. Ao aplicar H a k entradas aleatórias, é necessário determinar o valor de k necessário para que haja uma probabilidade de pelo menos duas dessas entradas produzirem o mesmo valor de $hash$, ou seja, uma colisão $H(x) = H(y)$. Do resultado de (5.6) é obtido

$$k = \sqrt{2^m} = 2^{m/2}. \tag{5.7}$$

Da Equação (5.7), conclui-se que para uma função com 2^m possíveis saídas, é esperado encontrar uma colisão após $2^{m/2}$ entradas aleatórias diferentes. É interessante observar que, quanto maior o valor de m, maior será o número de tentativas necessárias para encontrar uma colisão, tornando a função *hash* mais segura contra ataques de colisão.

5.1.2 Ataque de Pré-imagem e Segunda Pré-imagem

Um ataque de pré-imagem ou segunda pré-imagem tem como objetivo encontrar um valor y tal que $H(y)$ seja igual a um dado valor de *hash* h, isto é, encontrar uma pré-imagem a

partir do *hash* de saída. Para realizar esse ataque, o método mais comum é usar força bruta, em que o atacante escolhe valores de y aleatoriamente e testa cada um deles até encontrar um que gere o valor de *hash* desejado h, em outras palavras, até encontrar uma colisão. Para isso, um atacante precisaria testar 2^{m-1} valores de y até encontrar um que gere h (STALLINGS, 2023).

Esse valor é determinado através de um problema geral relacionado a uma função *hash*, conforme discutido em (STALLINGS, 2023). Considerando uma função *hash* H, com n possíveis saídas e um valor específico de $H(x)$, o objetivo é determinar o número de k entradas aleatórias necessárias para que haja uma probabilidade de 0,5 de que pelo menos uma entrada y satisfaça $H(y) = H(x)$.

A probabilidade de um único valor de y satisfazer a condição $H(y) = H(x)$ é de apenas $1/n$. Por outro lado, a probabilidade quando $H(y) \neq H(x)$ é de $[1 - (1/n)]$. Ao gerar k valores aleatórios de y, a probabilidade é de $[1 - (1/n)]^k$ quando nenhum deles coincidem e de $1 - [1 - (1/n)]^k$ quando existe pelo menos uma correspondência. Dando continuidade, o Teorema Binomial pode ser expresso como

$$(1 - a)^k = 1 - ka + \frac{k(k-1)}{2!}a^2 - \frac{k(k-1)(k-2)}{3!}a^3... \tag{5.8}$$

Essa expressão pode ser aproximada como $(1 - ka)$, para valores de a muito pequenos. Logo, $1 - [1 - (1/n)]^k \approx 1 - [1 - (k/n)] = k/n$ é a probabilidade de acontecer pelo menos uma correspondência. Por conseguinte, para uma probabilidade de 0,5, $k = n/2$.

Para um *hash* com m-bits de tamanho, existem 2^m combinações possíveis. Dessa forma, o valor de k que produz uma probabilidade de 0,5 é $k = 2^{(m-1)}$. Isso significa que metade do número total de combinações possíveis precisaria ser testada para ter uma chance razoável de encontrar uma correspondência entre uma entrada y e um valor de *hash* $H(y)$ que seja igual a um valor de *hash* $H(x)$ pré-determinado.

5.1.3 Resultado dos Testes de Resistência

Foram realizados testes de resistência para avaliar a robustez dos algoritmos de *hash* frente aos ataques de pré-imagem, segunda pré-imagem e colisão, conforme abordado nesta seção. No teste de resistência à pré-imagem, para cada algoritmo, um *hash* inicial é gerado a partir de uma entrada aleatória. Em seguida, são calculados 10000 *hashes* adicionais, utilizando entradas também aleatórias, com o objetivo de encontrar alguma correspondência com o *hash* original. A não ocorrência de correspondência em nenhum dos 10000 resultados com o *hash* original é um critério para a aprovação do algoritmo neste teste.

No teste de resistência à segunda pré-imagem, o *hash* é computado a partir da *string* de entrada “EMBARCADOS”. Posteriormente, 10000 entradas distintas de “EMBARCADOS” são geradas na tentativa de encontrar uma entrada que produza o mesmo *hash* final. E por último, no teste de resistência à colisão, duas entradas aleatórias são geradas e seus *hashes* são calculados.

Esse processo é repetido 10000 vezes, e os *hashes* resultantes são comparados para detectar colisões.

Todos os algoritmos, tanto aqueles que utilizam o novo bloco de permutação quanto os originais, demonstraram sucesso no conjunto de testes.

5.2 ENTROPIA

Entropia é um parâmetro estatístico que mede a incerteza ou imprevisibilidade de um conjunto de dados (SHANNON; WEAVER, 1949; CACHIN, 1997). A entropia H de uma variável aleatória x é dada por:

$$H = -\sum Prob(x) \log_2 (Prob(x)), \tag{5.9}$$

em que $Prob(x)$ é a probabilidade de ocorrência da variável x. Em algoritmos criptográficos, é desejado que a entropia seja alta, pois uma baixa entropia pode indicar falhas a serem exploradas para a quebra do sistema criptográfico (STALLINGS, 2023).

Foram feitos testes de entropia para avaliar a qualidade das saídas geradas pelo bloco proposto e também para compará-las com as saídas da permutação original do ASCON-128. No bloco proposto, apenas duas rodadas da permutação foram aplicadas, enquanto que na versão original o número de rodadas permaneceu inalterado. Para o cálculo da entropia, foi utilizada a Equação (5.9), em que x representa um bit que pode assumir os valores 0 ou 1. É gerada uma lista de probabilidades de valor de bits para cada byte na entrada e, em seguida, são calculadas as entropias correspondentes. O resultado final é a soma dessas entropias, fornecendo a entropia total da sequência de bits.

Primeiramente, para realizar testes de entropia de cifragem, foram geradas mensagens de entrada aleatórias com 256 bits de comprimento e baixa entropia. Posteriormente, as mensagens foram cifradas utilizando a chave secreta “2B7E151628AED2A6ABF7158809CF4F3C”, utilizada nos testes do NIST, disponibilizada pelo NIST (NIST, 2016). Além disso, foi utilizado um *nonce* fixo gerado aleatoriamente e não foram acrescentados dados associados. Foram executados um total de 10000 testes, comparando os valores de entropia entre a entrada e a saída. Os resultados são apresentados na Tabela 13 e incluem valor médio, desvio padrão σ, valor máximo e mínimo, obtidos utilizando tanto o bloco de permutação proposto quanto os algoritmos originais. Nesse caso, as mensagens foram geradas a partir da mesma função que produz uma sequência pseudoaleatória em ambientes de desenvolvimento distintos para o bloco proposto e o original, o que explica a diferença nos valores de entropia da entrada.

Tabela 13 – Testes de entropia comparando a entrada e saída para o bloco de permutação proposto e o algoritmo original do ASCON-128 para cifragem.

	Bloco Proposto (Entrada)				Bloco Proposto (Saída)			
β	Média	σ	Máximo	Mínimo	Média	σ	Máximo	Mínimo
0	0,2470	0,0966	0,4861	0,0000	0,9981	0,0023	1,0000	0,9799
1	0,2471	0,0967	0,4615	0,0000	0,9982	0,0024	1,0000	0,9730
2	0,2474	0,0967	0,4861	0,0000	0,9983	0,0024	1,0000	0,9716
3	0,2469	0,0965	0,4739	0,0000	0,9984	0,0025	1,0000	0,9745
4	0,2473	0,0969	0,4861	0,0000	0,9983	0,0023	1,0000	0,9700
5	0,2471	0,0967	0,4739	0,0000	0,9989	0,0018	1,0000	0,9811
6	0,2468	0,0963	0,4615	0,0000	0,9986	0,0020	1,0000	0,9799
	Original (Entrada)				Original (Saída)			
Original	Média	σ	Máximo	Mínimo	Média	σ	Máximo	Mínimo
	0,3498	0,1956	0,6794	0,0000	0,9983	0,0028	1,0000	0,9600

Fonte: Autora.

O próximo teste visa avaliar a entropia das mensagens cifradas ao variar a chave secreta. Foi utilizada uma mensagem de entrada fixa com 256 bits de comprimento e baixa entropia e foram geradas 10000 chaves aleatórias para realizar a cifragem destas mensagens. Os resultados das entropias da entrada e da saída são exibidos na Tabela 14.

Tabela 14 – Testes de entropia comparando a entrada e saída para o bloco de permutação proposto e o algoritmo original do ASCON-128 alterando a chave secreta para cifragem.

	Bloco Proposto (Entrada)				Bloco Proposto (Saída)			
β	Média	σ	Máximo	Mínimo	Média	σ	Máximo	Mínimo
0	0,0369	0,0000	0,0369	0,0369	0,9981	0,0027	1,0000	0,9685
1	0,0369	0,0000	0,0369	0,0369	0,9981	0,0027	1,0000	0,9700
2	0,0369	0,0000	0,0369	0,0369	0,9981	0,0027	1,0000	0,9730
3	0,0369	0,0000	0,0369	0,0369	0,9981	0,0027	1,0000	0,9652
4	0,0369	0,0000	0,0369	0,0369	0,9981	0,0026	1,0000	0,9759
5	0,0369	0,0000	0,0369	0,0369	0,9981	0,0026	1,0000	0,9652
6	0,0369	0,0000	0,0369	0,0369	0,9981	0,0027	1,0000	0,9685
	Original (Entrada)				Original (Saída)			
Original	Média	σ	Máximo	Mínimo	Média	σ	Máximo	Mínimo
	0,0369	0,0000	0,0369	0,0369	0,9981	0,0026	1,0000	0,9745

Fonte: Autora.

Também foi realizado um teste de entropia em que a mensagem cifrada é decifrada utilizando uma chave alterada em 1 bit. São geradas mensagens de entrada aleatórias com 256 bits de comprimento e baixa entropia, da mesma forma que na Tabela 13. A cifragem é realizada com uma chave gerada aleatoriamente e, em seguida, a mensagem cifrada é decifrada usando a mesma chave, mas com cada um de seus bits invertido. Foram executados 10000 testes e seus resultados são apresentados na Tabela 15.

Tabela 15 – Testes de entropia comparando a entrada e saída para o bloco de permutação proposto e o algoritmo original do ASCON-128 alterando a chave secreta para decifragem.

	Bloco Proposto (Entrada)				Bloco Proposto (Cifrado)				Bloco Proposto (Decifrado*)			
β	Média	σ	Máximo	Mínimo	Média	σ	Máximo	Mínimo	Média	σ	Máximo	Mínimo
0	0,3369	0,0397	0,4228	0,2196	0,9980	0,0030	1,0000	0,9857	0,9973	0,0039	1,0000	0,9626
1	0,3346	0,0355	0,4093	0,2196	0,9982	0,0026	1,0000	0,9867	0,9972	0,0040	1,0000	0,9600
2	0,3350	0,0381	0,4359	0,2380	0,9983	0,0026	1,0000	0,9867	0,9972	0,0039	1,0000	0,9573
3	0,3310	0,0412	0,4228	0,2380	0,9981	0,0024	1,0000	0,9905	0,9972	0,0040	1,0000	0,9485
4	0,3413	0,0450	0,4359	0,2196	0,9976	0,0039	1,0000	0,9759	0,9971	0,0042	1,0000	0,9454
5	0,3279	0,0422	0,4093	0,2196	0,9981	0,0028	1,0000	0,9857	0,9972	0,0040	1,0000	0,9626
6	0,3375	0,0391	0,4359	0,2557	0,9983	0,0029	1,0000	0,9835	0,9972	0,0040	1,0000	0,9544
	Original (Entrada)				Original (Cifrado)				Original (Decifrado*)			
Original	Média	σ	Máximo	Mínimo	Média	σ	Máximo	Mínimo	Média	σ	Máximo	Mínimo
	0,5415	0,0411	0,6439	0,3815	0,9978	0,0026	1,0000	0,9867	0,9972	0,0040	1,0000	0,9573

Fonte: Autora.

Os resultados exibidos nas Tabelas 13 a 15 mostram que tanto o bloco proposto quanto a permutação original apresentam valores similares de entropia e produzem mensagens cifradas com alta entropia. Tais entropias se mostraram estáveis e consistentes, independentemente das configurações ou variações testadas.

Na Tabela 15, a entropia da mensagem decifrada com uma chave errada é significativamente mais alta do que a da mensagem de entrada, indicando alta aleatoriedade e falta de similaridades entre elas. Adicionalmente, as alterações no β não afetam substancialmente a entropia das mensagens cifradas, indicando que diferentes valores de β são eficazes para gerar boa entropia.

5.3 SUITE DE TESTES DO NIST

A suíte de testes do NIST é uma ferramenta projetada para avaliar a aleatoriedade de geradores de números aleatórios e pseudoaleatórios. Esses testes visam identificar possíveis padrões ou fraquezas nas sequências binárias geradas. Além disso, eles também servem para o desenvolvimento e verificação de novos PRNGs, como também assegurar a precisão das implementações existentes (BASSHAM et al., 2010). A suíte é composta por um conjunto de 15 testes estatísticos, os quais são:

- **Teste de Frequência (Monobit):** verifica se existe uma distribuição equilibrada entre "zeros" e "uns" em uma sequência de bits, como seria esperado para uma sequência verdadeiramente aleatória.
- **Teste de Frequência dentro de um Bloco:** avalia a distribuição de "uns" em blocos de bits, verificando se sua proporção é cerca de metade do comprimento do bloco.
- **Teste de Sequências:** verifica a quantidade e a distribuição de sequências contínuas de "zeros" e "uns" em uma sequência de bits e determinam se está dentro do esperado para uma sequência aleatória.
- **Teste para a Maior Sequência de Uns em um Bloco:** avalia a sequência mais longa de "uns" em blocos de bits e verifica se está de acordo com o esperado em uma sequência aleatória.
- **Teste de Rank de Matriz Binária:** verifica se uma *substring* da sequência de bits é linearmente dependente da sequência total.
- **Teste da Transformada Discreta de Fourier (Espectral):** verifica se há padrões periódicos na sequência de bits, indicando que existe uma falta de aleatoriedade.
- **Teste de Correspondência de Modelo Não-Sobreposto:** analisa a quantidade de vezes que sequências de padrões aperiódicos aparecem na sequência de bits.

- **Teste de Correspondência de Modelo Sobreposto:** verifica a quantidade de vezes que sequências específicas aparecem na sequência de bits.
- **Teste Estatístico Universal de Maurer:** verifica a possibilidade de comprimir uma sequência de bits sem perder informações. Sequências que podem ser facilmente comprimidas são consideradas não aleatórias.
- **Teste de Complexidade Linear:** utiliza um LFSR e analisa o comprimento da sequência de bits, determinando a complexidade e aleatoriedade da sequência de bits. LFSR's mais curtos indicam uma sequência não aleatória.
- **Teste Serial:** avalia a frequência de diferentes padrões de bits na sequência de bits. Uma frequência alta sugere uma semelhança com o que é observado em uma sequência aleatória.
- **Teste de Entropia Aproximada:** compara a frequência de padrões de um determinado tamanho com a de padrões de um tamanho ligeiramente maior na sequência e verifica se existe semelhança com uma sequência aleatória.
- **Teste de Soma Cumulativa:** verifica se a soma acumulada de dígitos ajustados (-1 e 1) na sequência de bits desvia muito do zero. Em sequências aleatórias, essa soma oscila perto de zero.
- **Teste de Excursões Aleatórias:** avalia se a quantidade de vezes que a caminhada aleatória atinge pontos específicos em seus ciclos difere do que seria esperado em uma sequência aleatória.
- **Teste de Variante de Excursões Aleatórias:** avalia o número de visitas a estados específicos em uma caminhada aleatória cumulativa e identifica desvios do esperado em uma sequência aleatória.

A fim de obter resultados significativos no conjunto de testes é recomendado que a sequência de bits a ser testada tenha pelo menos 10^6 bits. Ao finalizar os testes, cada um gera um "Valor-P", o qual é um valor estatístico que indica a probabilidade de que a estatística de teste escolhida assuma valores iguais ou piores do que o valor observado da estatística de teste ao considerar a hipótese nula.

Para executar a suíte de testes do NIST, primeiramente foram geradas sequências de bits aleatórias utilizando o ASCON-PRF. No total, foram produzidas 10 sequências de 10^6 bits, totalizando 10^7 bits. Em seguida, aplicaram-se os 15 testes disponíveis na suíte a essas sequências. O bloco de permutação original do ASCON-PRF foi substituído pelo bloco proposto, garantindo que todos os betas fossem testados com a mesma quantidade de dados e comparados com os resultados da permutação original.

Os resultados dos testes estatísticos do NIST aplicados às 10 sequências fornecidas são apresentados na Tabela 16. Esta tabela inclui a distribuição uniforme do Valor-P, que deve ser igual ou superior a $0,0001$, e as proporções de sequências que passaram nos testes para cada β, além dos resultados para o algoritmo original do ASCON-PRF.

A análise dos resultados da Tabela 16 mostra que todas as sequências foram consideradas aleatórias em todos os testes, com exceção do $\beta - 2$, que não obteve êxito apenas no teste de Variante de Excursões Aleatórias. Esse foi o único teste que não indicou aleatoriedade, enquanto nos demais, o $\beta - 2$ apresentou resultados satisfatórios. Os resultados indicam que todos os β's testados apresentam características mínimas para serem empregados no bloco de permutação. Além disso, é observado que o desempenho do bloco de permutação proposto é comparável ao do ASCON original.

Tabela 16 – Resultado do conjunto de testes estatísticos do NIST

RESULTADO DA DISTRIBUIÇÃO UNIFORME DOS VALORES-P E A PROPORÇÃO DAS SEQUÊNCIAS DE BITS FORNECIDAS								
TESTE ESTATÍSTICO	β - 0		β - 1		β - 2		β - 3	
	VALOR-P	*PROPORÇÃO*	*VALOR-P*	*PROPORÇÃO*	*VALOR-P*	*PROPORÇÃO*	*RESULTADO*	*VALOR-P*
Frequência	0,9114	10/10	0,1223	10/10	0,1223	9/10	0,7399	9/10
Frequência de Bloco	0,5341	10/10	0,7399	10/10	0,1223	10/10	0,5341	10/10
Soma Cumulativa	0,5341	10/10	0,1223	10/10	0,3504	10/10	0,7399	9/10
Sequências	0,7399	10/10	0,5341	9/10	0,3504	9/10	0,2133	10/10
Maior Sequência	0,3504	10/10	0,3504	10/10	0,1223	10/10	0,3504	10/10
Rank	0,3504	9/10	0,1223	10/10	0,3504	10/10	0,0004	10/10
FFT	0,5341	10/10	0,7399	10/10	0,7399	10/10	0,5341	10/10
Modelo Não-Sobreposto	0,7399	10/10	0,9114	10/10	0,7399	10/10	0,1223	10/10
Modelo Sobreposto	0,7399	10/10	0,9114	10/10	0,9114	10/10	0,2133	10/10
Universal	0,9114	9/10	0,3504	9/10	0,9914	10/10	0,1223	10/10
Entropia Aproximada	0,3504	10/10	0,3504	9/10	0,7399	10/10	0,9914	10/10
Excursões Aleatórias	—-	6/7	—-	6/6	—-	7/8	—-	5/6
Variante de Excursões Aleatórias	—-	7/7	—-	6/6	—-	6/8	—-	6/6
Serial	0,5341	10/10	0,7399	10/10	0,5341	10/10	0,5341	10/10
Complexidade Linear	0,7399	10/10	0,5341	10/10	0,9114	10/10	0,0351	10/10
TESTE ESTATÍSTICO	β - 4		β - 5		β - 6		ASCON-PRF	
	VALOR-P	*PROPORÇÃO*	*VALOR-P*	*PROPORÇÃO*	*VALOR-P*	*PROPORÇÃO*	*RESULTADO*	*VALOR-P*
Frequência	0,3504	10/10	0,7399	10/10	0,0088	10/10	0,5341	10/10
Frequência de Bloco	0,7399	10/10	0,5341	10/10	0,3504	10/10	0,5341	10/10
Soma Cumulativa	0,7399	10/10	0,3504	10/10	0,7399	10/10	0,3504	10/10
Sequências	0,2133	10/10	0,5341	10/10	0,1223	10/10	0,9114	10/10
Maior Sequência	0,5341	10/10	0,3504	10/10	0,5341	10/10	0,5341	10/10
Rank	0,3504	10/10	0,7399	10/10	0,7399	10/10	0,5341	10/10
FFT	0,3504	10/10	0,7399	10/10	0,0179	10/10	0,7399	10/10
Modelo Não-Sobreposto	0,7399	10/10	0,1223	10/10	0,3504	10/10	0,5341	10/10
Modelo Sobreposto	0,5341	10/10	0,5341	9/10	0,7399	10/10	0,7399	10/10
Universal	0,5341	10/10	0,7399	9/10	0,1223	10/10	0,9114	9/10
Entropia Aproximada	0,1223	10/10	0,0668	10/10	0,5341	9/10	0,7399	10/10
Excursões Aleatórias	—-	8/8	—-	5/5	—-	6/7	—-	6/7
Variante de Excursões Aleatórias	—-	7/8	—-	5/5	—-	6/7	—-	7/7
Serial	0,2133	10/10	0,2133	9/10	0,7399	9/10	0,7399	10/10
Complexidade Linear	0,3504	10/10	0,9114	10/10	0,5341	10/10	0,7399	10/10

Fonte: Autora.

5.4 EFEITO AVALANCHE

Como mencionado em 4.1, o efeito avalanche é uma propriedade desejada em algoritmos criptográficos que serve para medir a sensibilidade das mudanças na entrada em relação às mudanças na saída. Para calcular o efeito avalanche, foi utilizado (4.1), nesta função, para cada bit nas duas entradas, é contado quantos são idênticos, ou seja, tem o mesmo valor na mesma posição. A contagem de bits idênticos é então dividida pelo total de bits nas duas entradas, resultando na proporção de bits diferentes entre elas.

Os testes de efeito avalanche foram conduzidos de maneira sistemática. Foi utilizada uma mensagem de entrada com 80 bits de comprimento, gerada aleatoriamente, e o *hash* foi calculado para cada algoritmo como etapa inicial. Em seguida, para avaliar a sensibilidade do *hash* a pequenas alterações no texto de entrada, um bit por vez foi invertido, resultando em um total de 80 *hashes* distintos para cada algoritmo. Durante esse processo, a função de efeito avalanche foi aplicada a cada *hash* resultante das alterações de um único bit na mensagem original. Este procedimento foi repetido 100 vezes para cada algoritmo, totalizando em 8000 *hashes* analisados para cada iteração de cada um desses algoritmos.

Na Tabela 17, estão apresentados os resultados detalhados dos testes de efeito avalanche realizados para os algoritmos SHA3, Quark, SPONGENT e ASCON-*hash*.

Tabela 17 – Testes de efeito avalanche comparando o bloco de permutação proposto com o algoritmo original.

Algoritmo	Com o Bloco Proposto				Algoritmo Original			
	Média	σ	Máximo	Mínimo	Média	σ	Máximo	Mínimo
ASCON-*hash*	0,4998	0,0311	0,6055	0,3867	0,5003	0,0309	0,6250	0,3867
S-Quark	0,5004	0,0312	0,6055	0,3906	0,4992	0,0314	0,6172	0,3672
SHA3-224	0,4994	0,0359	0,6250	0,3795	0,4996	0,0335	0,6250	0,3750
SHA3-256	0,4999	0,0343	0,6250	0,3594	0,5002	0,0311	0,6370	0,3906
SHA3-384	0,5003	0,0284	0,5938	0,3802	0,4994	0,0252	0,6020	0,3906
SHA3-512	0,4995	0,0241	0,5859	0,4121	0,4995	0,0222	0,5780	0,4199
SPONGENT-1	0,5003	0,0439	0,6641	0,3438	0,5003	0,0443	0,6800	0,3125
SPONGENT-2	0,5000	0,0392	0,6438	0,3625	0,4999	0,0393	0,6563	0,3687
SPONGENT-3	0,5004	0,0334	0,6205	0,3750	0,5001	0,0332	0,6161	0,3839
SPONGENT-4	0,5005	0,0320	0,6172	0,3828	0,5003	0,0313	0,6055	0,3867
SPONGENT-5	0,4999	0,0309	0,6094	0,3789	0,5005	0,0311	0,6094	0,3867

Fonte: Autora.

Para o algoritmo ASCON-128, cuja mensagem de entrada consistia em 256 bits de comprimento, o processo foi adaptado. Nele não foi escolhido apenas um único β, todos foram testados e comparados com a permutação original. A mensagem de entrada, assim como a chave e o *nonce*, foram gerados aleatoriamente, enquanto que para os dados associados foi utilizada a palavra em hexadecimal "6C616973". Esses valores foram mantidos constantes em todos os testes. Neste teste, primeiramente foram calculadas as cifras. Um bit da mensagem de entrada

foi então alterado por vez em seu comprimento total. Após essa modificação, a cifragem foi realizada novamente, e o efeito avalanche entre as duas mensagens cifradas foi calculado. Este processo resultou em um total de 25600 resultados, os quais estão apresentados na Tabela 18.

Tabela 18 – Testes de efeito avalanche comparando o bloco de permutação proposto com o algoritmo original do ASCON-128 para cifragem.

	Com o Bloco Proposto			
β	Média	σ	Máximo	Mínimo
0	0,2944	0,0953	0,5078	0,1198
1	0,2945	0,0953	0,5000	0,1172
2	0,2942	0,0949	0,5000	0,1120
3	0,2942	0,0951	0,5078	0,1172
4	0,2943	0,0953	0,5078	0,1146
5	0,2942	0,0951	0,5000	0,1172
6	0,2943	0,0954	0,5286	0,1146
	Algoritmo Original			
Original	Média	σ	Máximo	Mínimo
	0,2942	0,0951	0,5026	0,1172

Fonte: Autora.

No caso da função de decifragem do ASCON-128, uma mensagem de entrada de 256 bits de comprimento, gerada aleatoriamente, é cifrada. Em seguida, um bit é invertido na mensagem cifrada e ela é decifrada. Esse processo é repetido para todos os 256 bits do texto cifrado. Após cada decifragem, o efeito avalanche é calculado comparando a mensagem decifrada original com a mensagem decifrada após a alteração de um único bit. Esse procedimento gerou 25600 resultados que estão apresentados na Tabela 19 e permite avaliar como a alteração de um bit no texto cifrado afeta a mensagem decifrada.

Tabela 19 – Testes de efeito avalanche comparando o bloco de permutação proposto com o algoritmo original do ASCON-128 para decifragem.

	Com o Bloco Proposto			
β	Média	σ	Máximo	Mínimo
0	0,1914	0,1410	0,4883	0,0039
1	0,1913	0,1410	0,4883	0,0039
2	0,1917	0,1412	0,4766	0,0039
3	0,1913	0,1411	0,4805	0,0039
4	0,1912	0,1410	0,4766	0,0039
5	0,1915	0,1411	0,4805	0,0039
6	0,1914	0,1412	0,4688	0,0039
	Algoritmo Original			
Original	Média	σ	Máximo	Mínimo
	0,1913	0,1409	0,4727	0,0039

Fonte: Autora.

Outro teste realizado é com a chave do ASCON-128. Uma mensagem de entrada de 256 bits de comprimento, gerada aleatoriamente, é cifrada utilizando uma chave de 128 bits de comprimento, que também é gerada aleatoriamente. Neste teste, cada um dos bits da chave é invertido e, em seguida, a mensagem é cifrada novamente. Esse processo é repetido para cada bit da chave. O efeito avalanche entre a mensagem cifrada original e a mensagem cifrada com a chave alterada é calculado, gerando 12800 resultados, exibidos na Tabela 20.

Tabela 20 – Testes de efeito avalanche comparando o bloco de permutação proposto com o algoritmo original do ASCON-128 alterando a chave secreta para cifragem.

	Com o Bloco Proposto			
β	Média	σ	Máximo	Mínimo
0	0,4998	0,0255	0,6042	0,3958
1	0,5000	0,0255	0,6042	0,4089
2	0,4997	0,0255	0,6042	0,3828
3	0,4999	0,0254	0,6068	0,3880
4	0,4999	0,0255	0,5964	0,4089
5	0,5003	0,0253	0,5911	0,3984
6	0,4996	0,0255	0,5911	0,4036
	Algoritmo Original			
Original	Média	σ	Máximo	Mínimo
	0,5000	0,0313	0,6133	0,3555

Fonte: Autora.

No último teste, uma mensagem de 256 bits de comprimento é gerada aleatoriamente, assim como a chave, que também é gerada de forma aleatória. É realizada a cifragem da mensagem de entrada e em seguida, um bit da chave é invertido e o texto cifrado é decifrado utilizando essa chave modificada. Esse processo é repetido alterando cada bit da chave, resultando em um total de 128000 testes de efeito avalanche entre a mensagem de entrada e a mensagem decifrada, exibidos na Tabela 21.

Tabela 21 – Testes de efeito avalanche comparando o bloco de permutação proposto com o algoritmo original do ASCON-128 alterando a chave secreta para decifragem.

	Com o Bloco Proposto			
β	Média	σ	Máximo	Mínimo
0	0,4999	0,0315	0,6172	0,3672
1	0,5003	0,0315	0,6211	0,3828
2	0,5000	0,0312	0,6328	0,3984
3	0,5000	0,0315	0,6172	0,3750
4	0,5001	0,0311	0,6250	0,3672
5	0,5001	0,0312	0,6211	0,3789
6	0,5003	0,0313	0,6445	0,3750
	Algoritmo Original			
Original	Média	σ	Máximo	Mínimo
	0,5003	0,0258	0,6120	0,3984

Fonte: Autora.

Com base nos testes realizados, observa-se que os algoritmos implementados com o bloco de permutação exibem um valor satisfatório para o efeito avalanche, com uma média próxima de $0{,}50$. Tanto os valores máximos quanto os mínimos estão próximos das faixas desejadas de $0{,}60$ e $0{,}45$, respectivamente, o que indica uma boa aleatoriedade e segurança nos resultados. Essa tendência é observada em todos os algoritmos testados, tanto nos originais quanto nos implementados com o bloco de permutação.

Os resultados apresentados na Tabela 17 mostram que, mesmo com apenas duas aplicações de rodadas de permutação, os valores de efeito avalanche permanecem semelhantes entre as versões original e com o bloco de permutação. Isso também é evidente no ASCON-128, nas Tabelas 20 e 21 dos testes de cifragem e decifragem, respectivamente, com chaves geradas aleatoriamente. Além disso, pode-se notar que a mensagem decifrada difere em cerca de metade dos bits da mensagem de entrada, indicando que os algoritmos são sensíveis a mudanças nos dados de entrada.

No caso das Tabelas 18 e 19 de cifragem e decifragem, respectivamente, a média do efeito avalanche está fora do intervalo $0,45 - 0,60$. Vale ressaltar que os resultados dessa tabela são interessantes devido à estrutura específica do ASCON-128, onde a difusão varia dependendo da posição do bit invertido na mensagem de entrada. Por exemplo, na cifragem, a difusão é maior quando o bit é invertido no início da mensagem, enquanto na decifragem, ocorre o inverso. Essa observação destaca a influência do posicionamento do bit, ao longo do comprimento da mensagem, na aleatoriedade e segurança do algoritmo.

5.5 CONSIDERAÇÕES

Neste capítulo, os testes realizados avaliaram a aleatoriedade e segurança do novo bloco de permutação e dos algoritmos originais para fins de comparação. Observa-se que o bloco proposto apresenta valores comparáveis aos dos algoritmos originais mesmo com apenas duas rodadas de aplicação do bloco de permutação, sugerindo difusão e confusão adequadas para um algoritmo criptográfico.

6 CONCLUSÕES

No presente trabalho, destaca-se a relevância de estruturas algébricas finitas utilizadas na definição de primitivas criptográficas na construção de sistemas de segurança e comunicação segura. Nesse contexto, foram sumarizados os seguintes pontos:

1. Foi analisada a importância de primitivas criptográficas na construção de sistemas criptográficos seguros, tais como algoritmos de cifragem e decifragem, e funções *hash*.
2. Foram investigadas estruturas algébricas finitas, uma vez que elas podem ser utilizadas na definição de primitivas criptográficas.
3. Foi explorado o uso de corpos finitos na aritmética empregada em criptografia, bem como seu papel na aplicação de técnicas como a transformada do cosseno de tipo 1 e polinômios de permutação.
4. A construção esponja foi destacada como uma primitiva versátil devido à sua flexibilidade na manipulação do tamanho do estado interno. Essa versatilidade é evidenciada pela sua aplicação em algoritmos criptográficos, como o ASCON e o SHA3, ambos vencedores de competições do NIST.
5. Foi apresentada a criação de um novo bloco de permutação, que integra polinômios de permutação e transformada do cosseno de tipo 1 na estrutura de uma construção esponja, para gerar difusão e confusão, visando fornecer uma segurança adequada para algoritmos criptográficos.
6. Foi realizada a implementação bem-sucedida do novo bloco de permutação em algoritmos criptográficos, como SHA3, S-Quark, SPONGENT e ASCON, demonstrando um desempenho equivalente aos algoritmos originais em suas iterações completas.
7. Foram conduzidos testes de entropia, teste de efeito avalanche e um conjunto de testes estatísticos do NIST para avaliar a aleatoriedade do bloco proposto em comparação com os algoritmos originais.
8. Também foram realizados testes de resistência, para avaliar a robustez do bloco proposto contra ataques de força bruta, também comparando-os com os originais.
9. Mesmo sendo composto por apenas duas rodadas, o novo bloco de permutação demonstra robustez e aleatoriedade, de acordo com os testes realizados.

6.1 TRABALHOS FUTUROS

Como trabalhos futuros são sugeridos:

- Avaliar o desempenho do novo bloco de permutação implementado em uma linguagem de baixo nível como C ou mesmo Assembler, e utilizar técnicas de otimização feitas para funções *hash*, como a utilização de tabelas de substituição para calcular a multiplicação de matriz da transformada do cosseno.
- Implementar o bloco proposto em hardware dedicado, utilizando linguagem de descrição de hardware e testar o seu desempenho utilizando hardware reconfigurável.
- Estudar modificações na arquitetura do bloco de permutação proposto que visem melhorar o desempenho, em termos de velocidade, mantendo os resultados de segurança alcançados.
- Avaliar o uso de outras transformadas definidas sobre estruturas algébricas finitas, como por exemplo a FFCT-IV ou outro polinômio de permutação que também seja uma involução, avaliando sua aplicabilidade para o projeto de premissas criptográficas.
- Realizar uma análise mais abrangente para avaliar a resistência do novo bloco contra diversos tipos de criptoanálises, como ataques lineares e diferenciais.

6.2 ARTIGOS RELACIONADOS À DISSERTAÇÃO

A seguir, é indicado o artigo submitido e aceito para apresentação no XLIII Congresso Nacional de Matemática Aplicada e Computacional a ser realizado dos dias 16 a 20 de Setembro de 2024, Porto de Galinhas, PE.

- DE ARAÚJO, L. M. R.; LIMA, J. B. DE OLIVEIRA NETO, J. R. “Um Bloco de Permutação para Construções Esponja Baseado na Transformada do Cosseno sobre Corpos Finitos de Característica Dois”. **Em: CNMAC 2024 - XLIII Congresso Nacional de Matemática Aplicada e Computacional**, Setembro de 2024, Porto de Galinhas, PE.

REFERÊNCIAS

ABOUSHOSHA, B. et al. Slim: A lightweight block cipher for internet of health things. **IEEE Access**, v. 8, p. 203747–203757, 2020. Citado na página 12.

ADOMNICAI, A.; FOURNIER, J. J. A.; MASSON, L. Masking the lightweight authenticated ciphers acorn and ascon in software. **IACR Cryptol. ePrint Arch.**, v. 2018, p. 708, 2018. Citado na página 48.

AHMED, N.; NATARAJAN, T.; RAO, K. Discrete cosine transform. **IEEE Transactions on Computers**, C-23, n. 1, p. 90–93, 1974. Citado na página 25.

AKBARY, A.; GHIOCA, D.; WANG, Q. On constructing permutations of finite fields. **Finite Fields and Their Applications**, v. 17, n. 1, p. 51–67, 2011. ISSN 1071-5797. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1071579710000869>. Citado 2 vezes nas páginas 23 e 24.

AL-MHADAWI, M. M.; ALBAHRANI, E. A.; LAFTA, S. H. Efficient and secure chaotic prng for color image encryption. **Microprocessors and Microsystems**, v. 101, p. 104911, 2023. ISSN 0141-9331. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0141933123001552>. Citado 2 vezes nas páginas 12 e 32.

AL-SHATARI, M. O. A. et al. FPGA-based lightweight hardware architecture of the photon hash function for iot edge devices. **IEEE Access**, v. 8, p. 207610–207618, 2020. Citado na página 28.

ALAWIDA, M. et al. A novel hash function based on a chaotic sponge and DNA sequence. **IEEE Access**, v. 9, p. 17882–17897, 2021. Citado 2 vezes nas páginas 12 e 29.

ALAWIDA, M. et al. A new hash function based on chaotic maps and deterministic finite state automata. **IEEE Access**, v. 8, p. 113163–113174, 2020. Citado na página 12.

ALFRHAN, A.; MOULAHI, T.; ALABDULATIF, A. Comparative study on hash functions for lightweight blockchain in internet of things (IoT). **Blockchain: Research and Applications**, v. 2, n. 4, p. 100036, 2021. ISSN 2096-7209. Disponível em: <https://www.sciencedirect.com/science/article/pii/S2096720921000312>. Citado na página 29.

ALTAWY, R. et al. Towards a cryptographic minimal design: The sliscp family of permutations. **IEEE Transactions on Computers**, v. 67, n. 9, p. 1341–1358, 2018. Citado na página 12.

ASSCHE, G. V. **XKCP: eXtended Keccak Code Package**. [S.l.]: GitHub, 2024. <https://github.com/XKCP/XKCP>. Citado na página 60.

ASTUTI, N.; ARFIANI, I.; ARIBOWO, E. Analysis of the security level of modified CBC algorithm cryptography using avalanche effect. In: IOP PUBLISHING. **IOP Conference Series: Materials Science and Engineering**. [S.l.], 2019. v. 674, n. 1, p. 012056. Citado na página 53.

AUMASSON, J.-P. et al. Quark: A lightweight hash. **Journal of cryptology**, Springer, v. 26, p. 313–339, 2013. Citado 7 vezes nas páginas 28, 30, 37, 38, 39, 40 e 52.

AUMASSON, J.-P.; KNELLWOLF, S.; MEIER, W. Heavy Quark for secure AEAD. **DIAC-Directions in Authenticated Ciphers**, 2012. Citado na página 37.

BASSHAM, L. et al. **A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications**. Special Publication (NIST SP), National Institute of Standards and Technology, Gaithersburg, MD, 2010. Disponível em: <https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=906762>. Citado na página 67.

BERTONI, G. et al. Sponge functions. In: **ECRYPT hash workshop**. [S.l.: s.n.], 2007. v. 2007, n. 9. Citado 4 vezes nas páginas 12, 28, 29 e 30.

BERTONI, G. et al. On the indifferentiability of the sponge construction. In: SPRINGER. **Annual International Conference on the Theory and Applications of Cryptographic Techniques**. [S.l.], 2008. p. 181–197. Citado na página 30.

BERTONI, G. et al. Sponge-based pseudo-random number generators. In: SPRINGER. **Cryptographic Hardware and Embedded Systems, CHES 2010: 12th International Workshop, Santa Barbara, USA, August 17-20, 2010. Proceedings 12**. [S.l.], 2010. p. 33–47. Citado na página 30.

BERTONI, G. et al. Duplexing the sponge: Single-pass authenticated encryption and other applications. In: MIRI, A.; VAUDENAY, S. (Ed.). **Selected Areas in Cryptography**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. p. 320–337. Citado 2 vezes nas páginas 30 e 31.

BLAHUT, R. E. **Fast algorithms for signal processing**. [S.l.]: Cambridge University Press, 2010. Citado na página 25.

BOGDANOV, A. et al. spongent: A lightweight hash function. In: PRENEEL, B.; TAKAGI, T. (Ed.). **Cryptographic Hardware and Embedded Systems – CHES 2011**. [S.l.]: Springer Berlin Heidelberg, 2011. Citado 4 vezes nas páginas 28, 32, 43 e 52.

BOGDANOV, A. et al. Spongent: The design space of lightweight cryptographic hashing. **IEEE Transactions on Computers**, v. 62, n. 10, p. 2041–2053, 2013. Citado 3 vezes nas páginas 43, 44 e 45.

BOGDANOV, A. et al. Present: An ultra-lightweight block cipher. In: SPRINGER. **Cryptographic Hardware and Embedded Systems-CHES 2007: 9th International Workshop, Vienna, Austria, September 10-13, 2007. Proceedings 9**. [S.l.], 2007. p. 450–466. Citado 2 vezes nas páginas 41 e 43.

BOUTIN, C. **NIST Selects 'Lightweight Cryptography' Algorithms to Protect Small Devices**. 2023. <https://www.nist.gov/news-events/news/2023/02/nist-selects-lightweight-cryptography-algorithms-protect-small-devices> [Accessed: (18 de Março de 2024)]. Citado 2 vezes nas páginas 12 e 45.

BRUCE, S. **Applied Cryptography: Protocols, Algorthms, and Source Code in C.-2nd**. [S.l.]: John Wiley & Sons, 1996. Citado na página 13.

CACHIN, C. **Entropy measures and unconditional security in cryptography**. Tese (Doutorado) — ETH Zurich, 1997. Citado na página 64.

CAMPELLO DE SOUZA, M. M. et al. The discrete cosine transform over prime finite fields. In: SPRINGER. **Telecommunications and Networking-ICT 2004: 11th International Conference on Telecommunications, Fortaleza, Brazil, August 1-6, 2004. Proceedings 11**. [S.l.], 2004. p. 482–487. Citado 3 vezes nas páginas 13, 25 e 26.

CANNIERE, C. D. Trivium: A stream cipher construction inspired by block cipher design principles. In: SPRINGER. **International Conference on Information Security**. [S.l.], 2006. p. 171–186. Citado na página 32.

CHAN, Y. Y. et al. On the resistance of new lightweight block ciphers against differential cryptanalysis. **Heliyon**, v. 9, n. 4, p. e15257, 2023. ISSN 2405-8440. Disponível em: <https://www.sciencedirect.com/science/article/pii/S2405844023024647>. Citado na página 61.

CHARPIN, P.; MESNAGER, S.; SARKAR, S. On involutions of finite fields. In: IEEE. **2015 IEEE International Symposium on Information Theory (ISIT)**. [S.l.], 2015. p. 186–190. Citado 2 vezes nas páginas 24 e 25.

CHENG, J.; GUO, S.; HE, J. An extended type-1 generalized feistel networks: Lightweight block cipher for iot. **IEEE Internet of Things Journal**, v. 9, n. 13, p. 11408–11421, 2022. Citado na página 12.

CHOI, H.; SEO, S. C. Fast implementation of SHA-3 in GPU environment. **IEEE Access**, v. 9, p. 144574–144586, 2021. Citado 2 vezes nas páginas 36 e 37.

CUI, J. et al. An efficient message-authentication scheme based on edge computing for vehicular Ad Hoc networks. **IEEE Transactions on Intelligent Transportation Systems**, v. 20, n. 5, p. 1621–1632, 2019. Citado na página 12.

DING, L. et al. A practical key recovery attack on the lightweight WG-5 stream cipher. **Heliyon**, v. 10, n. 2, p. e24197, 2024. ISSN 2405-8440. Disponível em: <https://www.sciencedirect.com/science/article/pii/S2405844024002287>. Citado na página 61.

DING, Y. et al. Adaptive chosen-plaintext collision attack on masked AES in edge computing. **IEEE Access**, v. 7, p. 63217–63229, 2019. Citado na página 61.

DOBRAUNIG, C. et al. Ascon PRF, MAC, and short-input MAC. **Cryptology ePrint Archive**, 2021. Citado 4 vezes nas páginas 12, 50, 51 e 52.

DOBRAUNIG, C. et al. Ascon v1.2: Lightweight authenticated encryption and hashing. **Journal of Cryptology**, Springer Science and Business Media LLC, v. 34, n. 3, jun 2021. Citado 8 vezes nas páginas 12, 28, 45, 46, 47, 49, 50 e 52.

DOULIGERIS, C.; SERPANOS, D. N. Appendix a: Cryptography primer: Introduction to cryptographic principles and algorithms. In: ____. **Network Security: Current Status and Future Directions**. [S.l.: s.n.], 2007. p. 459–479. Citado na página 12.

DWORKIN, M. **SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions**. [S.l.]: Federal Inf. Process. Stds. (NIST FIPS), National Institute of Standards and Technology, Gaithersburg, MD, 2015. Citado 6 vezes nas páginas 12, 34, 35, 36, 37 e 52.

DWORKIN, M. **Hash Functions: SHA-3 Project**. 2017. <https://csrc.nist.gov/projects/hash-functions/sha-3-project> [Accessed: (07 de Abril de 2024)]. Citado na página 12.

DWORKIN, M. et al. **Advanced Encryption Standard (AES)**. [S.l.]: Federal Inf. Process. Stds. (NIST FIPS), National Institute of Standards and Technology, Gaithersburg, MD, 2001. Citado 3 vezes nas páginas 13, 32 e 42.

EASTTOM, C. **Modern Cryptography: Applied Mathematics for Encryption and Information Security**. 1st. ed. [S.l.]: McGraw-Hill Education Group, 2015. ISBN 1259588084. Citado 2 vezes nas páginas 12 e 28.

EICHLSEDER, M.; VISSOULTCHEV, V.; FAZ, A. **Python implementation of Ascon**. [S.l.]: GitHub, 2023. <https://github.com/meichlseder/pyascon>. Citado na página 60.

ESGIN, M. F.; KARA, O. Practical cryptanalysis of full sprout with TMD tradeoff attacks. In: SPRINGER. **Selected Areas in Cryptography–SAC 2015: 22nd International Conference, Sackville, NB, Canada, August 12–14, 2015, Revised Selected Papers 22**. [S.l.], 2016. p. 67–85. Citado na página 12.

FEISTEL, H. Cryptography and computer privacy. **Scientific American**, Scientific American, a division of Nature America, Inc., v. 228, n. 5, p. 15–23, 1973. ISSN 00368733, 19467087. Disponível em: <http://www.jstor.org/stable/24923044>. Citado na página 46.

FLAJOLET, P.; SEDGEWICK, R. **Analytic combinatorics**. [S.l.]: cambridge University press, 2009. Citado na página 24.

GAO, W. et al. Construction of nonlinear component of block cipher by action of modular group PSL(2, z) on projective line PL(gf(28)). **IEEE Access**, v. 8, p. 136736–136749, 2020. Citado na página 13.

GAO, Y.; ZHOU, Y. Side-channel attacks with multi-thread mixed leakage. **IEEE Transactions on Information Forensics and Security**, v. 16, p. 770–785, 2021. Citado na página 61.

GROSS, H. et al. Ascon hardware implementations and side-channel evaluation. **Microprocessors and Microsystems**, v. 52, p. 470–479, 2017. ISSN 0141-9331. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0141933116302721>. Citado 2 vezes nas páginas 29 e 48.

GUIDO, B. et al. **Cryptographic sponge functions**. [S.l.]: Citeseer, 2011. Citado 2 vezes nas páginas 30 e 31.

GUO, H.; YU, X. A survey on blockchain technology and its security. **Blockchain: Research and Applications**, p. 100067, 2022. ISSN 2096-7209. Disponível em: <https://www.sciencedirect.com/science/article/pii/S2096720922000070>. Citado na página 33.

GUO, J.; PEYRIN, T.; POSCHMANN, A. The photon family of lightweight hash functions. In: ROGAWAY, P. (Ed.). **Advances in Cryptology – CRYPTO 2011**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. Citado 8 vezes nas páginas 28, 30, 32, 40, 41, 42, 43 e 52.

GUO, Y.; LI, L.; LIU, B. Shadow: A lightweight block cipher for IoT nodes. **IEEE Internet of Things Journal**, v. 8, n. 16, p. 13014–13023, 2021. Citado na página 12.

GUPTA, M.; GARG, A. K. Analysis of image compression algorithm using DCT. **International Journal of Engineering Research and Applications (IJERA)**, Citeseer, v. 2, n. 1, p. 515–521, 2012. Citado na página 25.

HARDY, G.; WRIGHT, E. **An Introduction to the Theory of Numbers**. Clarendon Press, 1979. (Oxford science publications). ISBN 9780198531715. Disponível em: <https://books.google.com.br/books?id=FlUj0Rk_rF4C>. Citado na página 22.

HE, Y. et al. Improved cube attacks on some authenticated encryption ciphers and stream ciphers in the internet of things. **IEEE Access**, v. 8, p. 20920–20930, 2020. Citado na página 61.

HERSTEIN, I. **Tópicos de Algebra, Ed**. [S.l.]: Polıgono, 1970. Citado na página 16.

JIMALE, M. A. et al. Parallel sponge-based authenticated encryption with side-channel protection and adversary-invisible nonces. **IEEE Access**, v. 10, p. 50819–50838, 2022. Citado na página 12.

KHAN, S.; LEE, W.-K.; HWANG, S. O. Scalable and efficient hardware architectures for authenticated encryption in IoT applications. **IEEE Internet of Things Journal**, v. 8, n. 14, p. 11260–11275, 2021. Citado 2 vezes nas páginas 46 e 48.

KHAN, S.; LEE, W.-K.; HWANG, S. O. Aechain: A lightweight blockchain for IoT applications. **IEEE Consumer Electronics Magazine**, v. 11, n. 2, p. 64–76, 2022. Citado na página 49.

LIMA, J.; BARONE, M.; CAMPELLO DE SOUZA, R. Cosine transforms over fields of characteristic 2. **Finite Fields and Their Applications**, v. 37, p. 265–284, 2016. ISSN 1071-5797. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1071579715001069>. Citado 4 vezes nas páginas 26, 27, 53 e 54.

LIMA, J.; CAMPELLO DE SOUZA, R.; PANARIO, D. The eigenstructure of finite field trigonometric transforms. **Linear Algebra and its Applications**, v. 435, n. 8, p. 1956–1971, 2011. ISSN 0024-3795. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0024379511002552>. Citado na página 27.

LIMA, J. B.; SILVA, E. S. da; CAMPELLO DE SOUZA, R. M. Cosine transforms over fields of characteristic 2: fast computation and application to image encryption. **Signal Processing: Image Communication**, Elsevier, v. 54, p. 130–139, 2017. Citado na página 27.

MAETOUQ, A.; DAUD, S. M. HMNT: Hash function based on new mersenne number transform. **IEEE Access**, v. 8, p. 80395–80407, 2020. Citado na página 12.

MASUDA, A. M.; PANARIO, D. **Tópicos de corpos finitos com aplicações em criptografia e teoria de códigos**. [S.l.]: IMPA, 2007. Citado 2 vezes nas páginas 16 e 22.

MENEZES, A. J. A. J. **Handbook of applied cryptography**. Boca Raton: CRC Press, 1997. (Discrete mathematics and its applications. lat). ISBN 0849385237. Citado 4 vezes nas páginas 17, 32, 33 e 61.

MOUMNI, S. E.; FETTACH, M.; TRAGHA, A. High throughput implementation of SHA3 hash algorithm on field programmable gate array (FPGA). **Microelectronics Journal**, v. 93, p. 104615, 2019. ISSN 0026-2692. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0026269218308061>. Citado na página 36.

MUHAMMAD, Z. M. Z.; ÖZKAYNAK, F. An image encryption algorithm based on chaotic selection of robust cryptographic primitives. **IEEE Access**, v. 8, p. 56581–56589, 2020. Citado na página 28.

MULLEN, G. L.; PANARIO, D. **Handbook of finite fields**. [S.l.]: CRC press, 2013. Citado 2 vezes nas páginas 16 e 23.

NABEEL, N.; HABAEBI, M. H.; ISLAM, M. D. R. Security analysis of LNMNT-lightweight crypto hash function for IoT. **IEEE Access**, v. 9, p. 165754–165765, 2021. Citado 3 vezes nas páginas 29, 30 e 60.

NAGAMANI, K.; MONISHA, R. Physical layer security using cross layer authentication for AES-ECDSA algorithm. **Procedia Computer Science**, v. 215, p. 380–392, 2022. ISSN 1877-0509. 4th International Conference on Innovative Data Communication Technology and Application. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1877050922021111>. Citado na página 32.

NGUYEN, N. T. et al. Designing a pseudorandom bit generator with a novel five-dimensional-hyperchaotic system. **IEEE Transactions on Industrial Electronics**, v. 69, n. 6, p. 6101–6110, 2022. Citado na página 12.

NIST. **Cryptographic Standards and Guidelines: Examples with Intermediate Values**. 2016. <https://csrc.nist.gov/projects/cryptographic-standards-and-guidelines/example-values> [Accessed: (24 de Março de 2024)]. Citado na página 64.

NIU, T. et al. New constructions of involutions over finite fields. **Cryptography and Communications**, v. 12, p. 165–185, 03 2020. Citado 6 vezes nas páginas 13, 16, 24, 25, 53 e 54.

NOURA, H. et al. Lesca: Lightweight stream cipher algorithm for emerging systems. **Ad Hoc Networks**, v. 138, p. 102999, 2023. ISSN 1570-8705. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1570870522001718>. Citado na página 32.

OLADIPUPO, E. T. et al. An efficient authenticated elliptic curve cryptography scheme for multicore wireless sensor networks. **IEEE Access**, v. 11, p. 1306–1323, 2023. Citado na página 13.

PISTONO, M.; BELLAFQIRA, R.; COATRIEUX, G. Cryptosystem conversion, packing and matrix processing of homomorphically encrypted data: Application to IOT devices. **IEEE Access**, v. 9, p. 28302–28316, 2021. Citado na página 29.

PRENEEL, B. et al. Propagation characteristics of boolean functions. In: SPRINGER. **Advances in Cryptology—EUROCRYPT'90: Workshop on the Theory and Application of Cryptographic Techniques Aarhus, Denmark, May 21–24, 1990 Proceedings 9**. [S.l.], 1991. p. 161–173. Citado na página 37.

PUDI, V.; CHATTOPADHYAY, A.; LAM, K.-Y. Secure and lightweight compressive sensing using stream cipher. **IEEE Transactions on Circuits and Systems II: Express Briefs**, v. 65, n. 3, p. 371–375, 2018. Citado na página 32.

RAJSKI, J. et al. H2b: Crypto hash functions based on hybrid ring generators. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 43, n. 2, p. 442–455, 2024. Citado na página 12.

RASHIDI, B. Efficient full data-path width and serialized hardware structures of spongent lightweight hash function. **Microelectronics Journal**, v. 115, p. 105167, 2021. ISSN 0026-2692.

Disponível em: <https://www.sciencedirect.com/science/article/pii/S0026269221001750>. Citado 2 vezes nas páginas 29 e 45.

RIJNEVELD, J. **Readable-crypto**. [S.l.]: GitHub, 2014. <https://github.com/joostrijneveld/readable-crypto>. Citado na página 60.

SHAH, D. et al. Cryptographically strong S-P boxes and their application in steganography. **Journal of Information Security and Applications**, v. 67, p. 103174, 2022. ISSN 2214-2126. Disponível em: <https://www.sciencedirect.com/science/article/pii/S2214212622000588>. Citado na página 13.

SHANNON, C.; WEAVER, W. **The Mathematical Theory of Communication**. [S.l.]: University of Illinois Press, 1949. (Illini books, v. 1). ISBN 9780252725487. Citado na página 64.

SHARMA, M.; RANJAN, R. K.; BHARTI, V. A pseudo-random bit generator based on chaotic maps enhanced with a bit-XOR operation. **Journal of Information Security and Applications**, v. 69, p. 103299, 2022. ISSN 2214-2126. Disponível em: <https://www.sciencedirect.com/science/article/pii/S2214212622001521>. Citado na página 12.

SOUSSI, A. **QuarkPython**. [S.l.]: GitHub, 2020. <https://github.com/ayoubSoussi/QuarkPython>. Citado na página 60.

SREELAJA, N.; PAI, G. V. Stream cipher for binary image encryption using ant colony optimization based key generation. **Applied Soft Computing**, v. 12, n. 9, p. 2879–2895, 2012. ISSN 1568-4946. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1568494612001640>. Citado na página 32.

STALLINGS, W. **Cryptography and Network Security: Principles and Practice**. 8. ed. [S.l.]: Pearson Education Limited, 2023. ISBN 1292437480. Citado 5 vezes nas páginas 33, 60, 61, 63 e 64.

TILBORG, H. C. V.; JAJODIA, S. **Encyclopedia of Cryptography and Security**. 2nd ed.. ed. Germany: Springer, 2011. ISBN 978-1-44195905-8. Citado 2 vezes nas páginas 37 e 61.

TSUDIK, G. Message authentication with one-way hash functions. **ACM SIGCOMM Computer Communication Review**, ACM New York, NY, USA, v. 22, n. 5, p. 29–38, 1992. Citado na página 36.

UPADHYAY, D. et al. Investigating the avalanche effect of various cryptographically secure hash functions and hash-based applications. **IEEE Access**, IEEE, v. 10, p. 112472–112486, 2022. Citado na página 53.

WEBSTER, A. F.; TAVARES, S. E. On the design of s-boxes. In: SPRINGER. **Conference on the theory and application of cryptographic techniques**. [S.l.], 1985. p. 523–534. Citado na página 53.

WINDARTA, S. et al. Lightweight cryptographic hash functions: Design trends, comparative study, and future directions. **IEEE Access**, v. 10, p. 82272–82294, 2022. Citado 2 vezes nas páginas 28 e 33.

WINDARTA, S. et al. Two new lightweight cryptographic hash functions based on saturnin and beetle for the internet of things. **IEEE Access**, v. 11, p. 84074–84090, 2023. Citado 2 vezes nas páginas 12 e 28.

WU, X. et al. Artificial-noise-aided message authentication codes with information-theoretic security. **IEEE Transactions on Information Forensics and Security**, v. 11, n. 6, p. 1278–1290, 2016. Citado 2 vezes nas páginas 12 e 32.

XU, Z. et al. Linear cryptanalysis of SPECK and SPARX. **Journal of Information Security and Applications**, v. 83, p. 103773, 2024. ISSN 2214-2126. Disponível em: <https://www.sciencedirect.com/science/article/pii/S2214212624000760>. Citado na página 61.

YAN, H. et al. Spmac: Scalable prefix verifiable message authentication code for internet of things. **IEEE Transactions on Network and Service Management**, v. 19, n. 3, p. 3453–3464, 2022. Citado na página 12.

YOUSSEF, A.; TAVARES, S. E.; HEYS, H. A new class of substitution-permutation networks. In: **Workshop on Selected Areas in Cryptography, SAC**. [S.l.: s.n.], 1996. v. 96, p. 132–147. Citado 2 vezes nas páginas 13 e 24.

ZIEVE, M. E. Some families of permutation polynomials over finite fields. **International Journal of Number Theory**, World Scientific, v. 4, n. 05, p. 851–857, 2008. Citado 2 vezes nas páginas 23 e 24.