UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CIÊNCIA DA COMPUTAÇÃO

Paola de Oliveira Abel

Esquema de Assinatura Digital Tolerante a Modificações

Florianópolis
2023

Paola de Oliveira Abel

Esquema de Assinatura Digital Tolerante a Modificações

Trabalho de Conclusão de Curso submetido ao Curso de Graduação em Ciência da Computação do Centro Tecnológico da Universidade Federal de Santa Catarina como requisito para obtenção do título de Bacharela em Ciência da Computação.
Orientadora: Prof[a]. Thaís Bardini Idalino, Dr[a]
Coorientador: Gustavo Zambonin, Me.

Florianópolis
2023

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Abel, Paola de Oliveira
Esquema de Assinatura Digital Tolerante a Modificações / Paola de Oliveira Abel ; orientadora, Thaís Bardini Idalino, coorientador, Gustavo Zambonin, 2023.
76 p.

2. Ciências da computação. 3. Assinaturas digitais. 4. Cover-free families. I. Idalino, Thaís Bardini. II. Zambonin, Gustavo. III. Universidade Federal de Santa Catarina. . IV. Título.

Paola de Oliveira Abel

Esquema de Assinatura Digital Tolerante a Modificações

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de Bacharela em Ciência da Computação e aprovado em sua forma final pelo curso de Graduação em Ciência da Computação.

Florianópolis, 11 de Julho de 2023.

Prof. Lúcia Helena Martins Pacheco, Dr[a]

Coordenadora do Curso

Banca Examinadora:

Prof[a]. Thaís Bardini Idalino, Dr[a]

Orientadora

Universidade Federal de Santa Catarina

Gustavo Zambonin, Me.

Coorientador

Universidade Federal de Santa Catarina

Prof. Ricardo Felipe Custódio, Dr

Avaliador

Universidade Federal de Santa Catarina

Prof. Jean Everson Martina, Dr

Avaliador

Universidade Federal de Santa Catarina

RESUMO

Tradicionalmente, as assinaturas digitais são geradas de forma a verificar a integridade e autenticidade do arquivo assinado como um todo, não permitindo qualquer tipo de modificação ao documento. Esse esquema de construção de assinatura funciona para a maior parte dos propósitos, contudo não é possível flexibilizar a forma de validação, permitindo que certas áreas do documento sejam modificáveis ou detectar onde elas ocorreram. Por isso, para permitir a localização e correção de modificações em documentos assinados, foi criado o esquema de assinatura digital tolerante a modificações (MTSS) (IDALINO; MOURA; ADAMS, 2019), que aplica conceitos de group testing a assinaturas digitais a partir da utilização de *cover-free families*, que permitem realizar a verificação de agrupamentos específicos de resumos criptográficos vindos de diferentes partes do mesmo documento para permitir a localização das seções modificadas. Contudo, o funcionamento do esquema foi definido apenas de forma teórica, já que não houve implementação prática de seus algoritmos. Assim, o objetivo deste trabalho é estudar o esquema para criar uma aplicação prova de conceito para ele e analisar o seu funcionamento para obter resultados sobre sua viabilidade prática. O trabalho conclui que o MTSS funciona na prática para a aplicação desenvolvida. Ele não é tão eficiente quanto esquemas de assinatura tradicionais e possui algumas restrições acerca dos valores de entrada possíveis por causa das construções utilizadas para implementá-lo, porém ele permite localizar e até corrigir modificações ao invés de apenas descartar o documento caso ele tenha sido alterado, e a aplicação desenvolvida foi a primeira que permite mensurar na prática a escolha entre eficiência (de desempenho ou armazenamento) e capacidade de identificar e corrigir modificações.

Palavras-chave: Ciências da computação. Assinaturas digitais. *Cover-free families*.

ABSTRACT

Traditionally, digital signatures are created in order to verify the integrity and authenticity of the signed file as a whole, not allowing any king of modification to the document. This signature construction scheme is well suited for most purposes, however it is not possible to make the validation more flexible by allowing some parts of the document to be modified or to detect where alterations occurred. As a result, to allow for the location and correction of modifications in signed documents, the modification tolerant signature scheme (MTSS) was created (IDALINO; MOURA; ADAMS, 2019), that applies group-testing concepts coming from the usage of cover-free families, which allow for the verification of specific groupings of hashes from different parts of the same document to identify the modified sections. However, the behavior of the scheme was defined only in theory, since there was no implementation of its algorithms. Therefore, the purpose of this work is to study the scheme to create a proof of concept application and analyse its operation in practice to obtain results over its practical viability. In conclusion, the MTSS works in practice for the developed application. The scheme is not as efficient as traditional signature schemes and it has some restrictions over the possible entry values of the algorithms, because of the structures used in the implementation, however the scheme allows for localizing and even correcting modifications instead of just discarding the document if there have been alterations, and the development application is the first to measure in practice the trade-off between efficiency (regarding performance or storage) and the ability to identify and correct modifications.

Keywords: Computer science. Digital signatures. Cover-free families.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ALGORITMOS

LISTA DE SÍMBOLOS

$\{,\}$	Conjunto
$(,)$	Tupla
$\in$	Pertence ao conjunto
$\forall$	Quantificação universal
$\leftarrow$	Atribuição
:	Definição
$\lvert A \rvert$	Cardinalidade (número de elementos) do conjunto A
$\times$	Produto cartesiano
B_i	Elemento i do conjunto B
a_{ij}	Elemento da linha i e coluna j de uma matriz a
$\cup$	União de conjuntos
$\setminus$	Subtração de conjuntos
$\binom{n}{m}$	Número de combinações possíveis de m números de 1 até n; $m, n \in \mathbb{Z}$
$\lfloor n \rfloor$	Parte inteira de n; $n \in \mathbb{R}$
$\lvert\rvert$	Concatenação de caracteres
ε	*String* (sequência de caracteres) vazia
$\emptyset$	Conjunto vazio
$\wedge$	Conjunção lógica

SUMÁRIO

1 INTRODUÇÃO

As assinaturas digitais foram criadas para fornecer a validação de veracidade de documentos dentro do ambiente digital, visando manter as propriedades de segurança das assinaturas em papel e caneta. Tradicionalmente, as assinaturas digitais são geradas de forma a verificar a integridade e autenticidade do arquivo como um todo, não permitindo qualquer tipo de modificação ao documento e retornando apenas respostas de sim ou não para sua validade.

Esse esquema de construção de assinatura funciona para a maior parte dos propósitos, contudo não é possível flexibilizar a forma de validação, permitindo que certas áreas do documento sejam modificáveis, por exemplo. Essa construção também inviabiliza visualizar onde ocorreu a alteração, o que pode ser útil para identificar se houve falha acidental ou tentativa intencional de modificar o conteúdo para benefício próprio. Por fim, sem a localização de erros, também não se pode pensar em corrigi-los, o que tornaria os sistemas que precisam de arquivos assinados mais autônomos e resistentes a falhas caso possam recuperar os documentos originais de forma automatizada.

Para permitir a localização e correção de modificações em documentos assinados, foi criado o esquema de assinatura digital tolerante a modificações (com sigla em inglês MTSS) (IDALINO; MOURA; ADAMS, 2019), mas seu funcionamento foi definido apenas de forma teórica, já que não houve implementação prática de seus algoritmos para analisar sua viabilidade em aplicações reais. Dessa forma, este trabalho se propõe a estudar o esquema, implementá-lo e analisar o funcionamento prático.

O trabalho será organizado da seguinte forma: a seguir serão apresentados os objetivos do trabalho e sua metodologia de desenvolvimento, para então apresentar os conceitos teóricos necessários para entendimento do MTSS e sua implementação nas Seções 2 e 3. Depois, a Seção 4 começará discutindo as escolhas necessárias para a implementação do esquema e apresentação da aplicação prova de conceito. Também serão expostos os resultados obtidos a partir da execução prática da implementação desenvolvida e análise dos dados, para então concluir quais as melhores escolhas de parâmetros para a viabilidade prática do esquema na Seção 5.

1.1 OBJETIVOS

1.1.1 Objetivo geral

Analisar a viabilidade prática de um esquema de assinatura digital tolerante a modificações.

1.1.2 Objetivos específicos

i. Estudar o esquema de assinatura digital tolerante a modificações proposto em (IDALINO; MOURA; ADAMS, 2019);

ii. Analisar a viabilidade de implementação do esquema;

iii. Criar aplicação prova de conceito para o esquema;

iv. Obter resultados de desempenho e viabilidade do esquema via experimentação prática;

v. Propôr os parâmetros adequados para utilização prática do esquema.

1.2 METODOLOGIA

O trabalho será desenvolvido em etapas, começando pela revisão bibliográfica, estudo dos conceitos básicos acerca de assinaturas digitais e testagem de grupos, e estudo dos algoritmos que compõem o esquema de assinatura digital proposto em (IDALINO; MOURA; ADAMS, 2019). Após o estudo teórico, os conceitos aprendidos serão aplicados para determinar quais parâmetros são necessários para a implementação do esquema, como tamanho dos blocos, tipo de arquivo de entrada e algoritmo de resumo criptográfico utilizado, para então desenvolver uma aplicação prova de conceito. Então, serão realizados experimentos práticos utilizando a implementação desenvolvida para analisar a sua viabilidade prática e desempenho a partir de diferentes valores para os parâmetros existentes. Dessa forma, será possível determinar quais as melhores escolhas para permitir a viabilidade do uso do esquema.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo é dedicado a apresentar os conceitos e definições necessárias para entendimento do trabalho.

2.1 CRIPTOGRAFIA

A criptografia surgiu a partir da necessidade de transmissão segura de informação, ou seja, alguém "escutando" uma conversa confidencial de outras partes não deve conseguir entender a mensagem (GATHEN, 2015). Essa propriedade da criptografia é chamada de confidencialidade. Ao longo do tempo, a área da segurança da informação evoluiu para criar e combinar mecanismos além da criptografia para atingir outros objetivos, como autenticidade, integridade, não-repúdio etc.

Um dos objetivos principais da segurança da informação é a autenticidade, que serve para garantir a identidade do emissor de uma mensagem. Uma propriedade fortemente relacionada é a de não-repúdio, que garante que o emissor não pode negar a sua identidade posteriormente à comunicação. Outra propriedade importante a ser seguida é a de integridade, que está relacionada com a garantia que a mensagem original não sofreu alterações, seja por meio de erros ou modificações intencionais. Apenas criptografia não é suficiente para garantir a integridade da mensagem, mas certas ferramentas criptográficas permitem detectar se as modificações ocorreram (STINSON, 2018). A principal dessas ferramentas é o uso de funções de *resumo criptográfico*, que são usadas em conjunto a *assinaturas digitais* para combinar as propriedades de integridade e autenticidade, o que será explicado em detalhes nas seções seguintes.

2.1.1 Resumos criptográficos (*hashes*)

Uma função de resumo criptográfico, também conhecida como função de *hash*, é uma função que reduz uma mensagem, como por exemplo um documento digital ou um arquivo de formato qualquer, a um resumo (em inglês, *digest*) de tamanho fixo. Devido ao seu uso em aplicações digitais, essas funções trabalham com sequências binárias tanto na entrada quando na saída. Dessa forma, a mensagem é convertida para uma sequência de bits sem limitações de tamanho e é transformada em um resumo de n bits cujo tamanho não depende do comprimento da sequência original e que assume-se que foi produzido de forma que aparenta ser uniformemente aleatória, aproximadamente (MENEZES, 1997). O n escolhido costuma ser curto; 224 ou mais bits são tamanhos comuns (STINSON, 2018), contudo, ele deve ter tamanho suficiente para identificar a mensagem original de forma única, pelo menos na prática (GATHEN, 2015). Como a mensagem pode ter tamanho arbitrário, a entrada de uma função de *hash* costuma ter tamanho maior que a sua saída.

As funções resumo criptográfico são conhecidas como funções *one-way*, ou funções de sentido único. Isso porque convertem uma mensagem para uma sequência de bits que não

deve poder ser convertida de volta à entrada original, ou seja, dada uma função f, uma entrada x e um "resultado" da função $y = f(x)$ que ocorre dentro da imagem de f, não deve ser computacionalmente viável recuperar a entrada x que resultou em y (GATHEN, 2015). Ou seja, deve ser difícil achar a mensagem original a partir do resumo criptográfico. Essa propriedade também é chamada de *resistência à inversão* (GATHEN, 2015) ou de *resistência à pré-imagem*, pois, se for encontrado o x a partir da imagem $y = f(x)$, ele é denonimado como uma pré-imagem de y (STINSON, 2018).

O fato de uma função de resumo criptográfico ser uma função de sentido único é necessário para que ela seja usada para garantir a integridade de uma mensagem em assinaturas digitais, para que não seja possível forjar uma assinatura para uma mensagem (MENEZES, 1997, Seção 9.2.2). Para ser considerada segura para este propósito, a função ainda precisa respeitar mais duas propriedades, conforme descrito em (MENEZES, 1997, Seção 9.2.3).

A segunda propriedade necessária é chamada de *resistência à segunda pré-imagem*, ou seja, dada a mensagem x, não deve ser computacionalmente viável encontrar alguma outra mensagem x' diferente de x de forma que $f(x') = f(x)$. A terceira propriedade é chamada de *resistência à colisão*, e diz que não deve ser computacionalmente viável achar um par de mensagens x e x' tal que seu resumo criptográfico seja o mesmo (STINSON, 2018). As duas propriedades são similares, mas a diferença é que para existir segunda pré-imagem, é necessário encontrar a segunda pré-imagem x' que colida com uma pré-imagem x já determinada, enquanto que para haver colisão de forma geral, é necessário apenas existir algum par de mensagens x e x', com $x \neq x'$, que resultem no mesmo *hash*.

Existem relações matemáticas nessas últimas duas propriedades, o que leva à resistência a colisões implicar em resistência à segunda pré-imagem, mas a resistência a pré-imagem não tem implicação de "existência" baseada nas outras duas propriedades (MENEZES, 1997). Graças a esses relacionamentos, é possível construir funções de hash pondo mais ênfase na resistência a colisões ou no seu sentido único, o que faz com que os algoritmos de resumo criptográfico possam ser mais adequados a aplicações específicas, de acordo com o grau de segurança necessário vindo da aplicação de certas propriedades. Contudo, para os propósitos deste trabalho, a aplicação dos resumos criptográficos será apenas na garantia de integridade das mensagens em assinaturas assimétricas, e portanto as funções de hash que serão utilizadas devem ter segurança próxima à ideal considerando as três propriedades apresentadas.

A segurança de uma função de hash considerada segura depende do tamanho de sua saída, já que, como assume-se que ela segue as propriedades acima de forma ideal, então a melhor forma de achar alguma colisão ou pré-imagem é via o uso de força bruta. Para testar todas as possibilidades de saídas considerando o comprimento n do resumo em sequência binária, são necessárias 2^n operações, o que significa que resumos maiores são mais difíceis (ou até mesmo computacionalmente inviáveis) de serem atacados por esse método. Uma exceção à regra é a resistência a colisões, que mesmo em condições ideais precisa de apenas $2^{n/2}$ operações para encontrar uma colisão devido ao ataque do aniversário (STINSON, 2018, Seção 5.2.2). Dessa forma, uma função de hash é considerada como tendo *segurança ideal* se ela requer em torno

de 2^n operações para encontrar uma pré-imagem ou segunda pré-imagem e em torno de $2^{n/2}$ operações para encontrar uma colisão (MENEZES, 1997, Seção 9.3.4). A partir disso, pode-se definir o *nível de segurança n* de um algoritmo como sendo o número de operações necessário (2^n) para quebrar suas propriedades de segurança.

A Tabela 1 apresenta a segurança prática de algumas funções de *hash* da família SHA em relação ao comprimento *n* de suas saídas. Os valores nas colunas de "Força da Segurança em Bits" definem que aproximadamente 2^n operações de testagem por força bruta são necessárias para violar a segurança das propriedades correspondentes e o valor *m* é o comprimento da mensagem de entrada em bits. A tabela demonstra que as funções SHA2-256 e SHA2-512 perdem um pouco de resistência à segunda pré-imagem quanto maior for a mensagem de entrada (DWORKIN, 2015).

Função	Tamanho da Saída em Bits	Força da Segurança em Bits		
		Colisão	Pré-Imagem	Segunda Pré-Imagem
SHA2-256	256	128	256	$256 - log_2(m/512)$
SHA2-384	384	192	384	384
SHA2-512	512	256	256	$512 - log_2(m/1024)$
SHA3-256	256	128	256	256
SHA3-512	512	256	512	512

Tabela 1 – Níveis de segurança de diferentes funções da família SHA.

2.1.2 Assinaturas digitais

Assinaturas digitais são uma forma de garantir a autenticidade de uma mensagem digital usando assinaturas com identidades verificáveis de forma algorítmica, com o propósito de simular a segurança das assinaturas de documentos físicos, feitas com papel e caneta. De forma concreta, uma assinatura digital é um número, representado digitalmente pela sua sequência binária, que só pode ser gerado na prática usando-se dados exclusivos ao assinante e o conteúdo da mensagem assinada (MENEZES, 1997). Os requisitos para que assinaturas digitais possam substituir assinaturas físicas são que elas também devem ser firmemente anexadas ao documento que assinam, precisam impedir ou ao menos dificultar que identidades sejam roubadas com o fim de forjar assinaturas, devem impossibilitar que assinaturas realizadas sejam negadas posteriormente (ou seja, devem garantir a propriedade de *não-repúdio*), e precisam ter validade legal, o que necessita que a verificação de validade da assinatura seja realizável por terceiros. Para fins de conveniência, também é importante que o documento seja fácil de assinar pelos meios legítimos (ou seja, desconsiderando forjas) e que a assinatura resultante seja fácil de verificar (GATHEN, 2015).

As assinaturas digitais ajudam a garantir a identidade dos assinantes devido ao uso de criptografia assimétrica ou de chave pública, em que cada indivíduo ou entidade que deseja assinar documentos utiliza um par de chaves único, com uma chave pública e uma privada. A chave privada é utilizada para estabelecer a identidade do assinador no documento e criar

a assinatura, e portanto precisa ser armazenada de forma segura para que a autencidade não seja comprometida. Já a chave pública é utilizada para estabelecer a identidade do assinador para outras entidades, já que ela é disponibilizada publicamente para que qualquer um consiga verificar se assinatura foi criada usando a chave privada correspondente. Neste contexto, uma chave é um arquivo que contém uma sequência de bits que define quais são as assinaturas possíveis considerando a mensagem de entrada e o esquema de assinatura utilizado.

O esquema de assinatura é o que define o funcionamento das assinaturas digitais, já que ele especifica os métodos a serem usados para criar as chaves assimétricas, assinar mensagens e verificar as assinaturas. Um esquema de assinatura tradicional é composto por três funções que fornecem as funcionalidades acima, sendo elas as funções de geração de chaves, assinatura e verifição.

Um algoritmo de geração de chaves cria um par de chaves assimétricas, de tal forma que a chave privada consiga gerar um mapeamento um-para-um de mensagens possíveis para assinaturas resultantes quando usada de parâmetro para a função de assinatura. A chave pública é criada a partir da chave privada, de forma que consiga verificar as assinaturas da chave privada com base no mapeamento criado por ela sem precisar conhecê-la (MENEZES, 1997), quando usada de parâmetro para a função de verificação. O algoritmo recebe de entrada um parâmetro de segurança n, cujo uso na criação das chaves depende do esquema de assinatura utilizado. O funcionamento genérico de um algoritmo do tipo é apresentado no Algoritmo 1, adaptado a partir da definição de Menezes (1997, Algoritmo 11.4).

Um algoritmo de geração de assinatura cria uma assinatura digital a partir de uma mensagem e uma chave privada, enquanto que um algoritmo de verificação de assinatura verifica uma assinatura de acordo com a mensagem de entrada e uma chave pública. O comprimento da saída resultante do primeiro algoritmo pode variar de acordo com a mensagem de entrada; quando não se resume a mensagem antes de assiná-la, o comprimento da assinatura é pelo menos tão longo quanto o da mensagem original e pode ser ainda maior dependendo do esquema de assinatura utilizado (GATHEN, 2015, Seção 8.1). Dessa forma, foi adotado o método de calcular o resumo criptográfico da mensagem para usá-lo de entrada para o algoritmo de geração de assinatura. Assinar a mensagem utilizando seu *hash* será tão seguro quanto a função de resumo criptográfico utilizada, com o benefício de diminuir o tempo de computação e uso de memória. O funcionamento genérico dos dois algoritmos utilizando *hashes* é apresentado nos Algoritmos 2 e 3.

Um dos esquemas de assinatura mais utilizados hoje é o RSA, que surgiu para solucionar os problemas de compartilhamento prévio de chaves simétricas. O tamanho do par de chaves que ele gera depende de um parâmetro de segurança n (em bits), chamado de módulo da chave. Um n maior gera chaves maiores, que são mais difíceis de forjar, mas também geram assinaturas maiores, pois o comprimento das assinaturas geradas é igual ao valor de n (KALISKI, 1998). Para esse algoritmo, o tamanho das chaves precisa ser um múltiplo de 256, sendo que o tamanho mínimo recomendado pela NIST é 2048 bits, considerado como gerando nível de segurança estimado em 112 bits (BARKER, 2019). Esse esquema será utilizado na

implementação deste trabalho.

Outro esquema de assinatura que será utilizado na análise de viabilidade prática é o Ed25519. Ele é uma instância do algoritmo EdDSA, outro esquema de assinatura assimétrico cujos parâmetros de segurança dependem de qual *curva elíptica* é utilizada como parâmetro. O Ed25519 utiliza a curva elíptica Curve25519, que define para o algorimo um nível de segurança de 128 bits, comprimento de chave de 256 bits e comprimento de assinatura de 512 bits, porém essa instância específica do algoritmo requer o uso da função SHA512 para gerar os seus resumos criptográficos. Esse esquema tem como vantagens seu desempenho em operações para criar e verificar assinaturas, menor comprimento de assinatura e alto nível de segurança mesmo com maior velocidade e chaves menores comparado a outros algoritmos, como o próprio RSA (BERNSTEIN et al., 2012).

Devido ao uso de funções de resumo criptográfico que resumem toda a mensagem a ser assinada, as assinaturas digitais tradicionais só conseguem verificar a integridade da mensagem como um todo, e portanto não é possível localizar a modificação dentro da mensagem assinada; apenas é detectado se houve ou não alguma alteração. Caso a mensagem assinada tenha sido alterada após o processo de assinatura, o resumo h_m obtido no Passo 1 do Algoritmo 2 será muito diferente do resumo a ser verificado no Passo 2 do Algoritmo 3. Tal característica é a motivação do esquema de assinaturas apresentado por este trabalho, que permite a localização e correção de modificações ao construir o *hash* da mensagem baseando-se em construções de *cover-free families*.

Algoritmo 1 Geração de chaves genérica

Entrada: Um parâmetro de segurança n.
Saída: Um par de chaves (s_k, p_k).

1: Definir $assinar(m, s_k) \forall m \in \{0,1\}^*$
2: **para todo** $\sigma \in$ "possíveis assinaturas" **faça**
3: **para todo** $n \in \{0,1\}^*$ **faça**
4: Definir $verificar(\sigma, n, p_k) : verificar(\sigma, n, p_k) \leftarrow (\sigma = assinar(n, s_k))$
5: **fim para**
6: **fim para**
7: **retorne** (s_k, p_k)

Algoritmo 2 Geração de assinatura tradicional genérica

Entrada: Uma mensagem binária m, uma chave privada s_k.
Saída: Uma assinatura σ.

1: $h_m \leftarrow hash(m)$
2: $\sigma \leftarrow assinar(h_m, s_k)$
3: **retorne** σ

Algoritmo 3 Verificação de assinatura tradicional genérica

Entrada: Uma mensagem binária m, uma assinatura σ, uma chave pública p_k.
Saída: verdadeiro ou falso.

1: $h_m \leftarrow hash(m)$
2: **retorne** $verificar(\sigma, h_m, p_k)$

2.2 COVER-FREE FAMILIES

Group testing ("testagem de grupos") é um método de testagem em que os elementos são agrupados e testados em conjunto, ao invés de testados individualmente, com o propósito de se realizar menos testes para grandes números de elementos. O contexto histórico de sua criação vem da necessidade de testagem sanguínea de sífilis na Segunda Guerra Mundial, onde a ideia era agrupar várias amostras de sangue em um único teste, e caso esse conjunto de amostras não contivesse o antígeno de sífilis, então seria possível fazer o diagnóstico de diversas pessoas com um único teste. Em compensação, caso houvesse sinal de infecção na amostra, então todos os membros do grupo poderiam ser testados de forma individual para encontrar o portador (HWANG, 2000).

Apesar dessa exata aplicação não ter sido usada na prática, o conceito de *group testing* foi expandido de forma a ser utilizado em diversas aplicações, de acordo com as propriedades matemáticas dos agrupamentos de itens e testes. Neste trabalho, os testes serão definidos via objetos conhecidos por "*cover-free families*". Eles permitirão realizar a verificação de agrupamentos específicos de *hashes* para permitir a localização de modificações em um novo esquema de assinatura.

2.2.1 Definição e representações

Uma *d-cover-free family* (d-CFF) é uma família de conjuntos onde, ao unir quaisquer d conjuntos, nenhum dos outros conjuntos está contido nessa união. Ou seja, sempre existirão 1 ou mais elementos em todos conjuntos restantes que não pertencem à essa união. Ela é definida por um par $F = (X, B)$, em que X é um conjunto e B é um conjunto de subconjuntos de X, chamados de blocos. Uma d-CFF é denotada por d-CFF(t,n), sendo que a cardinalidade do conjunto X é representada por $|X| = t$, o número de blocos é representado por $|B| = n$, e ela é construída seguindo $d < t \leq n$, com o objetivo de utilizar o menor t possível para determinados d e n. O limite inferior teórico para t, quando $d \geq 2$ é dado por $t(d,n) \geq c\frac{d^2}{\log d}\log n$ para alguma constante c (WEI, 2006; RUSZINKÓ, 1994; FÜREDI, 1996), mas o limite real varia de acordo com o método utilizado para a construção da CFF. Já para $d = 1$, o limite é dado pelo Teorema de Sperner, como será apresentado na Seção 2.2.2.1.

Busca-se construir d-CFFs com o menor tamanho de X possível porque, conforme será detalhado a seguir, cada t_i é usado para representar um teste, e o propósito do *group testing* é agrupar os elementos para se economizar em números de teste, de forma que não seja necessá-

rio realizar n testes para n elementos. Para os propósitos desde trabalho, também é desejável maximizar o número possível de erros d para determinado número de blocos.

Uma d-CFF pode ser representada tanto por seu conjunto de blocos, listando-se todos os subconjuntos que compõem B, quanto por forma matricial com a sua matriz de incidência. Uma d-CFF(t,n) pode ser representada como uma matriz binária $t \times n$, também chamada de *d-disjunct matrix*, em que, para qualquer conjunto de $d+1$ colunas, a submatriz composta por essas colunas contém uma matriz de permutação de dimensão $d+1$. Tal representação é igual à matriz de incidência do sistema F correspondente, em que cada linha i corresponde a um ponto X_i de X, cada coluna j corresponde a um bloco B_i de B, e o valor binário a_{ij} indica se o ponto X_i está contido em B_j ou não. A Figura 1 mostra exemplos de representações de uma 1-CFF(4,6).

$$X = \{1, 2, 3, 4\}$$

$$B_1 = \{1, 2\}$$
$$B_2 = \{2, 3\}$$
$$B_3 = \{2, 4\}$$
$$B_4 = \{1, 3\}$$
$$B_5 = \{1, 4\}$$
$$B_6 = \{3, 4\}$$

$$\begin{array}{c} \\ X_1 \\ X_2 \\ X_3 \\ X_4 \end{array} \begin{array}{c} \begin{array}{cccccc} B_1 & B_2 & B_3 & B_4 & B_5 & B_6 \end{array} \\ \left[\begin{array}{cccccc} 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{array}\right] \end{array}$$

Figura 1 – Representação com subconjuntos, à esquerda, e representação matricial, à direita, de uma 1-CFF(4,6).

A utilização de *cover-free families* neste trabalho é baseada na sua aplicação em *combinatorial group testing* ("testagem de grupos combinatória"), ou CGT, em que subconjuntos de n itens são testados para presença ou não de defeitos dentre seus elementos. Se assume que no máximo d itens serão defeituosos, de forma que a combinação dos itens dentro dos t testes junto de seus resultados $\{verdadeiro, falso\}$ possibilita determinar exatamente quais itens causaram os erros. Caso existam mais que d itens com defeito, a testagem não será conclusiva.

Para realizar a testagem utilizando CFFs, o conteúdo das colunas da CFF agora representa os itens a serem testados, cada linha representa um teste, B representa o conjunto de itens e X representa o conjunto de testes. Se um teste der resultado verdadeiro, então nenhum dos itens envolvidos no teste tem defeito, o que significa que eles são livres de defeitos não importa em quais outros testes aparecerem. Dessa forma, é possível unir todos os itens contidos em testes verdadeiros e então subtraí-los do conjunto de itens total para encontrar os itens defeituosos. A Figura 2 ilustra o procedimento para a mesma CFF da Figura 3.

Quando se utiliza a representação matricial de CFFs, cada linha da matriz corresponde a um teste, cada coluna representa um dos itens sendo testados, e os valores binários dos elementos da matriz definem quais itens estão presentes em cada teste. Quando um teste tem resultado negativo, todos os seus itens podem ser potencialmente defeituosos, então os itens sem defeitos precisam ser determinados apenas pelas linhas cujos testes são positivos. Se há incidência de um item em um teste com resultado positivo, então toda a sua coluna pode ser desconsiderada

Itens = {a, b, c, d, e, f, g, h, i, j, k, l}

teste	itens em cada teste	resultado
1	{a, d, g, j}	✗
2	{a, e, h, k}	✗
3	{a, f, i, l}	✓
4	{b, d, i, k}	✗
5	{b, e, g, l}	✓
6	{b, f, h, j}	✗
7	{c, d, h, l}	✗
8	{c, e, i, j}	✓
9	{c, f, g, k}	✓

$$
\begin{array}{c} \\ \text{teste}_1 \\ \text{teste}_2 \\ \text{teste}_3 \\ \text{teste}_4 \\ \text{teste}_5 \\ \text{teste}_6 \\ \text{teste}_7 \\ \text{teste}_8 \\ \text{teste}_9 \end{array}
\begin{array}{c}
\begin{array}{cccccccccccc} a & b & c & d & e & f & g & h & i & j & k & l \end{array} \\
\left[\begin{array}{cccccccccccc}
\mathbf{1} & 0 & 0 & \mathbf{1} & 0 & 0 & \mathbf{1} & 0 & 0 & \mathbf{1} & 0 & 0 \\
\mathbf{1} & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & \mathbf{1} & 0 & 0 & \mathbf{1} & 0 \\
\mathbf{1} & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & \mathbf{1} & 0 & 0 & \mathbf{1} \\
0 & \mathbf{1} & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & \mathbf{1} & 0 \\
0 & \mathbf{1} & 0 & 0 & \mathbf{1} & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & \mathbf{1} \\
0 & \mathbf{1} & 0 & 0 & 0 & \mathbf{1} & 0 & \mathbf{1} & 0 & \mathbf{1} & 0 & 0 \\
0 & 0 & \mathbf{1} & \mathbf{1} & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & \mathbf{1} \\
0 & 0 & \mathbf{1} & 0 & \mathbf{1} & 0 & 0 & 0 & \mathbf{1} & \mathbf{1} & 0 & 0 \\
0 & 0 & \mathbf{1} & 0 & 0 & \mathbf{1} & \mathbf{1} & 0 & 0 & 0 & \mathbf{1} & 0
\end{array}\right]
\end{array}
$$

Não-defeituosos = $\text{teste}_3 \cup \text{teste}_5 \cup \text{teste}_8 \cup \text{teste}_9$
= {a, b, c, e, f, g, i, j, k, l}

Defeituosos = Itens \ Não-defeituosos = {d, h}

Figura 2 – *Group testing* usando uma 2-CFF(9,12) a partir da representação com subconjuntos dos testes, e representação matricial equivalente.

nos testes negativos. Por processo de eliminação, é possível remover dos testes negativos todos os itens que estão em testes positivos, até que restem apenas os itens defeituosos.

O procedimento é ilustrado pela Figura 3. No passo a), são pintadas em vermelho as linhas com testes negativos. Em b), são encontradas todas as incidências de itens em testes positivos. Em c), as incidências são usadas para desconsiderar os itens encontrados nos testes negativos, via sobreposição das linhas vermelhas por colunas verdes. Por fim, em d), as colunas que não foram sobrepostas indicam os itens defeituosos, o que também é comprovado por suas incidências em testes negativos (caso as colunas fossem vazias, não seria possível concluir que os itens têm defeito ou não).

Para o procedimento de *group testing* funcionar corretamente, é necessário assumir que as CFFs estão estruturadas corretamente e que o número de erros é igual ou menor ao número d previsto pela construção da CFF. A seguir serão descritas duas construções para CFFs, que foram escolhidas para este trabalho devido à facilidade de implementação, valores gerados de forma determinística e bons limites inferiores para o número de testes em relação ao ideal teórico.

2.2.2 Construções

2.2.2.1 1-CFFs

Quando $d = 1$, tem-se a construção mais simples para CFFs. Neste caso, a CFF é considerada um *sistema de conjuntos Sperner* ou *família de Sperner*, que é um sistema de conjuntos com subconjuntos disjuntos entre si. Considerando a representação matricial da 1-CFF, nenhuma de suas colunas pode conter o conteúdo de qualquer outra coluna (HWANG, 2000, Seção 7.5).

A construção de uma 1-CFF usando Sperner é apresentada a seguir. Dado n e $d =$

a)

	a	b	c	d	e	f	g	h	i	j	k	l
Teste 1 ✗	1	0	0	1	0	0	1	0	0	1	0	0
Teste 2 ✗	1	0	0	0	1	0	0	1	0	0	1	0
Teste 3 ✓	1	0	0	0	0	1	0	0	1	0	0	1
Teste 4 ✗	0	1	0	1	0	0	0	0	1	0	1	0
Teste 5 ✓	0	1	0	0	1	0	1	0	0	0	0	1
Teste 6 ✗	0	1	0	0	0	1	0	1	0	1	0	0
Teste 7 ✗	0	0	1	1	0	0	0	1	0	0	0	1
Teste 8 ✓	0	0	1	0	1	0	0	0	1	1	0	0
Teste 9 ✓	0	0	1	0	0	1	1	0	0	0	1	0

b)

	a	b	c	d	e	f	g	h	i	j	k	l
Teste 1 ✗	1	0	0	1	0	0	1	0	0	1	0	0
Teste 2 ✗	1	0	0	0	1	0	0	1	0	0	1	0
Teste 3 ✓	1	0	0	0	0	1	0	0	1	0	0	1
Teste 4 ✗	0	1	0	1	0	0	0	0	1	0	1	0
Teste 5 ✓	0	1	0	0	1	0	1	0	0	0	0	1
Teste 6 ✗	0	1	0	0	0	1	0	1	0	1	0	0
Teste 7 ✗	0	0	1	1	0	0	0	1	0	0	0	1
Teste 8 ✓	0	0	1	0	1	0	0	0	1	1	0	0
Teste 9 ✓	0	0	1	0	0	1	1	0	0	0	1	0

c)

	a	b	c	d	e	f	g	h	i	j	k	l
Teste 1 ✗	1	0	0	1	0	0	1	0	0	1	0	0
Teste 2 ✗	1	0	0	0	1	0	0	1	0	0	1	0
Teste 3 ✓	1	0	0	0	0	1	0	0	1	0	0	1
Teste 4 ✗	0	1	0	1	0	0	0	0	1	0	1	0
Teste 5 ✓	0	1	0	0	1	0	1	0	0	0	0	1
Teste 6 ✗	0	1	0	0	0	1	0	1	0	1	0	0
Teste 7 ✗	0	0	1	1	0	0	0	1	0	0	0	1
Teste 8 ✓	0	0	1	0	1	0	0	0	1	1	0	0
Teste 9 ✓	0	0	1	0	0	1	1	0	0	0	1	0

d)

	a	b	c	**d**	e	f	g	**h**	i	j	k	l
Teste 1 ✗	1	0	0	1	0	0	1	0	0	1	0	0
Teste 2 ✗	1	0	0	0	1	0	0	1	0	0	1	0
Teste 3 ✓	1	0	0	0	0	1	0	0	1	0	0	1
Teste 4 ✗	0	1	0	1	0	0	0	0	1	0	1	0
Teste 5 ✓	0	1	0	0	1	0	1	0	0	0	0	1
Teste 6 ✗	0	1	0	0	0	1	0	1	0	1	0	0
Teste 7 ✗	0	0	1	1	0	0	0	1	0	0	0	1
Teste 8 ✓	0	0	1	0	1	0	0	0	1	1	0	0
Teste 9 ✓	0	0	1	0	0	1	1	0	0	0	1	0

Figura 3 – Visualização do procedimento de identificação de itens defeituosos usando uma 2-CFF(9, 12).

1, o menor número de linhas possível para uma 1-CFF, de acordo com o número de colunas desejado, é determinado pelo Teorema de Sperner, em que t é igual ao menor t possível de forma que $\binom{t}{\lfloor t/2 \rfloor} \geq n$ seja verdadeiro (HWANG, 2000, Seção 7.5). Após t ser determinado, a 1-CFF é construída de forma que cada bloco é igual a um dos subconjuntos distintos de $\lfloor t/2 \rfloor$ elementos do conjunto $X = \{X_1, \ldots, X_t\}$. O conjunto B é formado por todos os subconjuntos de um tamanho fixo, e portanto nenhum $B_i \in B$ cobre nenhum outro $B_j \in B$. Desta forma, temos a propriedade requerida para uma 1-CFF. Esta construção dá uma 1-CFF(t, n) onde t se aproxima do limite inferior teórico de número de testes para $d = 1$, já que $t \approx \log n$.

A CFF da Figura 1 foi construída usando esse procedimento, em que o valor $n = 6$ foi predeterminado e t foi definido como sendo igual a 4 pois $\binom{4}{2} = 6$, o que torna $t = 4$ o menor valor de t que respeita a condição $\binom{t}{\lfloor t/2 \rfloor} \geq 6$. Neste exemplo, tem-se $X = \{X_1, X_2, X_3, X_4\}$ e $B = \{B_1, B_2, \ldots, X_6\}$, onde cada B_i é um subconjunto de X de tamanho $\lfloor t/2 \rfloor = 2$.

2.2.2.2 *Construção polinomial para d-CFFs*

Para CFFs com $d > 1$, existem uma variedade de construções, com diferentes limites inferiores para t e graus de complexidade. Neste trabalho, a construção escolhida para d-CFFs será a construção polinomial devido à facilidade de se montar os conjuntos necessários em relação a outras construções existentes e bom limite inferior para t considerando sua complexidade.

A construção da d-CFF se baseia no uso de polinômios cujos coeficientes são elementos de um determinado corpo finito. Um corpo finito é um conjunto de tamanho finito que possui as seguintes propriedades sob seus elementos e sob as operações binárias de adição e multiplicação: associatividade, comutatividade, distributividade e existência de elemento identidade e inverso da adição e multiplicação para todo elemento do corpo.

Corpos finitos são denotados por $\mathbb{F}_q$, sendo que q é o seu número de elementos, chamado de *ordem*. Para que se constitua um corpo finito, sua ordem q precisa ser uma potência de número primo. Uma potência de número primo é um número inteiro $q = p^j$, onde p é um número primo e j é um inteiro positivo ≥ 1. O corpo finito mais simples de ser construído é o corpo dos inteiros $mod\, p$, denotado por $\mathbb{Z}/p$, sendo que p precisa ser primo. Esse corpo tem ordem igual a p e os elementos do conjunto são os inteiros em $\{0,1,\ldots,p-1\}$. Esse é o corpo que será usado nos próximos exemplos, mas também existem corpos finitos com ordem não-prima que podem ser utilizados.

Para construir a d-CFF, é necessário um corpo finito $\mathbb{F}_q$, sendo q uma potência de primo, e um número inteiro $2 \leq k \leq q$. A partir de $\mathbb{F}_q$, é obtido $\mathbb{F}_q[x]_{<k}$, que será definido como o conjunto de todos os polinômios de uma única variável e grau menor que k: $a_0 + a_1x + a_3x^2 + \ldots + a_{k-1}^{k-1}$, cujos coeficientes $a_0, a_1, a_2, \ldots, a_k$ pertencem a $\mathbb{F}_q$. Por exemplo, se $\mathbb{F}_q = \mathbb{Z}/3$ e $k = 2$, teremos $\mathbb{F}_3[x]_{<2} = \{0,1,2,x,x+1,x+2,2x,2x+1,2x+2\}$.

Considere q uma potência de primo, k um inteiro positivo e $\mathbb{F}_q = \{a_1, a_2, \ldots, a_q\}$ um corpo finito. Definimos uma (X,B) como a seguir. $X = \mathbb{F}_q \times \mathbb{F}_q$. Para cada polinômio $p \in \mathbb{F}_q[x]_{<k}$ de grau menor que k, temos um subconjunto associado $B_p = \{(a_1, p(a_1)), \ldots, (a_q, p(a_q))\}$, e por fim definimos $B = \{B_p : p \in \mathbb{F}_q[x]_{<k}\}$. O resultado obtido é uma d-CFF(q^2, q^k), com $d \leq \lfloor (q-1)/(k-1) \rfloor$.

Como exemplo, consideramos $q = 3$, $k = 2$, $\mathbb{F}_3 = \{0,1,2\}$, $\mathbb{F}_3[x]_{<2}$ apresentado acima e $d \leq \lfloor (3-1)/(2-1) \rfloor = 2$. Abaixo apresentamos a representação matricial completa da 2-CFF(9,9) obtida, onde as linhas são rotuladas com elementos em $\mathbb{F}_3 \times \mathbb{F}_3$ e as colunas com os polinômios em $\mathbb{F}_3[x]_{<2}$ correspondentes. Para ilustração, considere a coluna rotulada pelo polinômio $p = x+1$. Neste caso, temos $B_{x+1} = \{(0, p(0) = 1), (1, p(1) = 2), (2, p(2) = 0)\}$.

	0	1	2	x	x+1	x+2	2x	2x+1	2x+2
(0,0)	**1**	0	0	**1**	0	0	**1**	0	0
(0,1)	0	**1**	0	0	**1**	0	0	**1**	0
(0,2)	0	0	**1**	0	0	**1**	0	0	**1**
(1,0)	**1**	0	0	0	0	**1**	0	**1**	0
(1,1)	0	**1**	0	**1**	0	0	0	0	**1**
(1,2)	0	0	**1**	0	**1**	0	**1**	0	0
(2,0)	**1**	0	0	0	**1**	0	0	0	**1**
(2,1)	0	**1**	0	0	0	**1**	**1**	0	0
(2,2)	0	0	**1**	**1**	0	0	0	**1**	0

Figura 4 – Representação matricial de uma 2-CFF(9,9) construída utilizando polinômios sobre corpos finitos.

3 ESQUEMA DE ASSINATURA DIGITAL TOLERANTE A MODIFICAÇÕES

Nesta seção é apresentado o objeto de estudo deste trabalho, o esquema de assinatura digital tolerante a d modificações apresentado em (IDALINO; MOURA; ADAMS, 2019, Seção 4). Ele permite a localização e correção de modificações em documentos assinados ao aplicar conceitos de *group testing* a assinaturas digitais, por meio da combinação de vários resumos criptográficos vindos do mesmo documento. Tal característica permite localizar alterações e o esquema também pode ser modificado para pemitir a correção de modificações indesejadas.

Um esquema de assinatura digital tolerante a d modificações, também sendo referido por sua sigla em inglês d-MTSS, divide a mensagem a ser assinada em blocos de tamanho arbitrário e permite identificar a integridade parcial do documento assinado caso quaisquer blocos sejam modificados, desde que o número de blocos modificados não ultrapasse d. É possível utilizar o esquema especificando quais blocos podem ser alterados, mas o foco deste trabalho está na implementação do caso genérico.

3.1 LOCALIZAÇÃO DE MODIFICAÇÕES

A localização das modificações é realizada ao se juntar a separação da mensagem em blocos com os testes de uma CFF. Ao invés do material a ser assinado ser composto apenas pelo hash da mensagem, agora ele é composto também pelos *hashes* de combinações de blocos da mensagem, de forma que a estrutura do material é montada a partir dos testes de uma d-CFF. A verificação da assinatura é realizada montando os testes da mesma CFF novamente para encontrar as combinações de blocos que permanecem inalteradas, e dessa forma saber exatamente quais blocos foram modificados de acordo com o mecanismo de *group testing*.

Para o funcionamento dos algoritmos de geração e verificação de assinatura desse esquema, é necessário que eles compartilhem o uso de algum esquema de assinatura tradicional, função de resumo criptográfico pública e matriz de incidência de uma d-CFF. O esquema de assinatura tradicional será usado para a geração de pares de chaves, assinatura das combinações de *hashes* de blocos junto do *hash* da mensagem original completa e subsequente verificação da assinatura. O MTSS tem sua segurança limitada pela segurança do esquema de assinatura e da função de resumo criptográfico utilizados (IDALINO; MOURA; ADAMS, 2019, Seção 5).

O funcionamento do algoritmo de geração de assinatura tolerante a d modificações é descrito em detalhes pelo Algoritmo 4. Ele recebe de entrada uma mensagem dividida em n blocos. Ele inicia criando os n *hashes* dos blocos da mensagem, que foram organizados anteriormente, pois o funcionamento do algoritmo não depende do tamanho dos blocos ou seu número, mas nesta descrição assume-se que são sequenciais. A seguir, são realizadas concatenações de blocos de acordo com a matriz de incidência. Considerando que cada concatenação de blocos representa um teste da CFF (ou linha da matriz) e cada bloco da mensagem representa uma coluna da matriz, os blocos são concatenados de forma que cada c_i tenha o conteúdo de um teste, ou seja, contenha os mesmos blocos na mesma combinação da matriz de incidência.

Por fim, é criado um *hash* para cada concatenação, assim como o resumo da mensagem original, e a sequência de todos os resumos dos testes junto do *hash* da mensagem são identificados no algoritmo pelo vetor T, que é então assinado, gerando σ'. A assinatura final é composta por $\sigma = (T, \sigma')$ para que seja possível a localização de erros na verificação da assinatura. O tamanho final da assinatura depende do comprimento da saída dos algoritmos de assinatura e resumo criptográfico utilizados assim como do número de testes da d-CFF. Sendo a o comprimento da saída do algoritmo de assinatura tradicional em bits, h o comprimento da saída da função de *hash* em bits e t o número de testes da CFF, o tamanho da assinatura do MTSS será igual a $a + h(t+1)$ bits. Dessa forma, para diminuir o tamanho da assinatura resultante sem comprometer a segurança do esquema ao utilizar algoritmos de assinatura ou *hash* com saídas menores, é importante minimizar t para o número de modificações de blocos permitidas e o número de blocos criados para a mensagem. O procedimento descrito é adicionalmente ilustrado pelas Figuras 5 e 6.

Algoritmo 4 Geração de assinatura tolerante a d modificações

Entrada: Um esquema de assinatura tradicional α, uma mensagem dividida em blocos $m = (m[1], m[2], \ldots, m[n])$, uma matriz de incidência A de uma d-CFF(t,n), uma função de resumo criptográfico *hash*, uma chave privada s_k.

Saída: Uma assinatura σ resistente à modificação de até d blocos de m.

1: $h_i \leftarrow hash(m[i])\ \forall\ 1 \leq i \leq n$
2: **para todo** $1 \leq i \leq t$ **faça**
3: $\quad c_i = \varepsilon$
4: $\quad$ **para todo** $1 \leq j \leq n$ **faça**
5: $\quad\quad c_i \leftarrow c_i || h_j$ se $a_{ij} = 1$
6: $\quad$ **fim para**
7: $\quad T[i] \leftarrow hash(c_i)$
8: **fim para**
9: $h_m \leftarrow hash(m)$
10: $T \leftarrow (T[1], T[2], \ldots, T[t], h_m)$
11: $\sigma' \leftarrow \alpha.assinar(T, s_k)$
12: $\sigma \leftarrow (\sigma', T)$
13: **retorne** σ

O funcionamento do algoritmo de verificação de assinatura tolerante a d modificações é descrito em detalhes pelo Algoritmo 5. Ele começa verificando a assinatura da junção de *hashes* dos testes e da mensagem, ao invés de verificar a assinatura da mensagem como um todo como no algoritmo tradicional. A verificação não irá funcionar caso algum resumo criptográfico em T, enviado junto da assinatura, tenha sido alterado, seja por erro ou alteração intencional. Se a assinatura é válida, então é possível afirmar se a mensagem foi alterada ou não. Como o *hash* h_m da mensagem foi assinado em conjunto aos testes, ele já foi verificado, então basta comparar o resumo enviado na assinatura com o *hash* da mensagem a ser verificada para determinar se a mensagem se manteve completamente íntegra. Caso exista diferença entre os dois resumos, então houve alteração na mensagem, e os passos seguintes do algoritmo servem para reconstruir as concatenações de blocos e compará-los com os enviados junto da assinatura. A localização

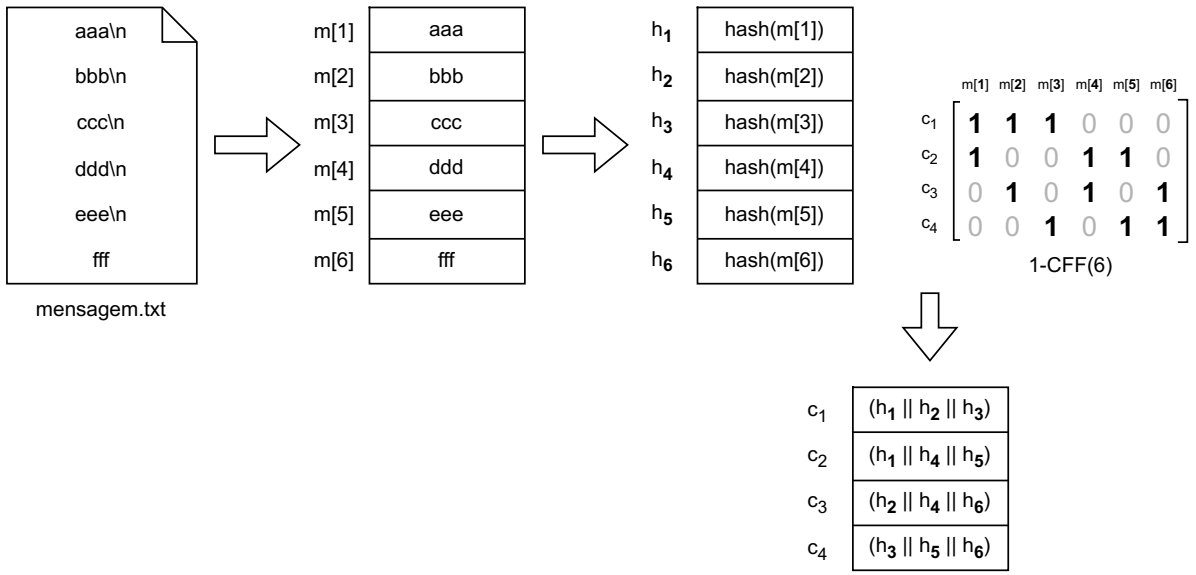

Figura 5 – Início do processo descrito para criação de assinatura, com o objetivo de assinar um documento de texto a partir de uma 1-CFF(4,6).

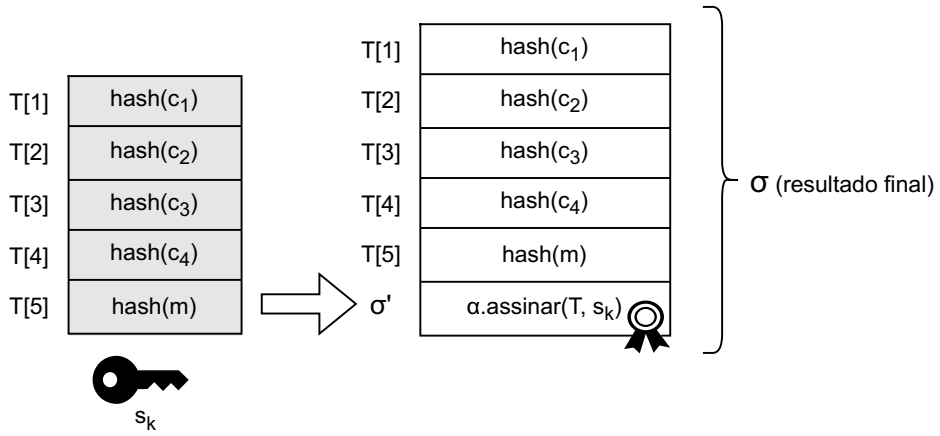

Figura 6 – Conclusão da criação de uma assinatura tolerante à modificações, continuando a partir da figura anterior.

das modificações é feita ao encontrar os índices dos blocos presentes em testes não modificados, e então subtraí-los do conjunto total de índices de blocos para encontrar os restantes, de forma similar aos procedimentos descritos na seção 2.2.1. Adicionalmente, o procedimento descrito acima é ilustrado pela Figura 8. A Figura 7 demonstra como a modificação do documento influencia na reconstrução das estruturas criadas na assinatura.

3.2 CORREÇÃO DE MODIFICAÇÕES

O esquema base MTSS pode ser modificado para permitir a correção de erros ao limitar o tamanho dos blocos do documento. O funcionamento da correção se dá via força bruta, em que os *hashes* de blocos modificados são recalculados utilizando todas as possibilidades de conteúdo até se encontrar as sequências binárias que fazem os testes corrigidos serem iguais

Algoritmo 5 Verificação de assinatura tolerante a d modificações ($verificar\_mtss$)

Entrada: Um esquema de assinatura tradicional α, uma mensagem dividida em blocos $m = (m[1], m[2], \ldots, m[n])$, uma assinatura $\sigma = (\sigma', T)$ com $T = (T[1], T[2], \ldots, T[t], h_m)$, uma matriz de incidência A de uma d-CFF(t,n), uma função de resumo criptográfico *hash*, uma chave pública p_k.

Saída: uma tupla (r, I), sendo r um valor booleano representando o resultado da verificação e I o conjunto de índices de blocos modificados.

1: **se** $\alpha.verificar(T, \sigma', p_k) = falso$ **então**
2: **retorne** $(falso, \emptyset)$
3: **fim se**
4: **se** $h_m = hash(m)$ **então**
5: **retorne** $(verdadeiro, \emptyset)$
6: **fim se**
7: $h_i \leftarrow hash(m[i])\ \forall\ 1 \leq i \leq n$
8: **para todo** $1 \leq i \leq t$ **faça**
9: $c_i = \varepsilon$
10: **para todo** $1 \leq j \leq n$ **faça**
11: $c_i \leftarrow c_i || h_j$ se $a_{ij} = 1$
12: **fim para**
13: $T'[i] \leftarrow hash(c_i)$
14: **fim para**
15: $V \leftarrow \emptyset$
16: **para todo** $1 \leq i \leq t$ **faça**
17: **se** $T[i] = T'[i]$ **então**
18: **para todo** $1 \leq j \leq n$ **faça**
19: $V \leftarrow V \cup \{j\}$ se $a_{ij} = 1$
20: **fim para**
21: **fim se**
22: **fim para**
23: $I \leftarrow \{1, \ldots, n\} \setminus V$
24: $r \leftarrow (|I| \leq d)$
25: **retorne** (r, I)

aos testes originais, considerando que já se sabe que os testes enviados junto da assinatura estão corretos devido à verificação prévia de sua assinatura. A limitação de comprimento dos blocos em s bits é necessária para o tempo de execução do algoritmo não ser inviável, já que, no pior caso, são necessárias 2^s operações para se encontrar todas as possibilidades de conteúdo de um bloco de s bits de comprimento (IDALINO; MOURA; ADAMS, 2019).

O funcionamento do algoritmo de verificação de assinatura tolerante a d modificações com correção de erros é descrito em detalhes pelo Algoritmo 6. Assim como no algoritmo de verificação sem correção, ele começa verificando a assinatura da tupla de *hashes* dos testes e da mensagem, mas fazendo o uso do Algoritmo 5 do MTSS, para encontrar os índices dos blocos modificados, ao invés de fazer a verificação de assinatura usando o esquema tradicional. Também há mudança no retorno do algoritmo, que agora inclui a mensagem corrigida. Caso o resultado da verificação seja falso, a correção retornada é vazia, para representar o erro na

computação. O restante do procedimento é executado para corrigir todos os blocos contidos em testes que falharam.

Para corrigir um bloco $m[k]$, é necessário encontrar o teste i (ou índice da linha da matriz de incidência da CFF) que contém apenas k dentre os blocos modificados, para que as modificações de outros blocos não influenciem na correção do bloco $m[k]$, e então calcular os resumos criptográficos dos blocos não modificados presentes no teste i (os blocos $m[g]$ no Algoritmo 6) para que sejam utilizados na reconstrução do teste com um novo valor para o bloco a ser corrigido. O conteúdo recebido de $m[k]$ não é utilizado para o cálculo do *hash* de i pois está alterado, e será utilizado em seu lugar sequências binárias calculadas por força bruta, que tenham tamanho até s, até que se encontre o valor que corrige o bloco. O bloco será considerado corrigido quando o teste i calculado a partir de seu novo valor for igual ao teste de mesmo índice recebido junto da assinatura, pois os testes assinados foram criados com os valores originais de todos os blocos. Quando todos os blocos forem corrigidos, o algoritmo retorna não só o resultado da verificação e os índices modificados, mas também uma nova versão da mensagem enviada, com todos os erros corrigidos.

Adicionalmente, o algoritmo detecta colisões nas correções, que em termos práticos significa duas ou mais *bitstrings* interpretadas como valores corretos para o mesmo bloco, utilizando o vetor booleano *corrigido*. Ele que assume o valor *verdadeiro* em determinada posição quando um bloco correspondente é corrigido, e a computação de correções possíveis não termina quando é encontrada uma resposta. Dessa forma, é possível que mais de uma sequência de bits resulte em um teste válido, e neste caso, o algoritmo retorna com resultado verdadeiro para a verificação, mas não envia uma correção já que ela não é única.

3.3 TRABALHOS RELACIONADOS

A localização de alterações em partes determinadas dos dados já foi proposta no contexto de funções de resumo criptográfico por (BONIS; CRESCENZO, 2011a) e (BONIS; CRESCENZO, 2011b) e no contexto de assinaturas digitais por (BIYASHEV; NYSSANBAYEVA, 2012) e (IDALINO et al., 2015). A correção de modificações é proposta por (BIYASHEV; NYSSANBAYEVA, 2012), mas permite a correção de um único erro. Os outros trabalhos não a propõem, porém utilizam técnicas utilizando *cover-free families* para gerar a informação redundante necessária para localizar modificações nos dados. Em (PöHLS, 2018), a divisão da mensagem em blocos é utilizada na criação de assinaturas em que blocos podem ser editados para esconder informações confidenciais.

Porém, nenhum dos trabalhos acima contém implementações de prova de conceito dos esquemas propostos, e portanto a contribuição desse trabalho para a literatura consiste na análise de viabilidade do esquema proposto em (IDALINO; MOURA; ADAMS, 2019), através da implementação de um esquema de localização e correção de até d blocos modificados de um documento assinado, para um limite superior d escolhido no momento da assinatura.

Algoritmo 6 Verificação de assinatura tolerante a d modificações com correção de erros

Entrada: Um esquema de assinatura tradicional α, uma mensagem dividida em blocos $m = (m[1], m[2], \ldots, m[n])$ de comprimento até s, uma assinatura $\sigma = (\sigma', T)$ com $T = (T[1], T[2], \ldots, T[t], h_m)$, uma matriz de incidência A de uma d-CFF(t,n), uma função de resumo criptográfico *hash*, uma chave pública p_k.

Saída: uma tupla (r, I, m), sendo r um valor booleano representando o resultado da verificação, I o conjunto de índices de blocos modificados e m a mensagem corrigida.

$(r, I) \leftarrow verificar\_mtss(\alpha, m, \sigma, A, hash, p_k)$
se $r = falso$ **então**
    retorne $(falso, I, \varepsilon)$
fim se
para todo $k \in I$ **faça**
    $i \leftarrow \{\exists\ i : a_{ik} = 1 \wedge a_{ij} = 0\ \forall j \in I \setminus \{k\}\}$
    $g \leftarrow \{g : a_{ig} = 1\ \forall g \in \{1, \ldots, n\} \setminus \{k\}\}$
    $h_g \leftarrow hash(m[g])$
    $corrigido[k] \leftarrow falso$
    para todo $b \in \{0,1\}^*, |b| \leq s$ **faça**
        $h_k \leftarrow hash(b)$
        $c_i = \varepsilon$
        para todo $1 \leq u \leq n$ **faça**
            $c_i \leftarrow c_i || h_u$ se $a_{iu} = 1$
        fim para
        $T'[i] \leftarrow hash(c_i)$
        se $T[i] = T'[i]$ **então**
            se $corrigido[k] = falso$ **então**
                $corrigido[k] = verdadeiro$
                $m[k] = b$
            senão
                retorne $(verdadeiro, I, \varepsilon)$
            fim se
        fim se
    fim para
fim para
retorne $(verdadeiro, I, m)$

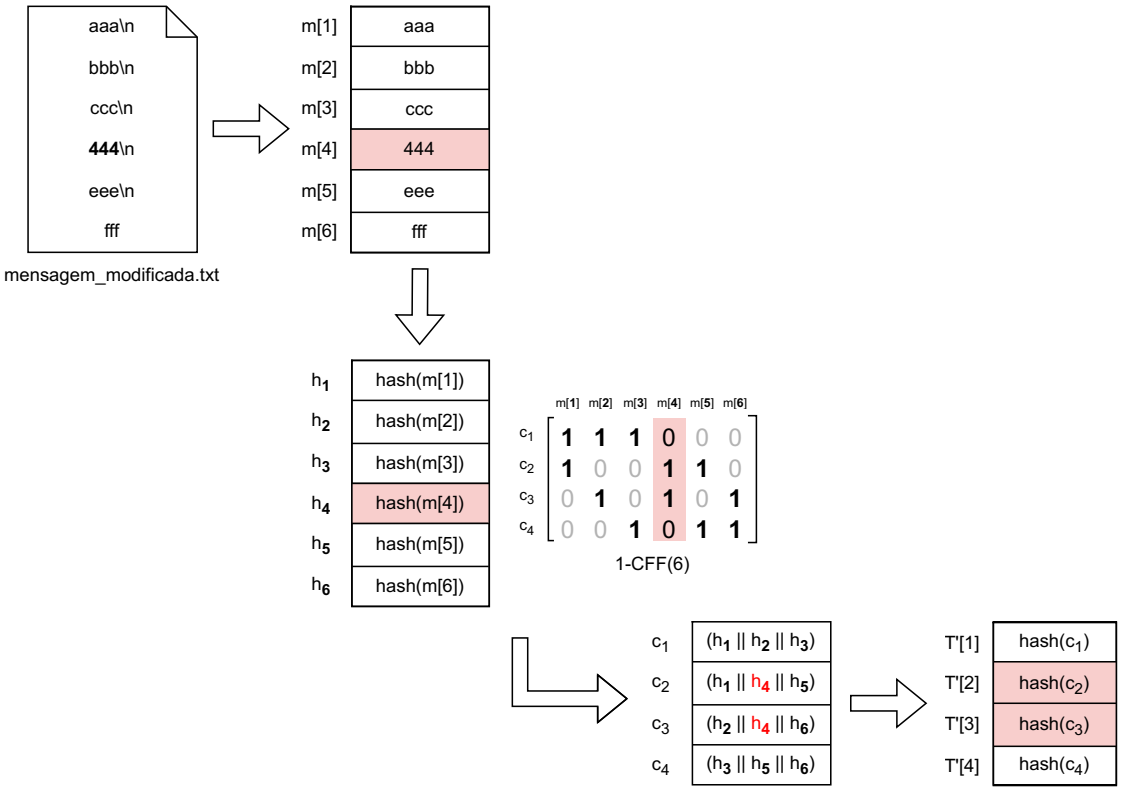

Figura 7 – Demonstração de como a modificação do documento influencia a reconstrução da estrutura T na verificação de assinatura, para a modificação do documento de exemplo das figuras 5 e 6.

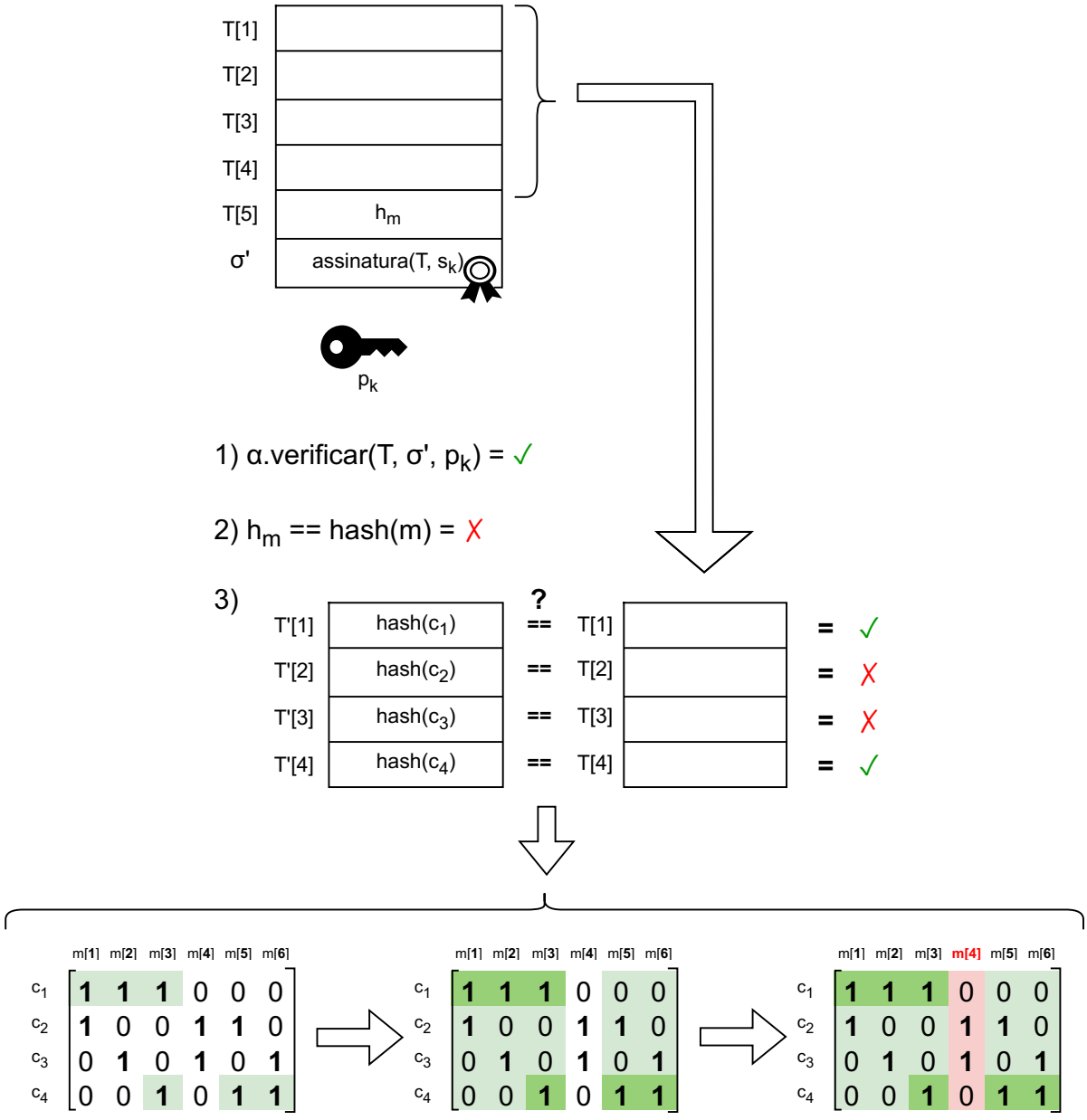

Figura 8 – Procedimento de verificação de assinatura e localização de modificações para a assinatura gerada nas Figuras 5 e 6, considerando o arquivo modificado e estruturas apresentadas na Figura 7.

4 ANÁLISE PRÁTICA E RESULTADOS

Esta seção apresenta a análise de viabilidade do esquema introduzido na Seção 3, feita por meio da análise acerca de questões de implementação, escolhas feitas para realizá-la e execução prática do MTSS a partir da aplicação desenvolvida para este fim.

O capítulo inicia discutindo questões acerca da formação de blocos da mensagem, utilizados como entrada dos algoritmos do MTSS. Em seguida, a implementação prática do esquema é descrita em mais detalhes e são especificadas as escolhas realizadas para seu desenvolvimento, para que então possam ser utilizadas em testes práticos de execução do MTSS.

A seguir, são esclarecidos detalhes de implementação da aplicação desenvolvida para testar o MTSS na prática, como linguagem e bibliotecas utilizadas, parâmetros de entrada e relação com questões de implementação previstas nos trabalhos anteriores. Na seção seguinte, é demonstrada a análise de desempenho das funções desenvolvidas para gerar CFFs, tanto de construção polinomial quanto utilizando famílias de Sperner.

Por fim, apresentam-se os resultados da experimentação prática com os algoritmos do MTSS utilizando a aplicação. Quando não especificado, os testes utilizados para montar as tabelas de resultados práticos usam par de chaves RSA de módulo 2048 para assinar, e como mensagem são usados documentos de texto com blocos de mesmo comprimento e conteúdo (após a divisão de blocos), com 1 caractere por bloco. Os experimentos incluem, por exemplo, análises de tempo de execução e comparações entre o MTSS e assinaturas tradicionais, sendo separados em subseções conforme cada funcionalidade do esquema (geração ou verificação de assinatura). Para distinguir quanto do tempo de execução obtido é responsabilidade do próprio algoritmo do MTSS, ao invés de ser afetado pela implementação da montagem da CFF, os resultados de tempo de execução para as Seções 4.4 a 4.6 leem as CFFs prontas a partir de arquivos de texto.

4.1 BLOCOS E TIPOS DE ARQUIVO

Os algoritmos apresentados na Seção 3 consideram a mensagem como já dividida em blocos, e portanto não fazem escolhas acerca o número ou tamanho dos blocos. Eles podem funcionar, em teoria, para qualquer número de blocos, sendo que a localização de modificações não depende do comprimento dos blocos, ao contrário da correção. Contudo, questões acerca da formação de blocos são importantes para as decisões de implementação, já que alteram o tamanho total da assinatura gerada, o tempo de execução do algoritmo e a granularidade dos erros detectáveis. Dessa forma, é importante analisar a viabilidade de diversos procedimentos para criação de blocos e considerar questões práticas das mensagens digitais, como formatos de arquivo, na hora de formá-los.

4.1.1 Número e tamanho dos blocos

A relação entre tamanho e número de blocos com as capacidades de localização e correção, além da alteração no tamanho da assinatura resultante, já estavam previstas a partir da construção teórica do esquema (IDALINO; MOURA; ADAMS, 2019, Seção 8). Em resumo, ao utilizar a construção polinomial de CFFs para gerar assinaturas com $d > 1$, para gerar assinaturas menores é desejável minimizar o número de blocos do arquivo, enquanto que, para corrigir modificações, é necessário limitar o tamanho dos blocos, o que tende a gerar muito mais blocos para o mesmo conteúdo. Além disso, blocos maiores levam a uma menor precisão na localização de erros, mas também levam a menos testes serem necessários para detectá-los. A utilização dessa construção também requer algumas restrições acerca do número de blocos que pode ser utilizado e qual o d máximo gerável.

A Tabela 2 a seguir mostra os números de blocos utilizáveis para CFFs polinomiais d-CFF(t,n), que precisam ser potências de primo seguindo a relação $t = q^2$ e $n = q^k$, sendo q qualquer potência de primo e k um inteiro igual ou maior que 2. Considerando blocos com 1 byte cada, um tamanho pequeno mesmo para arquivos de texto, 100 milhões de blocos seriam equivalentes a um arquivo de 100MB de tamanho. Para levar em conta a utilização comum do MTSS ou qualquer outro esquema de assinatura, que não costumam ser usados para assinar arquivos tão grandes ou com blocos tão pequenos, valores acima de 1 bilhão foram omitidos. Além disso, foi utilizada uma fonte mais clara para valores de blocos com número de testes inviáveis na prática, pois geram CFFs com $d < 2$, o que significa que geram mais testes do que os gerados por uma 1-CFF ideal, criada via famílias de Sperner. A Tabela 3 mostra algumas comparações práticas entre o número de testes necessário para criar 1-CFFs usando famílias de Sperner ou construção polinomial.

A análise da Tabela 2 mostra que valores de k acima de 7 não são bons para utilização prática, pois correspondem a um número muito alto de blocos ou a CFFs que não precisam utilizar construção polinomial. Ao comparar a coluna de número de testes com a segunda coluna, também percebe-se que CFFs polinomiais com $k = 2$ possuem o mesmo número de blocos e testes, o que gera matrizes quadradas. Dessa forma, não há economia de armazenamento entre a assinatura gerada por esse tipo de CFF e a gerada via resumo de todos os blocos da mensagem, que contém 100% de capacidade de localização de erros. Portanto, apesar de $k = 2$ ser utilizado nos testes e exemplos deste trabalho devido ao menor número de blocos e para fins de comparação, sua utilização prática não é recomendada.

A Figura 9 ilustra a relação $d = \lfloor (q-1)/(k-1) \rfloor$ para d-CFFs. A coluna $k = 2$ foi omitida devido a sua má usabilidade prática, e células com valores de d menores que 2 foram mantidas sem cor para dar destaque aos valores "de destaque", em que, quanto maior for d, mais intensa é a coloração. A figura demonstra que, para maximizar d, é mais importante utilizar parâmetros que resultem em menores valores para k, e também demonstra que valores de q menores que 5 não são boas escolhas práticas, a partir do fato que $k = 2$ já foi desconsiderado. Ao ser analisada em conjunto com a Tabela 2, demonstra-se que utilizar valores altos para k

t \ k	2	3	4	5	6	7	8
4	**4**	8	16	32	64	128	256
9	**9**	**27**	81	243	729	2187	6561
16	**16**	**64**	**256**	1024	4096	16384	65536
25	**25**	**125**	**625**	**3125**	15625	78125	390625
49	**49**	**343**	**2401**	**16807**	**117649**	**823543**	5764801
64	**64**	**512**	**4096**	**32768**	**262144**	**2097152**	**16777216**
81	**81**	**729**	**6561**	**59049**	**531441**	**4782969**	**43046721**
121	**121**	**1331**	**14641**	**161051**	**1771561**	**19487171**	**214358881**
169	**169**	**2197**	**28561**	**371293**	**4826809**	**62748517**	**815730721**
256	**256**	**4096**	**65536**	**1048576**	**16777216**	**268435456**	-
289	**289**	**4913**	**83521**	**1419857**	**24137569**	**410338673**	-
361	**361**	**6859**	**130321**	**2476099**	**47045881**	**893871739**	-
529	**529**	**12167**	**279841**	**6436343**	**148035889**	-	-
576	**576**	**13824**	**331776**	**7962624**	**244140625**	-	-
625	**625**	**15625**	**390625**	**9765625**	**387420489**	-	-
729	**729**	**19683**	**531441**	**14348907**	**594823321**	-	-
961	**961**	**29791**	**923521**	**28629151**	**887503681**	-	-
1024	**1024**	**32768**	**1048576**	**33554432**	-	-	-
1369	**1369**	**50653**	**1874161**	**69343957**	-	-	-
1681	**1681**	**68921**	**2825761**	**115856201**	-	-	-
1849	**1849**	**79507**	**3418801**	**147008443**	-	-	-
2209	**2209**	**103823**	**4879681**	**229345007**	-	-	-
2401	**2401**	**117649**	**5764801**	**282475249**	-	-	-
2809	**2809**	**148877**	**7890481**	**418195493**	-	-	-
3721	**3721**	**226981**	**13845841**	**844596301**	-	-	-
4096	**4096**	**262144**	**16777216**	-	-	-	-

Tabela 2 – Número de blocos (n) possíveis para n-CFFs usando construção polinomial, indexados por número de testes (t) e parâmetro k pela relação $n = t^{(k/2)}$. Valores superiores a 1 bilhão foram omitidos. Valores em cinza correspondem a CFFs com $d <= 1$, sendo que valores em cinza e negrito geram CFFs com $d = 1$.

não é interessante tanto no ponto de vista de tamanho de arquivos quanto no ponto de vista de aumentar a capacidade de localização de erros.

Para contornar a necessidade de gerar valores para n que sejam exclusivamente potências de primos para d-CFFs, é possível utilizar técnicas de arredondamento para a potência de primo mais próxima ou algum outro critério (como a próxima potência p^k que seja elevada a algum k desejado). Isso é feito sem necessidade de criar novos blocos, pois pode-se criar uma CFF com colunas extras, de modo a n ser potência de primo, e então descartá-las. Contudo, isso cria em uma matriz com um número maior de linhas do que de colunas, o que resulta em uma assinatura menos eficiente em armazenamento do que gerar uma assinatura resumindo todos os blocos individualmente e sem a vantagem de possuir 100% de blocos modificáveis ou número irrestrito de blocos de entrada. Portanto, não foi considerado vantajoso implementar o arredon-

q	k	n	t (Sperner)	t (polinomial)	% de aumento de t
2	2	4	4	4	0,00 %
3	3	27	7	9	28,57 %
4	3	64	8	16	100,00%
5	4	625	12	25	108,33%
5	5	3125	14	25	78,57 %
7	5	16807	17	49	188,24%
8	5	32768	18	64	145,00%
7	6	117649	20	49	113,04%
8	6	262144	21	64	255,56%
9	6	531441	22	81	204,76%
7	7	823543	23	49	166,67%
8	7	2097152	24	64	137,04%
9	7	4782969	25	81	268,18%
8	8	16777216	27	64	224,00%
11	7	19487171	27	121	179,31%
9	8	43046721	29	81	153,13%
11	8	214358881	31	121	348,15%
9	9	387420489	32	81	290,32%
11	9	2357947691	35	121	245,71%
11	10	25937424601	38	121	218,42%
11	11	285311670611	42	121	188,10%

Tabela 3 – Comparação de relação entre número de blocos (n) e número de testes (t) necessários para gerar 1-CFFs com famílias de Sperner (1-CFF(t,$n \approx \binom{t}{\lfloor t/2 \rfloor}$)) ou com construção polinomial (1-CFF($t = q^2$,$n = q^k$)).

damento de blocos no trabalho. Também é necessário considerar que, caso seja desejado apenas um bloco modificável, não há restrição no número de blocos que pode ser gerado via 1-CFFs utilizando famílias de Sperner.

Outra restrição que surge com o uso de CFFs para montar as assinaturas é a necessidade de manter o mesmo número de blocos quando a mensagem é modificada. O problema já era previsto (IDALINO, 2015, Seção 3.4), e algumas soluções propostas foram implementadas e serão discutidas em detalhes na Seção 4.1.2. A seção também discute em detalhes os problemas previstos de dividir a mensagem em blocos de comprimento pré-determinado.

4.1.2 Organização estrutural dos blocos

Há várias possibilidades de métodos para organizar e formar blocos a partir de uma mensagem. A forma mais simples de criar os blocos é apenas dividir a mensagem em seções de tamanho fixo de forma sequencial, por exemplo, criar um bloco para cada x bits do conteúdo. Essa abordagem pode ser utilizada para criar assinaturas mas causa muitos problemas caso ocorram modificações no documento, pois a inserção de qualquer bit a mais em um bloco faz com que se perca a referência de quais são os outros blocos originais (IDALINO, 2015,

q \ k	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	2	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
7	3	2	1	1	1	0	0	0	0	0	0	0	0	0	0	0
8	3	2	1	1	1	1	0	0	0	0	0	0	0	0	0	0
9	4	2	2	1	1	1	1	0	0	0	0	0	0	0	0	0
11	5	3	2	2	1	1	1	1	1	0	0	0	0	0	0	0
13	6	4	3	2	2	1	1	1	1	1	1	0	0	0	0	0
16	7	5	3	3	2	2	1	1	1	1	1	1	1	1	0	0
17	8	5	4	3	2	2	2	1	1	1	1	1	1	1	1	0
19	9	6	4	3	3	2	2	2	1	1	1	1	1	1	1	1
23	11	7	5	4	3	3	2	2	2	2	1	1	1	1	1	1
25	12	8	6	4	4	3	3	2	2	2	2	1	1	1	1	1
27	13	8	6	5	4	3	3	2	2	2	2	2	1	1	1	1
29	14	9	7	5	4	4	3	3	2	2	2	2	2	1	1	1
31	15	10	7	6	5	4	3	3	3	2	2	2	2	2	1	1
32	15	10	7	6	5	4	3	3	3	2	2	2	2	2	1	1
37	18	12	9	7	6	5	4	4	3	3	3	2	2	2	2	2
41	20	13	10	8	6	5	5	4	4	3	3	3	2	2	2	2
43	21	14	10	8	7	6	5	4	4	3	3	3	3	2	2	2
47	23	15	11	9	7	6	5	5	4	4	3	3	3	3	2	2
49	24	16	12	9	8	6	6	5	4	4	4	3	3	3	3	2
53	26	17	13	10	8	7	6	5	5	4	4	4	3	3	3	3
61	30	20	15	12	10	8	7	6	6	5	5	4	4	4	3	3
64	31	21	15	12	10	9	7	7	6	5	5	4	4	4	3	3

Figura 9 – Número máximo de modificações possível d para parâmetros $q > 1$ e $k > 2$ seguindo a relação $d = \lfloor (q-1)/(k-1) \rfloor$ para d-CFFs. Células com valores acima de 1 foram coloridos conforme seu valor.

Seção 3.4). O problema é ilustrado pela Figura 10, em que a inserção de um caracter no bloco 3 faz com que todos os blocos conseguintes sejam modificados.

Existem alternativas já discutidas à esse tipo de divisão. A estruturação mais básica para blocos é assumir que são sempre uma sequência formada linearmente a partir da mensagem, ou seja, uma lista, mas em (PöHLS, 2018, Seção 3.2) apresenta-se a possibilidade de blocos cuja ordem não importa ou de blocos organizados em forma de árvore. Além disso, em (IDALINO, 2015, Seção 3.4) foram propostas algumas metodologias de divisão de blocos. A primeira é usar delimitadores especiais para separar os blocos, como tags em um arquivo XML ou arquivos reservados em texto. A segunda é usar a própria organização de dados para separar os blocos, como em registros de bancos de dados, sendo também possível usar a semântica dos dados para aplicações específicas, como utilizar as seções de um documento. Adicionalmente, constatou-se a necessidade de acrescentar um cabeçalho ou outra construção no documento para

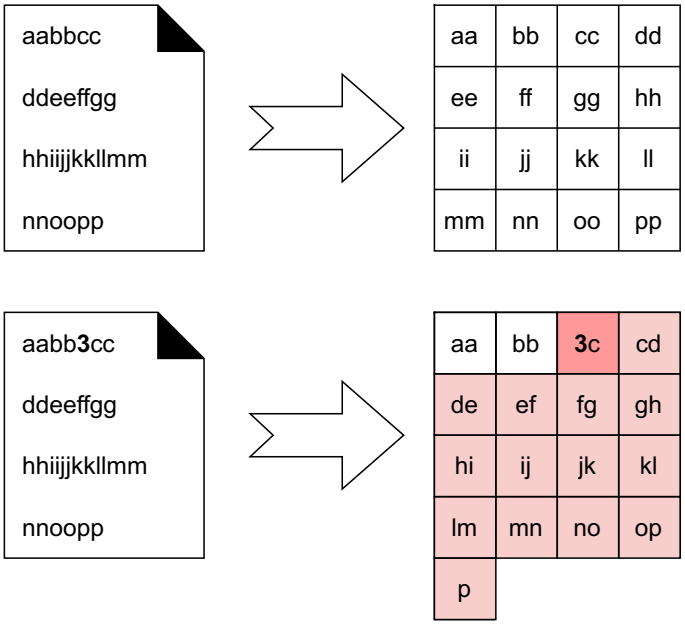

Figura 10 – Demonstração gráfica dos problemas na inserção (e por extensão, remoção) de caracteres em um procedimento de blocagem utilizando tamanho fixo (2 caracteres) por bloco. Na figura, cada célula das matrizes à direita representa um bloco.

descrever a estrutura de blocos.

Para atingir o objetivo de aplicação prática do MTSS, é útil analisar formatos de arquivos comumente usados para encontrar soluções de acordo com suas características próprias ao invés de criar novas metodologias do zero. Mesmo os arquivos para uso humano mais simples, como documentos em texto plano, são complexos o suficiente para conter caracteres de controle específicos a eles, como espaços, quebras de linha (`\n`) ou indentação (`\t`). Usar delimitadores próprios do formato da mensagem traz a vantagem do tamanho dos blocos não ser fixo, já que a divisão será feita por delimitador e o espaço entre eles pode variar, e dessa forma os blocos podem ter tamanhos variados entre si. Esse método é mais flexível para modificações do que criar blocos de tamanho fixo, pois as modificações podem variar o comprimento dos blocos sem alterar a capacidade de localização de erros, mas ainda é necessário manter o mesmo número de delimitadores. Caso seja utilizado apenas um método de divisão de blocos por tipo de arquivo, há a vantagem de não ser necessário um cabeçalho de informações pois o formato em si será correspondente à mesma divisão de blocos todas as vezes.

Em relação ao tamanho dos blocos, restrições adicionais surgem na correção de modificações, pois o Algoritmo 6 requer o teste de todas as possibilidades de correção por força bruta, ou seja, considerando um bloco de s bits, são necessárias 2^s operações para esgotar os possíveis valores do bloco. Experimentos práticos acerca da influência do tamanho dos blocos no tempo de execução do algoritmo serão mostrado em detalhes na Seção 4.7. Para tentar diminuir a influência do tamanho dos blocos nessa funcionalidade, pensou-se em pré-dividir os blocos em metades até atingir o tamanho desejado para então gerar e verificar a assinatura. Porém, conforme ilustra a Figura 11, tal procedimento só é viável caso as modificações resultem

no mesmo número de caracteres, pois caracteres adicionais ou faltantes podem gerar números diferentes de blocos ou então mover conteúdo de um bloco para outro, e portanto causar muito mais modificações do que o desejado.

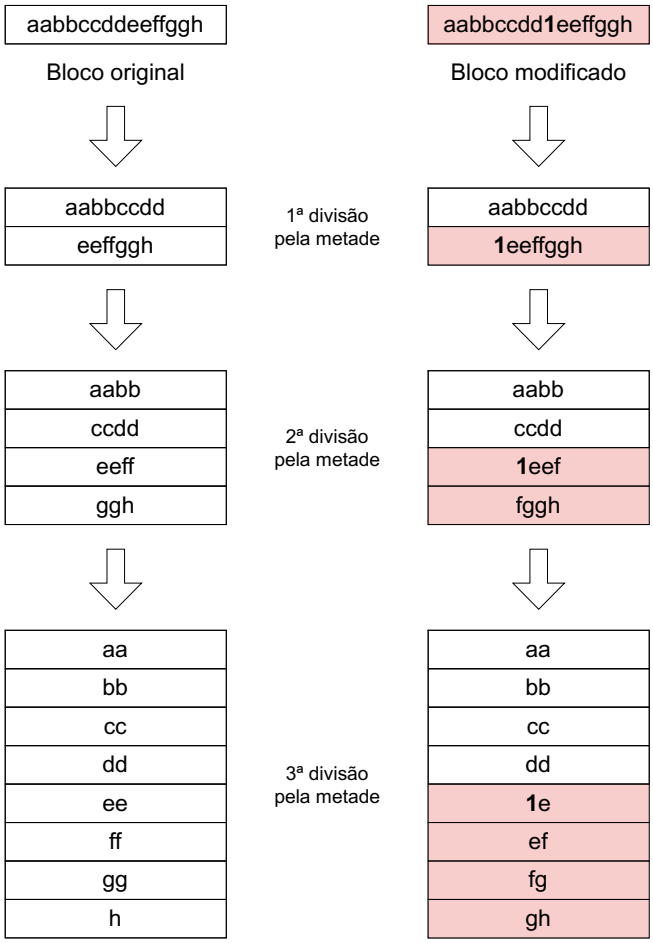

Figura 11 – Demonstração da inviabilidade de dividir blocos pela metade para alcançar um tamanho viável para a correção.

Considerando a restrição numérica, é importante escolher bem qual(is) delimitador(es) usar, pois sua frequência no documento irá ditar qual a granularidade de modificações que será detectável e também a restrição de *layout* do documento. Dependendo da aplicação ou arquivo, pode ser difícil modificar um documento de formato que ocasione número muito variável do(s) delimitador(es) escolhido(s), contudo, para arquivos mais estruturados, como documentos preenchidos com *templates* fixos, é possível que as modificações sejam bastante simples. Nas subseções a seguir são detalhados os procedimentos de divisão de blocos considerados para os tipos de arquivos aceitos pela aplicação desenvolvida e quais foram escolhidos na implementação final.

4.1.2.1 Arquivos de texto

Arquivos de texto são arquivos estruturados como uma sequência de caracteres, sendo que os bytes do documento serão convertidos para texto seguindo alguma codificação de carac-

teres. Esses documentos, considerados dados em texto plano, contrastam com tipos de dados em que é necessário interpretação do conteúdo binário além da simples conversão de caracteres para serem legíveis, como por exemplo arquivos PDF.

A codificação de caracteres mais utilizada hoje, de forma quase absoluta na Internet, é a UTF-8 (W3TECHS, 2023), em que são usados números variáveis de bytes para codificação dependendo do caractere. Os primeiros 128 caracteres em UTF-8 correspondem um a um aos caracteres da codificação ASCII, com 7 bits por caractere, que contém caracteres úteis para escrever a maioria das palavras em inglês e alguns símbolos gráficos utilizados em comunicação eletrônica. O restante dos caracteres UTF-8 é codificável por 2 a 4 bytes e consegue representar o restante dos alfabetos mundiais e outros símbolos (The Unicode Consortium, 2022). Os caracteres de texto são simbolizados por *code points*, que vão de U+0000 (primeiro caractere de 1 byte) até U+10000, sendo que os números são valores hexadecimais que representam a posição do caracter na lista da todos os caracteres, porém, esses números só correspondem exatamente ao valor binário do símbolo para os caracteres vindos do ASCII.

Devido a sua forte presença na comunicação digital de hoje, optou-se por utilizar apenas documentos codificados em UTF-8 ao invés de ASCII, que otimizaria a correção de modificações, como será detalhado na Seção 4.7.3, mas também tornaria a utilização da aplicação mais restrita.

A extensão mais comum para arquivos de texto é a extensão `.txt`, mas na verdade arquivos de qualquer extensão que sejam interpretáveis como texto plano pode ser lidos como arquivos TXT, como arquivos de código, CSV, XML, dentre outros, mas nesse caso não serão consideradas outras questões estruturais ou validações dos formatos específicos e os documentos serão lidos apenas como sequências de linhas.

Para manter a semântica do arquivo de texto como sendo uma sequência de linhas de caracteres, optou-se por implementar sua divisão em blocos usando os caracteres de controle de quebra de linha (`\n`). O procedimento escolhido é ilustrado pela Figura 12, onde cada linha do documento vira um bloco, e os delimitadores são removidos do conteúdo dos blocos pois são redundantes.

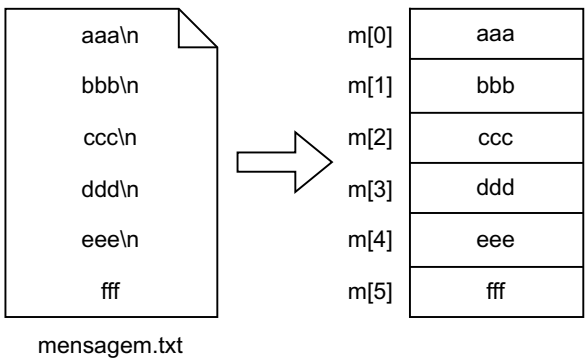

Figura 12 – Geração de blocos sequenciais implementada para documentos em texto plano, onde cada bloco equivale a uma linha do documento.

É possível utilizar quaisquer caracteres de texto para realizar a divisão, mas optou-se pelas quebras de linha pois assume-se que os documentos serão utilizados de fato para conter textos e portanto terão uma frequência regular desse delimitador. Além disso, dividir os blocos em linhas torna mais fácil definir quais modificações são possíveis e sua flexibilização, pois o número de linhas precisa se manter o mesmo mas o comprimento das linhas continua variável, e é mais intuitivo para o usuário identificar quando está adicionando ou removendo linhas.

Outra escolha interessante para esse formato de arquivo é dividir por caracteres de espaçamento, o que na prática faria com que cada palavra se tornasse um bloco. Isso resultaria em um número de blocos muito maior e uma maior restrição nas modificações, já que é difícil manter o mesmo número de palavras, mas a granularidade maior significaria identificar exatamente quais palavras foram modificadas.

Para arquivos em texto plano com formatos mais estruturados, as possibilidades aumentam. Por exemplo, para arquivos CSV, que representam registros de uma tabela separados por vírgulas, pode-se utilizar as próprias vírgulas como delimitadores, e dessa forma identificar exatamente quais células da tabela foram modificadas, como ilustrado pela Figura 13. O estudo de um caso especial de arquivo em texto plano, o arquivo XML, será apresentado a seguir, pois sua estruturação se adequa a procedimentos de criação de blocos mais complexos do que apenas uma leitura linear do conteúdo.

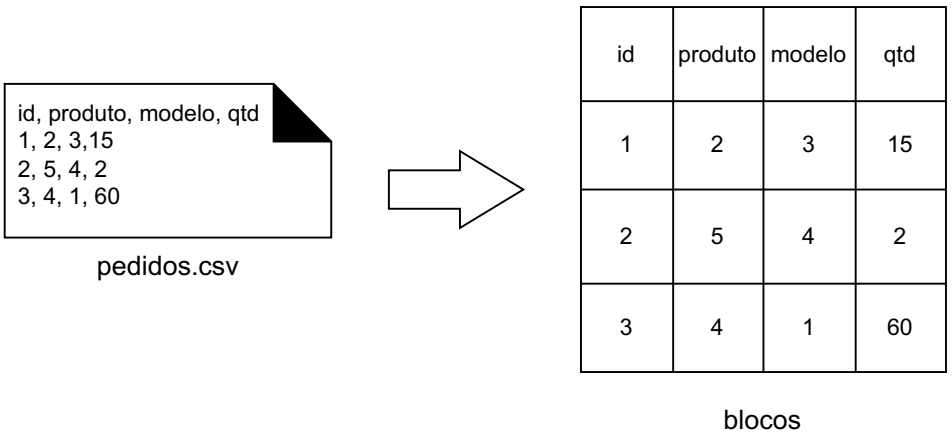

Figura 13 – Exemplo de divisão de arquivo CSV em blocos.

4.1.2.2 Arquivos XML

O XML (*Extensible Markup Language*) é um formato de arquivo e linguagem de marcação, onde são usadas estruturas chamadas de *tags* para categorizar, estruturar e formatar os dados do documento de forma facilmente legível tanto a texto plano quanto por aplicações digitais (W3C, 2006).

As *tags* podem ser de início/abertura (`<exemplo>`), de fim/fechamento (`</exemplo>`) ou *empty-element tags/self-closing tags* (`<exemplo/>`). O arquivo é composto de *elementos*,

que iniciam e terminam com um par de *tags* ou é apenas uma *empty-element tag*. Para o documento ser bem formado, todas as *tags* de abertura precisam ser pareadas com *tags* de fim de mesmo nome, e podem conter texto ou novos elementos entre elas. Caso um elemento esteja dentro de outros elementos, ele é chamado de *elemento filho*. Adicionalmente, para ser bem formado, o arquivo precisa ter um único par de *tags* como um *elemento raiz* que contém todos os outros elementos. *Tags* de início ou *self-closing* também podem ter *atributos*, que são pares de chave-valor dentro delas (por exemplo, `<img src="example.jpg" alt="example"/>`), cujos valores devem estar entre aspas. Adicionalmente, os documentos podem conter no seu início, antes da definição de elementos, uma *declaração XML*, que mostra informações como versão utilizada e codificação do documento (por exemplo, `<?xml version="1.0" encoding="UTF-8"?>`). Outro fator importante é que a indentação e quebras de linha entre os elementos são utilizadas para facilitar a leitura humana do formato, mas não são consideradas na hora de validá-lo.

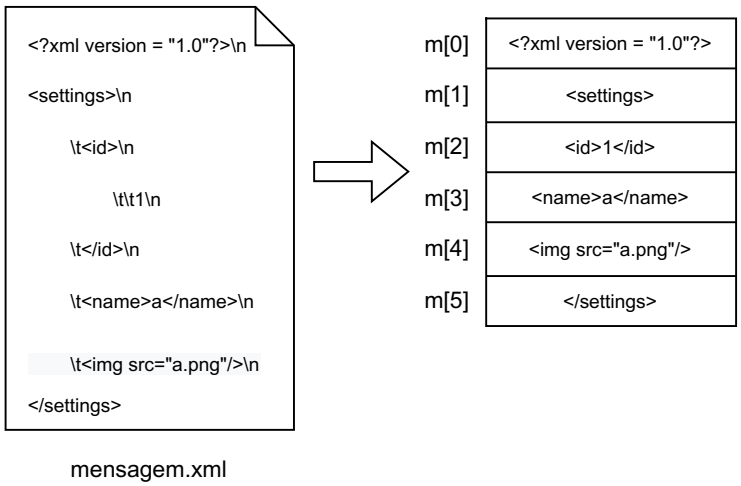

Figura 14 – Geração de blocos lineares implementada para arquivos XML, onde cada bloco contém uma *tag*.

Devido à sua presença obrigatória nesse tipo de arquivo, é natural utilizar as próprias *tags* como delimitadores ao separar a mensagem em blocos. De forma mais simples, pode-se utilizar o caractere "<" com procedimento similar ao ilustrado pela Figura 12, mas devido à redundância natural dos pares de *tags*, optou-se por implementar a divisão de blocos pareando as *tags* de abertura e fechamento no mesmo bloco, já que não se pode modificar apenas uma delas e permanecer com um arquivo de formato válido. Contudo, não se pode agrupar todos os pares "cegamente"para que não se perca a estrutura aninhada dos elementos. Para solucionar o problema, o par de *tags* dos elementos com filhos é mantido em blocos separados de forma a delimitar o fim do elemento. Esse procedimento, que foi o escolhido para ler esse formato de arquivo na aplicação desenvolvida para o trabalho, é ilustrado pela Figura 14. Uma desvantagem do método é que manter as *tags* em sua forma textual e possivelmente em pares gera blocos com grande número de caracteres (no mínimo três por bloco para abrigar o nome do elemento e os delimitadores "<" e ">"), o que pode impactar a correção de modificações nos blocos, conforme

será detalhado na Seção 4.7.

Adicionalmente, antes de criar os blocos, o arquivo é validado por um *parser* do formato, para garantir que é válido, e canonicalizado. A *canonicalização* para XML é uma operação que padroniza mudanças sintáticas do conteúdo que são logicamente equivalentes, ou seja, tais mudanças não alteram o significado ou conteúdo das informações contidas (W3C, 2013). Esse procedimento é útil na criação de assinaturas digitais, já que permite gerar resumos criptográficos que detectam, de forma geral (sem considerar o contexto específico da aplicação), apenas modificações relevantes para a semântica das informações.

A separação de blocos por *tags* descrita anteriormente interpreta o documento XML de forma similar a um arquivo de texto, onde os elementos são lidos sequencialmente de cima para baixo, e os blocos mantém o mínimo necessário de informações para que a estrutura possa ser reconstruída posteriormente caso a correção da mensagem seja esperada. Porém, devido à formatação do arquivo em elementos que podem conter novos elementos, a estrutura de um XML é melhor representada como uma árvore, onde cada nodo representa um elemento. Contudo, a estrutura em árvore do XML não garante a ordem dos filhos dos elementos por padrão, o que geraria problemas caso se queira criar um bloco por nodo. Para garantir a ordenação na criação dos blocos, é preciso ordenar os nodos da árvore, por exemplo por ordem alfabética do nome dos elementos e seus atributos e/ou filhos, e então escolher um algortimo de busca que resulte sempre na mesma ordem de nodos para a mesma árvore, para então criar um bloco para cada nodo visitado. Caso seja interessante utilizar algoritmos de busca para árvores binárias, também é possível transformar a árvore XML em uma árvore binária antes de percorrê-la. O procedimento descrito é ilustrado pela Figura 15. Esse tipo de separação de blocos pode ser usado para outros tipos de arquivos estruturados em árvore, como por exemplo arquivos JSON.

Uma das vantagens do procedimento é a de diminuir ou eliminar a redundância de informações dos pares de *tags*, e a estruturação em árvore também permite assinar e validar documentos cuja ordem dos elementos não seja garantida ou semanticamente importante, como por exemplo em documentos que representam retornos de buscas em bancos de dados. Caso seja necessário manter a ordem dos elementos, pode-se usar atributos numerados nas *tags* que definam a ordem esperada.

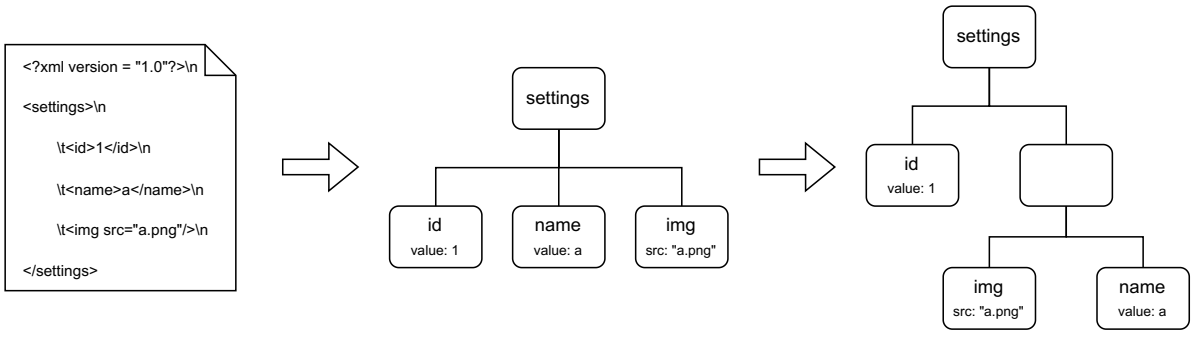

Figura 15 – Exemplo de formação de árvore binária ordenada alfabeticamente para arquivos XML.

4.1.3 Desempenho das implementações escolhidas

Nesta seção, é apresentado o desempenho das implementações escolhidas para divisão de blocos de texto e XML para compará-las em performance em relação ao tamanho do arquivo de entrada. Para a blocagem de arquivos de texto, os resultados obtidos para determinados números de blocos (n) serão diminuídos dos tempos de execução obtidos na execução dos algortimos do MTSS nas Seções 4.4 até 4.7, conforme o n utilizado, já que o desempenho desses algoritmos é dependente apenas indiretamente do método de divisão de blocos utilizado (pode ser afetado pelo número ou comprimento dos blocos gerados).

O comprimento em caracteres dos blocos (s) nesta seção e nas seções seguintes é avaliado em número de caracteres de 1 byte codificados em UTF-8. Para documentos de texto, cada linha de 1 caracter é codificada em 2 bytes pois a inserção de quebra de linha ($\backslash n$) é também um caracter UTF-8 de 1 byte, e logo blocos com $s = 1$ aumentarão o tamanho total do arquivo em 2 bytes, com exceção do último bloco. Para arquivos XML, o menor s foi estipulado como 7, considerando que os blocos do conteúdo principal do documento (filhos do elemento raiz) fossem pares de *tags* com nome de 1 caracter, seguindo a forma `<a><a/>`. Os documentos XML utilizados nos experimentos também contém declaração inicial com 23 caracteres, que é correspondente ao primeiro bloco do arquivo, e elemento raiz com nome de 5 caracteres. Além disso, os blocos também são separados por quebras de linha no documento original, mas não é utilizada indentação.

A Tabela 4 mostra os resultados de desempenho para criação de blocos de texto ou XML para arquivos com mesmo número de blocos, tamanho similar da mensagem, e $s = 7$, o mínimo para documentos XML, para os blocos conforme especificado anteriormente.

n	$\|m\|$ (`.txt`)	$\|m\|$ (`.xml`)	t. execução (`.txt`)	t. execução (`.xml`)	Δt.
64	511B	528B	0,0625 ms	0,7091 ms	0,6466 ms
512	4 kB	4,01 kB	0,0891 ms	6,1318 ms	6,0426 ms
4096	32 kB	32,01 kB	0,3195 ms	141,84 ms	141,5205 ms
32768	256 kB	256,02 kB	2,4778 ms	11,0652 s	11,06273 s

Tabela 4 – Tempo de execução médio (para 100 execuções) para criação de blocos de texto de acordo com número de blocos gerado (n), com 7 caracteres por bloco.

Os resultados mostram que o aumento de tempo de execução à medida que n cresce é muito maior para arquivos XML, devido à necessidade de canonicalizar a estrutura do documento e validar o formato antes de separar os blocos, além dos cálculos mais complexos para distribuir as *tags* corretamente. Assim, não há muita restrição para o tamanho dos arquivos de texto utilizados, porém, para arquivos XML, o ideal é não usar documentos com número de *tags* muito grande.

4.2 IMPLEMENTAÇÃO

Para realizar a análise de viabilidade prática do esquema apresentado anteriormente, ele foi implementado como prova de conceito usando Python 3.10 para facilitar o desenvolvimento. O resultado final da implementação foca em permitir a realização de testes acerca de todos as funcionalidades previstas pelo esquema, ou seja, seu objetivo não é garantir a maior otimização de desempenho ou usabilidade pelo usuário final possíveis. O código desenvolvido está disponível em repositório público no GitHub da autora (ABEL, 2023).

Como o objeto de estudo do trabalho é a avaliação do MTSS em si e não de seus algoritmos internos (usados para geração de assinaturas tradicionais e resumos criptográficos), o desenvolvimento da aplicação foi realizado utilizando implementações pré existentes de um esquema de assinatura tradicional e função de *hash* comumente utilizados. Para gerar assinaturas tradicionais utilizando chaves RSA, foi escolhido o esquema PKCS#1 v1.5 (KALISKI, 1998), utilizado devido à presença do esquema em bibliotecas criptográficas para Python. Para esse tipo de chave, um fator importante de segurança e desempenho que será avaliado é o tamanho das chaves utilizadas. Também há a opção de criar assinaturas utilizando o algoritmo Ed25519, que foi escolhido para análise devido a sua alta performance e alto nível de segurança mesmo com chaves consideradas pequenas. Já para a criação de resumos criptográficos, podem ser utilizadas como parâmetro as funções SHA2-256 (ou SHA256), SHA2-512 (ou SHA512), SHA3-256 e SHA3-512, com níveis de segurança apresentados na Tabela 1. Os algoritmos escolhidos serão comparados em desempenho nas próximas seções para que se entenda como suas diferenças se manifestam no seu uso prático com o MTSS. As implementações utilizadas de tais algoritmos vieram da biblioteca de código aberto PyCryptodome.

É preciso notar que tais escolhas foram feitas de forma a padronizar os estudos acerca do esquema novo, e que ele pode ser implementado utilizando quaisquer outros esquemas de assinatura ou funções de resumo criptográfico, sendo que sua segurança prática é ligada diretamente à versão e implementação dos procedimentos selecionados (IDALINO; MOURA; ADAMS, 2019). Também é importante considerar que implementações práticas podem permitir ataques não previstos pela definição original dos algoritmos do MTSS, e análises mais aprofundadas são necessárias para determinar os riscos possíveis.

Como parâmetros para controlar a criação da CFF que moldará a assinatura, é possível inserir o tamanho máximo em bytes desejado para a assinatura ou o parâmetro k da CFF, sendo que o número de blocos (n) é obtido diretamente do documento que se deseja assinar, a partir das divisões de blocos ilustradas pelas Figuras 12 ou 14, dependendo do tipo de arquivo. Caso o número de blocos gerado para a mensagem não seja uma potência de primo ou os parâmetros inseridos gerem uma 1-CFF de construção polinomial, optou-se por gerar a assinatura baseada em 1-CFF utilizando famílias de Sperner devido à maior eficiência de armazenamento, geração e verificação da assinatura, já que gera um número de testes (*hashes*) menor. Para usar essa construção, também implementou-se a possibilidade de inserir $k = 1$ como atalho, já que é um valor inválido para criação de CFFs polinomiais. O tamanho mínimo válido para gerar

uma assinatura depende do n gerado para a mensagem de entrada e do comprimento da assinatura necessário para o algoritmo de assinatura e chave privada escolhidos. Já a utilização do parâmetro k define o tamanho da assinatura gerada e a capacidade de localização de erros da assinatura, e pode ser usado para avaliar mudanças na viabilidade das operações do MTSS de acordo com a diferença ou variação dos parâmetros q ou k para criação de CFFs, porém esse parâmetro também determina quais valores de n podem ser utilizados, já que $n = q^k$. Devido à análise feita na Seção 4.1.1, não foi utilizado mecanismo de arredondamento de número de blocos para gerar sempre potências de primos independente do n da mensagem entrada.

Para auxiliar a geração dos polinômios e corpos finitos necessários para as CFFs de construção polinomial, foi utilizada a biblioteca de código aberto Galois. A aplicação possui funcionalidade de criação de CFFs para criar ou verificar assinaturas, mas como será mostrado na Seção 4.3, o desempenho da sua montagem utilizando essa biblioteca é baixo. Logo, para melhorar a performance dos algoritmos, desenvolveu-se a opção de ler CFFs pré montadas em arquivo de texto, que podem ser criadas utilizando o script `write_cff_to_file.py` e argumentos correspondentes a q e k, ou, para criar 1-CFFs de Sperner, n e $k = 1$.

A aplicação foi projetada para receber arquivos de texto (extensão .txt) ou arquivos XML, sendo que a extensão .txt pode ser utilizada para qualquer documento em texto plano (por exemplo, arquivos de código, CSV, e também XML). Os arquivos em formato PEM contendo as chaves precisam estar em formato PKCS#1 para chaves RSA ou PKCS#8 para chaves Ed25519, devido a limitações da biblioteca criptográfica utilizada. Novas chaves neste formato podem ser geradas usando a aplicação OpenSSL (necessário ser versão 1.1 para chaves RSA) ou com o programa `key_generator.py`, disponível no repositório. Caso a chave privada seja criptografada e não consiga ser lida pela aplicação diretamente, o OpenSSL é chamado internamente para que seja possível acessar o conteúdo do arquivo.

Para tornar viável a correção de modificações, foi necessário o uso de computação multiprocesso para paralelizar a correção via força bruta para o Algoritmo 6. Dessa forma, o desempenho dessa funcionalidade depende do número de núcleos físicos e virtuais disponíveis na máquina do usuário, além da memória RAM disponível para os núcleos virtuais. Também é uma possibilidade corrigir mais de um bloco de forma paralela, mas devido a limitações na máquina do desenvolvimento optou-se por apenas paralelizar o cálculo dos *hashes*.

Em (IDALINO; MOURA; ADAMS, 2019, Seção 7), são discutidas algumas abordagens para detalhes de implementação do esquema. Nesta implementação, optou-se por manter a criação do *hash* individual de cada bloco, de acordo com os algoritmos originais. Foram utilizados atributos globais na correção de modificações para não ser necessário recomputar parâmetros comuns aos Algoritmos 5 e 6, como a CFF recriada a partir da mensagem e assinatura, os blocos da mensagem e seus resumos criptográficos, dentre outros. Já em relação à detecção de colisões, o vetor `corrigido` foi mantido para registrar se algum bloco foi corrigido, no caso de erros na aplicação, mas por questões de desempenho optou-se por parar a busca ao ser encontrado qualquer valor que satisfaça a igualdade de *hashes*. O objetivo original era utilizar a abordagem de modificação parcial, em que, caso uma colisão seja encontrada, o

resultado alternativo é registrado em arquivo de *log* e a execução continua para o restante dos blocos modificados, mas para isso, é necessário testar todos os valores possíveis independente do conteúdo, o que é causa muita perda de desempenho conforme resultados da Seção 4.7.2.

4.3 DESEMPENHO DA CRIAÇÃO DE CFF

A Tabela 5 avalia a função de criação de 1-CFFs de Sperner para diferentes números de blocos, e mostra o aumento percentual entre cada linha das colunas de tempo de execução e número de operações. O número de operações é igual a $t * n$, que representa o número de operações necessárias para percorrer toda a CFF obtida de forma sequencial. A tabela mostra que o número de operações impacta no tempo de execução obtido, mas não parece ser o único fator relevante.

n	t	t. execução	aumento t.	nº operações	aumento op.
8	5	0,0351 ms	-	32	-
64	8	0,0804 ms	129,0859%	448	1300 %
216	10	0,2435 ms	202,8511%	1.944	333,9286%
512	12	0,7101 ms	191,6261%	5.632	189,7119%
1000	13	1,7593 ms	147,7584%	12.000	113,0682%
1728	14	3,0096 ms	71,0684 %	22.464	87,2 %
4096	15	6,3531 ms	111,0930%	57.344	155,2707%
8000	16	13,5312 ms	112,9844%	120.000	109,2634%

Tabela 5 – Tempo de execução médio (para 100 execuções) da função `create_1_cff(n)` e comparação de aumentos percentuais entre número de operações e tempo de execução obtido.

A Tabela 6 avalia a função de criação geral de CFFs, que escolhe entre criar CFF de Sperner ou de construção polinomial de acordo com o d obtido para os parâmetros q e k. As células da tabela em cinza correspondem CFFs de forma não polinomial pois possuem $d < 2$, sendo que os valores omitidos correspondem a CFFs com $d < 1$. Considerando o funcionamento da biblioteca Galois, que usa o máximo de processamento do computador para avaliar os valores dos polinômios com coeficientes dentro de $\mathbb{F}_q[x]$, os resultados dependem bastante da capacidade de processamento disponível.

A tabela demonstra que o tempo de criação das d-CFFs é mais influenciado pelo aumento de k do que de q por si só, mas o incremento de tempo causado pelo aumento exclusivo de q é maior conforme k aumenta. Esse padrão é similar ao obtido pela Tabela 2, que mostra a variação no número de blocos gerados de acordo com o incremento de q ou k. Devido aos resultados obtidos, concluiu-se que é obtido desempenho melhor pelos algoritmos do MTSS se eles receberem a CFF previamente montada a partir de um arquivo de texto.

q \ k	2	3	4
2	0,0495ms	-	-
3	0,9305 s*	0,0603 ms	-
4	1,4515 s*	0,0951 ms	0,3127 ms
5	0,9990 s*	1,3373 s*	0,8270 ms
7	1,1633 s*	2,8122 s*	14,2711 s*
8	1,8975 s*	5,1146 s*	32,8299 s*
9	2,5297 s*	8,6674 s*	55,0101 s*
11	2,4679 s*	19,5029 s*	172,4530 s*

Tabela 6 – Tempo médio de execução da função `create_cff(q,k)` para 100 execuções.

* - Valor provisório, para 5 execuções, que será corrigido na versão final.

4.4 GERAÇÃO DE ASSINATURA

Para a geração de assinaturas, é interessante analisar formas de reduzir o tamanho da assinatura final e como os parâmetros do algoritmo afetam não só o seu tamanho mas também o seu tempo de execução.

Considerando os dados apresentados pela Tabela 6, onde são avaliadas as diferenças de desempenho para criar CFFs polinomiais de acordo com a variação de q e k, é interessante comparar como os mesmos parâmetros afetam a criação de assinaturas. Na Seção 4.4.1 é analisado como q e k alteram a *compressão* de tamanho da assinatura, ou seja, como esses parâmetros podem diminuir o seu tamanho final quando comparada a uma assinatura com 1 *hash* por bloco.

Já em relação a diferenças de tempo de execução, a análise de desempenho para k fixo e variação de q, e também a análise com q fixo e variação de k, estão na Seção 4.4.2, sendo que, devido à mudança dos parâmetros, são assinados arquivos com diferentes números de blocos. Por isso, na Seção 4.4.3 é feita a comparação de tempo de execução para o mesmo arquivo mas com parâmetros q e k diferentes, de forma a variar o número de testes gerados para o mesmo arquivo.

Nas seções seguintes, a geração de assinatura é avaliada em relação ao uso de diferentes algoritmos de assinatura tradicional e funções de resumo criptográfico, de forma a comparar como os algoritmos diferentes afetam o seu desempenho e qual a diferença de tamanho e tempo de execução entre assinar o mesmo arquivo usando o MTSS e esquemas de assinatura tradicionais.

Por fim, na Seção 4.4.6 é determinado o número máximo de testes para diferentes tamanhos desejados de assinatura, com variados algoritmos de *hash* e assinatura tradicional, de forma a obter o d máximo possível para tamanhos limitados de assinatura e uso de k viável na prática, e demonstrar como os algoritmos escolhidos impactam nesse resultado.

4.4.1 Relação entre parâmetros q e k e compressão da assinatura

Como o objetivo do MTSS é localizar as modificações com o menor tamanho da assinatura possível, comparado ao tamanho obtido ao apenas fazer o resumo criptográfico de todos os blocos, é desejável aumentar a diferença entre o número de blocos (n) e número de testes (t) de forma a minimizar t e assim gerar menos *hashes*. Para CFFs de construção polinomial, a proporção entre n e t é dada pela relação q^{k-2}, e a partir dela pode se definir a taxa de compressão da assinatura (em relação a uma assinatura com 1 *hash* por bloco) como sendo $q^{k-2} : 1$. Por exemplo, para uma CFF de 125 (5^3) blocos, a taxa de compressão é 5:1. Como para $k = 2$ o número de blocos e de testes é o mesmo, não há compressão de tamanho (a taxa é de 1:1) para qualquer valor de q.

Essa relação é ilustrada pela Figura 16, onde os valores representam quantos blocos por teste são possíveis para determinados q e k. Os valores omitidos correspondem a CFFs que geram um número muito grande de blocos, conforme análise da Tabela 2. As células da tabela foram coloridas de forma a demonstrar graficamente a variação da compressão de acordo com o aumento dos parâmetros q e k, onde cores mais intensas representam uma maior compressão da assinatura, o que é desejável. A partir da figura, percebe-se que aumentar k traz um aumento mais significativo na taxa de compressão do que apenas aumentar q, porém, como já analisado anteriormente, não é viável utilizar valores muito altos para k devido ao grande número de blocos e baixa capacidade de localização de erros.

4.4.2 Análise de desempenho para q ou k variado

A variação nos parâmetros q e k afetam o desempenho na criação de CFFs, conforme visto na Seção 4.3, de formas diferentes, onde o mesmo incremento em k faz o tempo de criação da CFF aumentar mais rapidamente do que incrementar q. Para o Algoritmo 4, a CFF gerada é percorrida para gerar os testes (T) conforme as incidências de blocos nas linhas da CFF, portanto, o aumento desses parâmetros pode afetar o desempenho da geração de assinatura mesmo que o tempo de execução da criação da CFF seja desconsiderado. Assim, será avaliado o crescimento de tempo de execução para variação de q e k separadamente, e então os resultados serão comparados com os dados obtidos na Tabela 6.

Nesta seção, o arquivo foi assinado usando chave RSA de módulo 2048 e função SHA256, assim como na Seção 4.4.3.

A Tabela 7 mostra os resultados de desempenho obtidos ao gerar assinaturas com valor de k fixo e variação de q, ou seja, o desempenho varia ao longo do eixo q, considerando a Tabela 6. Os valores diferentes de q com k fixo resultam em t, d, n e comprimento da assinatura final ($|\sigma|$) diferentes para todos os experimentos.

A Tabela 8 mostra os resultados de desempenho obtidos ao gerar assinatura com $q = 8$, ou seja, 64 testes, para diferentes números de blocos (n), a partir da variação do parâmetro k. Não foram apresentados resultados para 1-CFFs de Sperner com n máximo para $t = 64$ devido

q \ k	3	4	5	6	7
5	5	25	125	625	3125
7	7	49	343	2401	16807
8	8	64	512	4096	32768
9	9	81	729	6561	59049
11	11	121	1331	14641	161051
13	13	169	2197	28561	371293
16	16	256	4096	65536	1048576
17	17	289	4913	83521	1419857
19	19	361	6859	130321	2476099
23	23	529	12167	279841	-
25	25	625	15625	390625	-
27	27	729	19683	531441	-
29	29	841	24389	707281	-
31	31	961	29791	923521	-
32	32	1024	32768	-	-
37	37	1369	50653	-	-
41	41	1681	68921	-	-
43	43	1849	79507	-	-
47	47	2209	103823	-	-
49	49	2401	117649	-	-
53	53	2809	148877	-	-
61	61	3721	226981	-	-
64	64	4096	-	-	-

Figura 16 – Compressão de tamanho da assinatura para valores de q e k. Valores para CFFs que geram n inviável foram omitidos. As células foram coloridas para ilustrar a diferença de valores.

k	q	t	n	$\|m\|$	d	t. execução	$\|\sigma\|$
4	7	49	2401	4,688 kB	2	0,2154 s	1,81 kB
	8	64	4096	7,999 kB	2	0,3660 s	2,28 kB
	9	81	6561	12,813 kB	2	0,5569 s	2,81 kB
	11	121	14641	28,595 kB	3	1,5369 s	4,06 kB
	13	169	28561	55,782 kB	4	3,5531 s	5,56 kB

Tabela 7 – Tempo médio de execução (para 100 execuções) para geração de assinatura de blocos de texto de mesmo tamanho e conteúdo com q variado, utilizando chave RSA de módulo 2048.

ao grande n gerado neste caso (equivalente a $\binom{64}{32}$), porém o último experimento utilizou uma 1-CFF de Sperner já que os parâmetros resultam em uma CFF polinomial com $d = 1$. Assim, é avaliada a geração de assinaturas de mesmo tamanho mas criadas a partir de arquivos diferentes, e com capacidade de localização variável.

A análise dos resultados das duas tabelas mostra que o incremento de k pode causar aumentos mais expressivos no tempo de execução em relação ao incremento de q, mesmo para t menor, devido ao rápido crescimento de n, o que segue o padrão de desempenho estabelecido pela Tabela 6.

t	q	k	n	$\|m\|$	d	t. execução	$\|\sigma\|$
64	8	2	64	127 B	8	0,1007 s	2,28 kB
		3	512	1023 B	4	0,1281 s	
		4	4096	8191 B	2	0,3660 s	
		5	32768	63.999 kB	1*	5,1193 s*	

Tabela 8 – Tempo médio de execução (para 100 execuções) para geração de assinatura com 81 testes e blocos de texto iguais com k variado, utilizando chave RSA de módulo 2048.

*- Resultado para 1-CFF gerada via famílias de Sperner.

O fato do comprimento da assinatura ser muito maior que o comprimento pela mensagem é causado pelo tamanho de cada bloco, já que cada bloco tem apenas 1 caractere, para padronizar as análises de todas as seções quando não especificado um valor para s. Porém, para a geração de assinatura, não há restrição de s mínimo. Logo, a proporção entre $|m|$ e $|\sigma|$ é mais vantajosa à medida que s aumenta.

4.4.3 Análise de desempenho para t variado

O número de testes (t) para a assinatura é produto dos valores para q e k, conforme detalhado acima, então o crescimento em tempo de execução para aumento de t mas com n fixo será diferente do crescimento para variação de apenas q ou k, e também permite testar todas as possibilidades de assinatura MTSS para o mesmo arquivo. Foi escolhido um arquivo com $n = 4096$ pois esse valor possui bom número de possibilidades de CFFs polinomiais sem ser um número muito grande, mas também será avaliado o desempenho na criação de uma 1-CFF de Sperner.

A Tabela 9 mostra os resultados obtidos na execução do Algoritmo 4 para o mesmo arquivo de 4096 blocos mas com q e k variado, de forma a testar todas as possibilidades de CFFs polinomiais para o mesmo arquivo, o que gera números de testes (t) diferentes mas consegue manter o n fixo.

n	$\|m\|$	t	q	k	d	t. de execução	$\|\sigma\|$
4096	8191 B	15*	-*	-*	1*	0,3100 s*	0,755 kB
		64	8	4	2	0,3660 s	2,28 kB
		256	16	3	7	0,6663 s	8,28 kB
		4096	64	2	63	3,9154 s	128,28 kB

Tabela 9 – Tempo médio de execução (tempo médio para 100 execuções) para assinatura de arquivo de texto com t variado, assinado usando chave RSA de módulo 2048.

*- Resultado para 1-CFF gerada via famílias de Sperner.

A análise dos resultados mostra que, para mesmo n, o tempo de execução é relacionado ao número de testes gerados para o arquivo e aumenta juntamente com d, já que, para o mesmo

arquivo, só é possível aumentar d ao gerar t maior. Assim como na Seção 4.4.2, a diferença de comprimento entre a mensagem e σ é resultante de ter sido utilizado o valor mínimo para s.

4.4.4 Análise de desempenho para diferentes algoritmos de assinatura e *hash*

O uso de diferentes esquemas de assinatura tradicional (α, como denotados nos Algoritmos 4, 5 e 6) e funções de resumo criptográfico para montar σ pode influenciar o desempenho do algoritmo, considerando que αs funcionam a partir de propriedades diferentes entre si e a escolha de função de *hash* altera o comprimento do conteúdo a ser assinado, além de influenciar o desempenho.

A Tabela 10 mostra os resultados de desempenho obtidos ao gerar assinatura para o mesmo arquivo e CFF usando diferentes funções de resumo criptográfico e αs, além de apresentar o comprimento final da assinatura gerada ($|\sigma|$) para as combinações de algoritmos. Para α = 'RSA', é considerado o esquema PKCS#1 assim como no restante das tabelas, mas o nome foi trocado nesta tabela por questões de tamanho. A coluna $|s_k|$ denota o comprimento da chave em bits, que é fixo para o esquema Ed25519 mas varia de acordo com o módulo da chave para chaves RSA, com o objetivo de avaliar essa variação impacta significativamente no desempenho da geração de assinatura. Para esta tabela, foi assinado um arquivo que gera uma CFF polinomial pequena (em comparação a todas as d-CFFs possíveis), para que o impacto do uso de diferentes algoritmos seja mais visível mas também manter uma boa capacidade de localização relativa ao número de blocos. O arquivo criado para o experimento possui $s = 10000$ para avaliar se o tamanho dos blocos gera diferenças de desempenho para as funções de *hash*, considerando que são utilizadas $n * t + 2$ vezes para gerar a assinatura do MTSS e tendem a levar mais tempo para executar conforme o comprimento do conteúdo de entrada. Os tempos de execução foram apresentados em milissegundos para que a comparação com a Tabela 11 fique mais clara.

n	$\|m\|$	t	q	k	d	α	$\|s_k\|$	função *hash*	t. execução	$\|\sigma\|$
125	11,92MB	25	5	3	2	RSA	1024	SHA256	102,58 ms	960 B
								SHA512	91,64 ms	1792 B
								SHA3-256	96,58 ms	960 B
								SHA3-512	123,70 ms	1792 B
							2048	SHA256	127,00 ms	1088 B
								SHA512	116,86 ms	1920 B
								SHA3-256	136,68 ms	1088 B
								SHA3-512	165,35 ms	1920 B
							4096	SHA256	200,55 ms	1344 B
								SHA512	188,06 ms	2176 B
								SHA3-256	205,50 ms	1344 B
								SHA3-512	227,61 ms	2176 B
						Ed25519	256	SHA512	102,63 ms	1728 B

Tabela 10 – Tempo de execução médio (para 100 execuções) da função de assinatura para esquemas de assinatura clássicos e funções de *hash* diferentes, desconsiderando o tempo para criação da CFF e blocos.

A tabela mostra que há pouca variação de desempenho entre diferentes funções de resumo criptográfico, mas para chaves RSA, a SHA512 tem melhor desempenho, a SHA3-256 demora mais do que a SHA256 para dois dos três comprimentos de chave e a SHA3-512 é a que leva mais tempo para executar, porém também é a função mais segura. Considerando a Tabela 1, o incremento em tempo de execução ao utilizar SHA3-512 é justificado para aplicações que precisem da maior resistência à pré-imagem e segunda pré-imagem. Caso o mais importante seja maximizar a resistência à colisão, a SHA512 é uma boa escolha devido ao desempenho um pouco melhor.

É demonstrado que o módulo da chave RSA utilizado impacta o desempenho de forma significativa, já que o tempo de execução quase dobrou entre o módulo 1024 e 4096, porém o nível de segurança quadriplicou, então para aplicações que necessitem de maior segurança o custo em desempenho é justificado. Para Ed25519, seu desempenho é levemente menor do que o resultado para chave RSA 1024 com SHA256, porém tem nível de segurança similar ao uso de uma chave RSA de módulo 2048, e também executa mais rápido do que o restante de comprimentos de chave para qualquer função de *hash*.

Em relação ao comprimento da assinatura final, o fator mais importante é a função de *hash* utilizada, já que os valores mostrados são para apenas 25 testes na assinatura. Porém, para t menor, a escolha de nível de segurança da chave RSA impacta de forma significativa, e para resumos criptográficos de 512 bits, o Ed25519 apresenta o menor $|\sigma|$.

4.4.5 Variação de desempenho em relação a esquemas tradicionais

O MTSS provê assinaturas que possuem informações extras sobre a integridade de blocos de um documento assinado. Para que isso seja possível, as assinaturas do esquema são naturalmente maiores e a verificação com localização e/ou correção é mais custosa. Aqui o MTSS é comparado com esquemas de assinatura tradicionais para ser possível quantificar esse custo. A Tabela 11 mostra os resultados obtidos ao usar a script próprio para gerar assinaturas tradicionais para o mesmo arquivo e com os mesmos algoritmos utilizados na Tabela 10, e compará-las em tamanho e tempo necessário para a execução. Devido à escolha de arquivo e CFF feita anteriormente, o custo de usar o MTSS é minimizado mas ainda não é tão pequeno a ponto de não gerar diferenças significativas de desempenho, como o que aconteceria com o uso de CFFs muito pequenas em dimensão.

A Tabela 11 mostra o tempo de execução e comprimento total da assinatura gerada ($|\sigma|$) ao assinar o mesmo arquivo da Tabela 10 usando os esquemas de assinatura clássicos PCKS#1 v1.5 e Ed25519 em Python, e as colunas Δt. e $\Delta|\sigma|$ fazem a comparação entre os tempos de execução e $|\sigma|$ obtidos nesta tabela e na Tabela 10, onde o valor obtido mostra quantos milissegundos/armazenamento em bytes a mais são necessários para assinar o mesmo arquivo usando o MTSS.

Os resultados mostram que, para CFFs menores e mesma biblioteca criptográfica utilizada, não há uma diferença de desempenho tão grande entre os esquemas tradicionais e o MTSS.

α	$\lvert s_k \rvert$	função *hash*	t. execução	Δ t.	$\lvert\sigma\rvert$	$\Delta\lvert\sigma\rvert$
PKCS#1 v1.5 (Tradicional)	1024	SHA256	74,87 ms	27,71 ms	128 B	832 B
		SHA512	70,61 ms	21,03 ms		1664 B
		SHA3-256	71,64 ms	24,94 ms		832 B
		SHA3-512	74,93 ms	48,77 ms		1664 B
	2048	SHA256	90,04 ms	36,96 ms	256 B	832 B
		SHA512	88,32 ms	28,54 ms		1664 B
		SHA3-256	88,70 ms	47,98 ms		832 B
		SHA3-512	94,68 ms	70,67 ms		1664 B
	4096	SHA256	143,42 ms	57,13 ms	512 B	832 B
		SHA512	143,64 ms	44,42 ms		1664 B
		SHA3-256	146,50 ms	59,00 ms		832 B
		SHA3-512	148,41 ms	79,20 ms		1664 B
Ed25519 (Tradicional)	256	SHA512	65,74 ms	36,89 ms	64 B	1664 B

Tabela 11 – Tempo de execução médio (para 100 execuções) da função de assinatura para chaves RSA PKCS#1 de módulos diferentes e chave Ed25519, para o mesmo arquivo da Tabela 10, e diferença de tempo de execução e comprimento da assinatura gerada entre as duas tabelas.

A variação de tempo de execução entre funções de *hash* diferentes é maior para a SHA3-512. Para chaves RSA, a diferença de desempenho aumenta levemente à medida que o comprimento da chave se torna maior. Porém, como mostram o restante das tabelas da seção, a parcela de tempo dedicada à montar os testes da assinatura do MTSS é maior à medida que as dimensões da CFF gerada aumentam, seja para q ou k, e nesses casos a diferença de desempenho é superior.

Como o comprimento da assinatura do MTSS em bytes segue a relação $((t+1)*|h|+|\sigma|)/8$, os resultados para $\Delta|\sigma|$ correspondem ao valor de $((t+1)*|h|)/8$, o que mostra que a variação de comprimento da assinatura aumenta para t e $|h|$ maiores, mas não depende do comprimento da assinatura gerada por α.

4.4.6 Parâmetros para máximo aproveitamento de tamanho da assinatura

Caso seja estipulado um tamanho máximo para a assinatura do MTSS, o número de testes máximo para tal depende dos algoritmos de assinatura e resumo criptográfico utilizados, enquanto que o número mínimo de testes depende principalmente do número de blocos que serão assinados (como demonstrado pela Tabela 2), além da compressão (dependente de k) desejada para a assinatura, conforme a Figura 16. Considera-se 'máximo aproveitamento de tamanho' como utilizar o máximo possível do tamanho fornecido para se preencher com testes, para prever quantos testes serão gerados para certos tamanhos de assinatura e assim determinar qual o d máximo gerável para o comprimento da assinatura e valor viável para k ($2 < k < 8$). Essa relação é exemplificada pela Tabela 12, onde o número máximo de testes e número mínimo de blocos foi estipulado para diferentes comprimentos máximos da assinatura final gerada pelo Algoritmo 4, com variação gerada pelo uso de algoritmos de assinatura tradicional e de resumo

criptográfico com comprimentos diferentes de saída.

$\|\sigma\|$ máx.	$\|\sigma'\|$	$\|h\|$	t teórico	q^2 mais próx.	q	n mín. (k = 3)	d máx.
100kB	512	256	3198	2809	53	148877	26
	2048		3192	2809	53	148877	26
	4096		3184	2809	53	148877	26
	512	512	1599	1369	37	50653	18
	2048		1596	1369	37	50653	18
	4096		1592	1369	37	50653	18
10kB	512	256	318	289	17	4913	8
	2048		312	289	17	4913	8
	4096		304	289	17	4913	8
	512	512	159	121	11	1331	5
	2048		156	121	11	1331	5
	4096		152	121	11	1331	5
1kB (2^{10}B)	512	256	30	25	5	125	2
	2048		24	16	4	64	1
	4096		16	16	4	64	1
	512	512	15	9	3	27	1
	2048		12	9	3	27	1
	4096		8	4	2	8	0

Tabela 12 – Número máximo de testes e número mínimo de blocos para diferentes tamanhos máximos da assinatura final (σ).

$|\sigma|$ denota o comprimento total da assinatura final e $|h|$ o comprimento da saída da função de *hash* escolhida, enquanto que $|\sigma'|$ corresponde ao comprimento da saída da assinatura gerada pelo esquema tradicional de assinatura utilizado. A coluna 't teórico' representa o maior valor possível para t considerando a relação $t = (|\sigma| - |\sigma'|)/|h|$, porém, não leva em consideração que t precisa ser uma potência de primo elevada a 2, e portanto os cálculos de n mínimo e d máximo consideram a maior potência de primo q^2 que seja igual ou menor que o t teórico. Como é demonstrado pela tabela, o fator que mais impacta o (t) gerado para uma assinatura de tamanho pré-determinado é a função de *hash* utilizada para resumir os testes. Também é demonstrado que $|\sigma|'$ só se torna relevante na variação de t à medida que $|\sigma|$ diminui.

4.5 VERIFICAÇÃO DE ASSINATURA SEM MODIFICAÇÕES

Para uma mensagem sem modificações, o funcionamento do algoritmo de verificação é similar ao de um esquema tradicional, com a adição de comparar o *hash* da mensagem da assinatura com o *hash* reconstruído a partir da mensagem a ser verificada. Nesta seção não serão analisadas de forma específica a variação de desempenho para variações nos parâmetros t, n ou tipo de arquivo, pois, para a verificação sem modificações na mensagem, a única diferença na execução que eles geram é a variação de desempenho da obtenção de blocos a partir do arquivo.

4.5.1 Análise de desempenho para diferentes algoritmos de assinatura e *hash*

A Tabela 13 mostra os resultados obtidos na execução da função de verificação, considerando a assinatura gerada para o arquivo assinado na Tabela 10 e usando o mesmo arquivo como entrada, ou seja, não há modificações na mensagem. A tabela mostra o comprimento da assinatura gerada pelo MTSS ($|\sigma|$) para as combinações de algoritmos, conforme é visto em detalhes na Tabela 10, e faz comparações de desempenho ao utilizar diferentes funções de resumo criptográfico e esquemas de assinatura tradicional (α) para realizar a verificação das assinaturas geradas por eles.

α	$\lvert s_k \rvert$	função *hash*	$\lvert\sigma\rvert$	t. execução
PKCS#1 v1.5	1024	SHA256	960 B	57,385 ms
		SHA512	1792 B	61,103 ms
		SHA3-256	960 B	65,263 ms
		SHA3-512	1792 B	64,039 ms
	2048	SHA256	1088 B	57,026 ms
		SHA512	1920 B	56,303 ms
		SHA3-256	1088 B	56,111 ms
		SHA3-512	1920B	59,473 ms
	4096	SHA256	1344 B	57,929 ms
		SHA512	2176 B	55,171 ms
		SHA3-256	1344B	58,795 ms
		SHA3-512	2176B	63,441 ms
Ed25519	256	SHA512	1728 B	67,403 ms

Tabela 13 – Tempo de execução médio (para 100 execuções) da função de verificação para mesmo arquivo da tabela 10, a partir da assinatura com diferentes esquemas de assinatura clássicos e funções de *hash*.

Os resultados mostram que, para a verificação de assinatura sem modificações, a escolha de algoritmos impacta menos no desempenho do que na geração de assinatura, via dados da Tabela 10. É possível que as variação dos valores seja causada por alterações comuns na execução da aplicação e não devido às escolhas realizadas.

4.5.2 Variação de desempenho em relação a esquemas tradicionais

A Tabela 14 compara o tempo de execução da verificação de assinatura do MTSS com o de esquemas tradicionais de assinatura (α), executados a partir de script próprio em Python. Para a verificação de uma assinatura tolerante a modificações, é verificado se σ' é uma assinatura válida para o T gerado pelo Algoritmo 4, enquanto que, para α, a verificação é feita para a mensagem assinada como um todo. A última coluna mostra a variação de tempo de execução (Δ t.) entre as implementações de verificação do MTSS e tradicional. Também é mostrada a diferença entre o comprimento das assinaturas geradas ($\Delta|\sigma|$), conforme é visto em detalhes na Tabela 10.

α	$\lvert s_k \rvert$	função *hash*	$\lvert\sigma\rvert$	$\Delta\lvert\sigma\rvert$	t. execução	Δ t.
PKCS#1 v1.5 (Tradicional)	1024	SHA256	128 B	832B	53,687 ms	3,698 ms
		SHA512		1664 B	53,804 ms	7,299 ms
		SHA3-256		832B	55,093 ms	10,170 ms
		SHA3-512		1664 B	60,511 ms	3,528 ms
	2048	SHA256	256 B	832B	56,231 ms	0,795 ms
		SHA512		1664 B	55,525 ms	0,778 ms
		SHA3-256		832B	55,474 ms	0,637 ms
		SHA3-512		1664 B	58,430 ms	1,043 ms
	4096	SHA256	512 B	832B	56,009 ms	1,920 ms
		SHA512		1664 B	54,508 ms	0,663 ms
		SHA3-256		832B	57,838 ms	0,957 ms
		SHA3-512		1664 B	59,612 ms	3,829 ms
Ed25519 (Tradicional)	256	SHA512	64 B	1664B	55,342 ms	12,06 ms

Tabela 14 – Tempo de execução médio (para 100 execuções) da função de verificação de assinatura tradicional, para chaves RSA PKCS#1 de módulos diferentes e chave Ed25519, para o mesmo arquivo da Tabela 13, e diferença de tempo de execução entre as duas tabelas.

Os resultados mostram que, para verificação de mensagem sem modificações, há pouca diferença de desempenho entre os algoritmos, sendo que a maior delas é para o α Ed25519. Para assinaturas criadas com chaves RSA, as maiores diferenças de desempenho estão para verificações feitas com chaves de 1024 bits de comprimento, mas a variação de tempo de execução não parece seguir um padrão baseado no comprimento da chave ou na função de *hash* utilizada. Porém, como já estabelecido pela Tabela 11, a maior variação de comprimento da assinatura final é causado pelo comprimento dos resumos criptográficos gerados e pelo número de testes gerados pelo MTSS.

4.6 LOCALIZAÇÃO DE MODIFICAÇÕES

Para a localização de modificações, é interessante analisar como a variação de parâmetros afeta a capacidade ou tempo de localização.

A seção inicia avaliando o efeito dos parâmetros q e k na proporção de blocos modificáveis gerados, sendo que o número absoluto de (d) para os mesmos parâmetros já foi apresentado na Tabela 9. Em seguida, são feitas análises de desempenho para variações dos parâmetros t, q e k, assim como na Seção 2, para utilizar as assinaturas geradas naquela seção para realizar a localização e comparar como os mesmos parâmetros afetam as duas funções.

Para esta seção, é verificada ainda se existe diferença de desempenho para números diferentes de modificações para o mesmo arquivo, e também se o comprimento dos blocos da mensagem altera o tempo de execução da localização.

4.6.1 Parâmetros para proporção de blocos modificáveis

A Figura 17 ilustra a porcentagem de blocos modificáveis obtidos para determinados parâmetros q e k de uma d-CFF(q^2,q^k), formatados em notação científica para facilitar a leitura da diferença de valores. As colunas $q < 5$, $k < 3$ e $k > 7$ foram omitidas conforme análise de viabilidade dos parâmetros realizada na Seção 4.1.1. As células da tabela foram coloridas para demonstrar visualmente a variação nos valores. Ao analisar esta tabela com a tabela da Figura 9, é possível observar que, mesmo que o uso de um q maior maximize o número total de modificações possíveis, a melhor proporção de blocos modificáveis é encontrada nos menores valores para q. Porém, ambas demonstram que é ideal minimizar k para otimizar a localização de modificações, tanto para obter mais blocos modificáveis quanto para aumentar a taxa de modificações possível da mensagem.

q \ k	3	4	5	6	7
5	1,60E+00	1,60E-01	3,20E-02	0,00E+00	0,00E+00
7	8,75E-01	8,33E-02	5,95E-03	8,50E-04	1,21E-04
8	5,86E-01	4,88E-02	3,05E-03	3,81E-04	4,77E-05
9	5,49E-01	3,05E-02	3,39E-03	1,88E-04	2,09E-05
11	3,76E-01	2,05E-02	1,24E-03	1,13E-04	5,13E-06
13	2,73E-01	1,40E-02	8,08E-04	4,14E-05	3,19E-06
16	1,71E-01	7,63E-03	2,86E-04	1,79E-05	7,45E-07
17	1,63E-01	5,99E-03	2,82E-04	1,24E-05	4,87E-07
19	1,31E-01	4,60E-03	1,62E-04	6,38E-06	3,36E-07
23	9,04E-02	2,50E-03	7,77E-05	2,70E-06	8,81E-08
25	7,68E-02	2,05E-03	6,14E-05	1,64E-06	6,55E-08
27	6,60E-02	1,51E-03	4,18E-05	1,29E-06	3,82E-08
29	5,74E-02	1,27E-03	3,41E-05	8,41E-07	2,32E-08
31	5,04E-02	1,08E-03	2,45E-05	6,76E-07	1,82E-08
32	4,58E-02	9,54E-04	2,09E-05	5,59E-07	1,46E-08
37	3,55E-02	6,40E-04	1,30E-05	2,73E-07	6,32E-09
41	2,90E-02	4,60E-04	8,63E-06	1,68E-07	3,08E-09
43	2,64E-02	4,10E-04	6,80E-06	1,27E-07	2,58E-09
47	2,22E-02	3,07E-04	4,80E-06	8,35E-08	1,38E-09
49	2,04E-02	2,78E-04	4,25E-06	6,50E-08	1,18E-09
53	1,75E-02	2,15E-04	3,11E-06	4,51E-08	6,81E-10
61	1,32E-02	1,44E-04	1,78E-06	2,33E-08	3,18E-10
64	1,18E-02	1,25E-04	1,40E-06	1,75E-08	2,27E-10

Figura 17 – Porcentagem de blocos modificáveis para parâmetros $q > 5$ e $k > 3$ seguindo a relação $d = \lfloor (q-1)/(k-1) \rfloor$ e $n = q^k$ para d-CFFs. As porcentagens foram formatadas com notação científica para facilitar a leitura. As células da tabela foram coloridas para ilustrar a variação de valores.

4.6.2 Análise de desempenho para *t* variado

O número de testes (t) na assinatura a ser verificada é importante no desempenho do Algoritmo 5 pois, após a CFF ser reconstruída a partir da mensagem de entrada, todos os testes obtidos pela assinatura (T) e os testes reconstruídos (T') são percorridos para encontrar os índices dos elementos de T' que se mantiveram iguais aos testes correspondentes em T, de forma a encontrar as modificações. Além disso, o aumento de t também é produto do aumento de q ou k (que será avaliado por si mesmo na próxima seção), e essa variação de parâmetros afeta o desempenho da construção de T'.

A Tabela 15 mostra os resultados obtidos na localização de 1 modificação (denotada por $|I|$, ou seja, o número de índices de blocos modificados via o Algoritmo 5) para as assinaturas geradas na Tabela 9, utilizando em todas as análises um arquivo de texto similar ao assinado, mantendo o tamanho e número de blocos, mas com o mesmo caractere modificado.

n	$\|m\|$	$\|I\|$	t	k	$\|\sigma\|$	d	t. de execução
4096	8191 B	1	15*	-*	736 B*	1*	0,1937 s*
			64	4	2,25 kB	2	0,2151 s
			256	3	8,25 kB	7	0,3379 s
			4096	2	128,25 kB	63	3,0888 s

Tabela 15 – Tempo médio de execução (para 100 execuções) da localização de 1 modificação no mesmo arquivo a partir das assinaturas geradas na Tabela 9.

*- Resultado para 1-CFF gerada via famílias de Sperner.

Os resultados mostram que o tempo de execução parece estar diretamente relacionado ao número de testes da assinatura, o que é esperado considerando o funcionamento do algoritmo, e é intensificado pela implementação sem multiprocesso da avaliação de T'. Dessa forma, para melhorar o desempenho dessa funcionalidade, o ideal é não criar assinaturas com t muito alto em relação a n.

4.6.3 Análise de desempenho para *q* ou *k* variado

Diferentes valores para q ou k impactam na formação de T' a partir da CFF reconstruída para a localização, e de forma relacionada, o aumento desses parâmetros também aumenta o t gerado, como visto anteriormente. Assim, será avaliado como esses parâmetros afetam a função de localização de modificações em relação à função de assinatura, como visto na Seção 4.4.2, e em relação à construção da CFF em si, como visto na Tabela 6.

A Tabela 16 mostra os resultados obtidos na localização de 1 modificação para as assinaturas geradas na Tabela 7, utilizando arquivos de textos similares aos assinados anteriormente, mantendo o tamanho e número de blocos, mas com o mesmo caractere modificado.

A Tabela 17 mostra os resultados obtidos na localização de 1 modificação para as assinaturas geradas na Tabela 8, utilizando mesma metodologia da Tabela 16.

k	$\|I\|$	q	t	n	$\|m\|$	d	$\|\sigma\|$	t. execução
4	1	7	49	2401	4,688 kB	2	1,81 kB	0,1139 s
		8	64	4096	7,999 kB	2	2,28 kB	0,1984 s
		9	81	6561	12,813 kB	2	2,81 kB	0,3487 s
		11	121	14641	28,595 kB	3	4,06 kB	1,2432 s
		13	169	28561	55,782 kB	4	5,56 kB	3,7925 s

Tabela 16 – Tempo médio de execução (para 100 execuções) de localização de 1 modificação para os arquivos e assinaturas geradas na Tabela 7, que variam em tamanho conforme variação do parâmetro q.

q	t	$\|\sigma\|$	$\|I\|$	k	n	$\|m\|$	d	t. execução
8	64	2,28 kB	1	2	64	127 B	8	0,0581 s
				3	512	1023 B	4	0,0668 s
				4	4096	8191 B	2	0,1984 s
				5*	32768*	63.999 kB*	1*	7,5582 s*

Tabela 17 – Tempo médio de execução (para 100 execuções) para localização de 1 modificação para as assinaturas geradas na Tabela 8, que possuem mesmo tamanho mas são válidas para n variado de acordo com a variação de k.

*- Resultado para 1-CFF gerada via famílias de Sperner.

Para a Tabela 17, assim como na Seção 4.6.2, o número de testes gerado é diretamente ligado ao desempenho, sendo que o incremento em tempo de execução aumenta mais rápido de um experimento para outro à medida que q aumenta. Já na Tabela 16, t se mantém, então o acréscimo de desempenho vem do grande número de blocos que precisa ser avaliado para reconstruir os testes. O incremento em tempo de execução se mantém para o aumento de k, porém a diferença de desempenho entre os experimentos à medida que k aumenta é maior do que para q. De forma geral, o padrão das Tabelas 7 e 8 se mantém.

4.6.4 Análise de desempenho para $|I|$ variado

O valor d de uma CFF indica quantos blocos podem ser modificados de forma a ser possível localizar as modificações, mas uma mensagem pode não ter alterações o suficiente para usar 100% da capacidade de localização oferecida para sua assinatura. Na execução prática do algoritmo, é possível que valores diferentes para $|I|$ alterem seu desempenho. Além disso, é importante analisar se o excesso de modificações na mensagem também pode causar alterações no tempo de execução.

A Tabela 18 apresenta os resultados obtidos ao tentar localizar diferentes números de blocos modificados para o arquivo assinado na Tabela 9, linha 3, que gera uma 7-CFF(256, 4096) com $q = 16$ e $k = 3$. Foi incluído o desempenho para $|I| = 0$ para obter a diferença em tempo de execução para verificação com ou sem localização, considerando que a mensagem e assinatura utilizadas nesta seção são diferentes das utilizadas na Seção 4.5. Para a linha $|I| = 10$, houveram mais modificações que o permitido pela CFF e portanto a localização teve resultado

inválido, o que significa que o resultado final da execução retornou não só os índices dos blocos verdadeiramente modificados mas também falsos negativos.

n	$\|m\|$	$\|\sigma\|$	t	d	$\|I\|$	t. de execução
4096	8191 B	8,28 kB	256	7	0	0,0502 s
					1	0,3379 s
					4	0,3181 s
					7	0,3319 s
					10	0,3356 s

Tabela 18 – Tempo médio de execução (para 100 execuções) da localização de número variado de blocos modificados ($|I|$) para o mesmo arquivo.

Os resultados mostram que, para $d > 0$, não há diferença de desempenho significativa para diferentes números de blocos modificados, mesmo quando há mais blocos modificados do que o permitido pela assinatura. A diferença entre $d = 0$ e $d = 1$ se dá pelo fato de não ser necessário reconstruir os testes da assinatura quando não há modificações na mensagem.

4.7 CORREÇÃO DE MODIFICAÇÕES

Para a correção, é interessante analisar quanto o tamanho dos blocos, número de modificações, conteúdo a ser corrigido ou função de *hash* utilizada influencia no tempo de execução do Algoritmo 6. Também é importante saber se, para parâmetros viáveis na prática, é possível que aconteçam colisões nos resultados da correção.

Nesta seção, os resultados de desempenho obtidos na Seção 4.6 serão diminuídos dos tempos de execução totais quando corresponderem aos experimentos executados, para que a execução prévia da localização não interfira nos resultados exclusivos para a correção.

4.7.1 Análise de desempenho para $|I|$ variado

A correção dos blocos é apresentada, no Algoritmo 6, de forma sequencial, onde é necessário concluir a correção de um bloco para iniciar a mesma operação para o próximo bloco. A implementação do algoritmo manteve o acesso sequencial, pois, conforme será detalhado na Seção 4.7.3, já foi utilizada computação paralela para encontrar o valor original do bloco e, para valor viável para k ($k > 2$), o número de modificações não é tão alto quanto o número de operações necessárias para realizar a correção.

A Tabela 19 apresenta os resultados obtidos na correção, a partir da mesma assinatura, de um documento com números diferentes de blocos modificados de mesmo conteúdo no arquivo original. Os resultados de desempenho de localização da Tabela 18 foram diminuídos dos tempos de execução totais para os valores de $|I|$ correspondentes, já que a execução da correção inclui o tempo necessário para concluir a localização.

n	$\|m\|$	$\|\sigma\|$	t	d	s	$\|I\|$	t. de execução
4096	8191 B	8,28 kB	256	7	1	1	0,0242 s
						4	0,1817 s
						7	0,2821 s

Tabela 19 – Tempo médio de execução (para 100 execuções) de correção de número variado de blocos modificados ($|I|$) de 1 caracter de comprimento para o mesmo arquivo.

Os resultados mostram que, de forma geral, o tempo de execução aumenta de acordo com o número de blocos modificados, com proporção de crescimento similar à proporção do incremento de $|I|$ entre os experimentos.

4.7.2 Análise de desempenho para diferentes caracteres

Considerando o uso prático de codificações de caracteres (neste trabalho, a UTF-8), cada caracter possui valor binário diferente. Como a correção por força bruta procura todos os valores binários possíveis para encontrar o conteúdo original da mensagem, a ordem dessa busca influencia no tempo de execução do algoritmo. Por exemplo, se a busca for feita sequencialmente do menor ao maior valor binário, caracteres que são codificados para valores menores serão encontrados primeiro e vice-versa.

Neste capítulo, até então, os blocos de texto assinados eram todos compostos apenas pelo caractere '!' (U+0021), que, devido à sua posição na lista de codificações, possui um menor valor binário em relação a quase todos os outros caracteres de texto e portanto leva menos tempo para ser corrigido. O conteúdo dos blocos a serem corrigidos não seria relevante caso a aplicação desenvolvida testasse todos os valores para tentar encontrar colisões, assim como no Algoritmo 6, mas conforme descrito na Seção 4.2, a correção para no primeiro resultado válido encontrado, e os resultados de desempenho desta seção buscam justificar essa escolha.

Para os experimentos desta seção, será analisada a correção de três documentos de texto de mesmo tamanho, n e $s = 3$ para todos os blocos. Os blocos do primeiro arquivo serão compostos apenas por exclamações (U+0021), os do segundo apenas por tils (U+007E), e os do terceiro arquivo serão compostos por caracteres com comprimento de 3 bytes (U+FFFD), que não são previstos para correção pela aplicação e portanto não são corrigíveis, já que ela calcula a duração do procedimento de correção considerando apenas caracteres de 1 byte de comprimento. Esse experimento será feito para executar todas as possibilidades possíveis para caracteres de 1 byte, da mesma forma que aconteceria caso a aplicação avaliasse todos os valores para encontrar colisões ao invés de parar de corrigir ao encontrar o primeiro resultado.

A Tabela 20 apresenta os resultados da correção de um único bloco, com mesma localização no documento em todos os experimentos, mas para caracteres diferentes na mensagem original. Os tempos de localização serão considerados os mesmos da Tabela 21 para $s = 3$, já que em ambas as tabelas apenas 1 bloco foi modificado. A coluna 'char' especifica o caractere que compõe os blocos, para os três caracteres descritos no parágrafo anterior, sendo que o

caracter especial da terceira linha não pôde ser inserido diretamente por texto.

n	$\|m\|$	t	d	$\|\sigma\|$	s	$\|I\|$	char	t. de execução
4096	16kB	256	7	8,28 kB	3	1	!	2 min 33 s
							~	9 min 40 s*
							U+FFFD	19 min 32 s**

Tabela 20 – Tempo médio de execução (para 50 execuções) para correção de arquivos de mesmo tamanho, n e $|\sigma|$, mas com caracteres diferentes preenchendo o conteúdo dos blocos.

*- Resultado para 10 execuções.
*- Resultado para 1 execução.

Os resultados mostram que a perda de desempenho para realizar a correção buscando colisões é muito alta para justificar essa escolha, e é mais eficiente executar o algoritmo como se elas não fossem uma possibilidade. A verificação da existência prática de colisão para parâmetros viáveis do algoritmo de correção será feita na Seção 4.7.5.

4.7.3 Análise de desempenho para *s* variado

O comprimento original dos blocos modificados é o fator mais importante no desempenho da execução do Algoritmo 6. Isso acontece porque a correção é feita pela testagem de todas as possibilidades de valores para o *hash* do teste que contenha apenas o bloco modificado sendo corrigido. Devido à propriedade de resistência à pré-imagem das funções de resumo criptográfico, é necessário gerar todos os valores possíveis para o conteúdo original e então resumí-lo antes de comparar com o *hash* do teste na assinatura. Considerando caracteres de 1 byte, para encontrar todos os valores binários para um bloco de $s = 1$ são necessárias 2^8 operações. Já se $s = 5$, por exemplo, são necessárias 2^{40} operações. Dessa forma, o máximo de operações necessárias para tal é definido pela relação 2^{8*s}, onde s é o tamanho máximo em bytes dos blocos.

É possível diminuir o crescimento assintótico da correção ao utilizar s de comprimento menor em bits. Porém, para documentos de texto, isso é difícil na prática. Como discutido na Seção 4.1.2, não é viável fixar comprimentos de blocos com antecedência, então não se pode criar blocos dividindo o conteúdo de 1 caracter em mais de um bloco. Outro fator é a compatibilidade de codificações para diferentes aplicações. Como apresentado na Seção 4.1.2.1, o UTF-8 tem uso tão prevalente na Web que usar apenas codificações menores, como a ASCII, tornaria o uso do MTSS muito limitado.

Dessa forma, optou-se por permanecer utilizando arquivos codificados em UTF-8, mas os resultados apresentados consideram apenas o uso dos seus primeiros 128 caracteres, equivalentes aos caracteres ASCII, pois precisam de apenas 1 byte para serem representados, o que torna $s = 1$ equivalente a 8 bits. Isso significa que, para diminuir a complexidade da correção,

não é recomendado utilizar o restante dos caracteres UTF-8 para não aumentar o comprimento de s, o que impede a correção de textos em português gramaticalmente correto, por exemplo.

Da forma que o UTF-8 distribui caracteres para valores binários, os primeiros 128 valores (U+0000 a U+007F) são codificados para corresponder um a um aos caracteres ASCII, mas o restante dos valores para 8 bits (em hexadecimal, `0x80` a `0xFF`) não são codificados para qualquer caractere. Assim, é possível criar codificações próprias a partir do UTF-8 para diferentes aplicações, de forma a preencher o restante dos valores e não ser necessário utilizar mais de 1 byte por caractere.

A Tabela 21 mostra os resultados obtidos para a correção da mesma linha de um arquivo de texto, mas para comprimento variado dos blocos na mensagem original, sendo que todos os caracteres corrigidos são exclamações (U+0021) para minimizar o tempo de execução. Os arquivos assinados são similares ao utilizado pela Seção 4.7.1, mas todos os blocos foram alterados, antes da assinatura, de forma a repetir caracteres e assim incrementar s de forma homogênea. O bloco modificado é o mesmo (em localização no documento) para todos os experimentos. Os tempos médios de execução (para 100 execuções) para a localização dos blocos são mostrados na coluna 't. verif.' para que os valores obtidos sejam diminuídos do resultado final na última coluna, que mostra o desempenho da correção como um todo incluindo a localização dos blocos modificados. Essa coluna foi incluida porque os arquivos utilizado nestes experimentos não foram utilizados na Seção 4.6, com exceção do primeiro.

n	t	$\lvert\sigma\rvert$	d	$\lvert I\rvert$	char	$\lvert m\rvert$	s	t. verif.	t. de execução
4096	256	8,28 kB	7	1	! (U+0021)	8 kB	1	0,3379 s	0,0242 s
						12 kB	2	0,3260 s	0,6180 s
						16 kB	3	0,3176 s	2 min 33 s*
						20 kB	4	0,3228 s	10 h 24 min**
						24 kB	5	0,3211 s	107,5 d***

Tabela 21 – Tempo médio de execução (para 100 execuções) de correção do mesmo bloco mas com comprimento em caracteres variável.

*- Resultado para 50 execuções.
**- Resultado para 1 execução.
***- Valor estimado com base na progressão da execução após 14,5 horas.

Os resultados mostram que há incremento drástico de tempo de execução para o aumento de s, mesmo com a utilização de computação paralela na implementação. Considerando que esse são os resultados para o melhor caso, onde todos os caracteres a serem corrigidos são U+0021, é perceptível a inviabilidade de correção de blocos com $s > 3$. O tempo de execução total para $s = 5$ foi estimado a partir dos registros temporais da execução, que executou 800 milhões de operações após 14h e 33 minutos, sendo que para completar a correção para 5 caracteres U+0021 são necessárias 215.590.515.250 operações. De forma secundária, a tabela também mostra que valores pequenos para s não influenciam o tempo de localização para o mesmo número de modificações, mesmo que a correção aumente exponencialmente.

Como estes são os resultados para documentos de texto, que possuem número de caracteres por bloco mais flexível, também é demonstrado que a correção de documentos XML para a implementação desenvolvida não é viável, pois para esse tipo de documento é difícil atingir comprimento tão curto para *tags* considerando o uso comum, mesmo que todas elas sejam postas em blocos individualmente.

É importante ressaltar que a correção foi feita de forma genérica, onde se testam todos os caracteres possíveis sem delimitar o formato ou semântica possíveis das modificações, e assim a correção pode ser otimizada para aplicações específicas. Além disso, caso o conteúdo original do documento seja imprescindível de ser recuperado, o custo da correção pode se justificar. Além disso, mesmo para os experimentos mais demorados, a informação de quais blocos foram modificados fica disponível de forma muito rápida.

4.7.4 Análise de desempenho para funções de *hash* variadas

Para encontrar o resumo criptográfico original para determinado bloco, o valor binário testado é resumido, inserido na concatenação de *hashes* para o teste sendo avaliado, e então essa concatenação também é resumida antes de determinar se o valor correto foi encontrado. Dessa forma, para cada tentativa de correção, a função de *hash* escolhida é executada duas vezes. Considerando que a correção pode levar até 2^{8s} tentativas no pior caso teórico, as diferenças de desempenho dentre as funções de resumo criptográfico pode ter impactos significativos no tempo de execução do algoritmo.

A Tabela 22 mostra os resultados obtidos na correção do mesmo arquivo e bloco com $s = 3$ em todos os experimentos, mas a partir de assinaturas resumidas com funções de resumo criptográfico diferentes. O arquivo utilizado foi o mesmo do segundo experimento (segunda linha da tabela) da Seção 4.7.2, com blocos compostos por U+007E com $s = 3$, para que a correção leve mais operações para ser concluída. Os tempos médios de execução (para 100 execuções) para a localização do bloco modificado são mostrados na coluna 't. verif.' para que os valores obtidos sejam diminuídos do resultado final na última coluna.

n	t	d	s	$\|I\|$	char	$\|\sigma\|$	função *hash*	t. verif.	t. de execução
4096	256	7	3	1	~ (U+007E)	8,28 kB	SHA256	0,339 s	9 min 40 s
						16,31 kB	SHA512	0,357 s	12 min 20 s**
						8,28 kB	SHA3-256	0,351 s	9 min 23 s**
						16,31 kB	SHA3-512	0,362 s	11 min 50 s**

Tabela 22 – Tempo médio de execução (para 10 execuções) de correção do mesmo arquivo e bloco, porém com utlização de diferentes funções de *hash* na sua assinatura.

**- Resultado para 3 execuções, que será corrigido na versão final do trabalho.

Os resultados mostram que o comprimento dos resumos gerados ($|h|$) parece ser o maior fator de desempenho da escolha de função na correção, já que os pares SHA256/SHA3-256 e SHA512/SHA3-512 possuem desempenho similar, porém o aumento de tempo de exe-

cução não é proporcional à diferença de $|h|$. Esse resultado difere do obtido para a geração de assinaturas na Tabela 10, mas aqui o impacto é muito mais significativo. De forma secundária, a tabela também mostra que o tempo de localização não varia para diferentes funções de *hash* quando há modificações, e, assim como na Tabela 21, que a diferença entre o tempo de execução necessário para localizar e corrigir modificações torna o desempenho da localização pouco significativo para $s > 2$.

4.7.5 Computação de colisão

Conforme discutido na Seção 4.2 e 4.7.2, a aplicação foi desenvolvida para concluir a correção ao encontrar o primeiro resultado, ou seja, não avalia a existência de colisões como é feito no Algoritmo 6. Porém, é possível calcular a possibilidade de colisão com antecedência para verificar se checá-la durante a correção é realmente necessário.

Para as funções de *hash* utilizadas na implementação deste trabalho, a Tabela 1 mostra que as funções SHA256 e SHA3-256 têm a menor resistência à colisão (nível de segurança de 128 bits), resultante do comprimento de sua saída. Contudo, considerando os resultados da Seção 4.7.3, o uso de $s > 3$ é inviável para a aplicação desenvolvida, e o comprimento em bits para $s = 3$ (24 bits) não é o suficiente para garantir a existência de alguma colisão. Logo, há possibilidade teórica mas a ocorrência é improvável na prática. A probabilidade de colisão teórica já é conhecida e extremamente baixa (IDALINO; MOURA; ADAMS, 2019, Proposição 2), porém a verificação prática foi realizada para até 20 bits de comprimento de bloco e utilizando apenas SHA256.

Para confirmar os resultados, foi computada a existência de colisão para o resumo de todos os valores binários com 24 bits de comprimento, o que leva à criação de 2^{8*s} *hashes*, utilizando o script `test_collision.py` para SHA256, SHA512, SHA2-256 e SHA3-512. Os resultados da execução mostram que, para blocos de até 24 bits de comprimento, não há colisão para quaisquer funções testadas. Dessa forma, caso implementações futuras consigam aumentar o s viável, esse resultado precisará ser reavaliado.

5 CONCLUSÃO

Este trabalho conclui que o MTSS funciona na prática para a prova de conceito desenvolvida. O esquema não é tão eficiente quanto esquemas tradicionais, porém ele permite localizar e até corrigir modificações ao invés de apenas descartar o documento caso ele tenha sido alterado, e a aplicação desenvolvida foi a primeira que permite mensurar na prática a escolha entre eficiência (de desempenho ou armazenamento) e capacidade de identificar e corrigir modificações. Contudo, devido às estruturas utilizadas para implementá-lo, suas vantagens vêm junto de certas restrições nos valores de entrada possíveis.

Devido à necessidade de gerar a mesma CFF para a geração e verificação de assinatura, ou seja, o número de blocos (n) precisa ser o mesmo para ambos os procedimentos, ele funciona melhor para arquivos mais rígidos em estrutura, ou com divisão de blocos bem demarcada. De forma geral, o esquema funciona melhor na compressão da assinatura para arquivos que geram n, porém a capacidade de localizar modificações é melhor para n menor. Para o tamanho final das assinaturas não ser maior que o documento de entrada, a quantidade de caracteres por bloco (s) não pode ser muito baixa, porém a correção dos blocos exige s pequeno.

Para a divisão de texto em um bloco por linha implementada, as variações de desempenho para n não são tão significativas. Porém, na divisão de blocos para documentos XML, o incremento em tempo de execução é muito maior à medida que o número de blocos aumenta. Assim, para serem utilizados nesse esquema, os documentos XML não podem ser muito grandes, e também não são viáveis para correção devido à sua estruturação em *tags*.

Sobre a geração de CFFs, as 1-CFFs utilizando famílias de Sperner são muito mais eficientes em desempenho e não tem restrição de n de entrada, mas permitem localizar apenas $d = 1$ modificações. Já para CFFs de construção polinomial, ela é fácil e consistente para gerar e permite localizar $d = \frac{q-1}{k-1}$ modificações. Porém, essa construção não é muito flexível, pois para tirar maior proveito da relação $n = q^k$ e $t = q^2$, é preciso considerar q como sempre sendo uma potência de primo. Assim, ela é restrita acerca dos números de blocos utilizáveis, já que precisam também ser potências de primos, e n não pode ser arredondado para cima caso não siga essa propriedade. Dessa forma, o ideal seria buscar outras construções para criar implementações menos restritas.

Para a geração de assinaturas e localização de modificações, o desempenho segue o padrão estabelecido pelos parâmetros da CFF criada. Para maximizar a compressão da assinatura, é ideal utilizar CFFs maiores, com k alto. Já para maximizar a capacidade proporcional de localizar modificações, o ideal é CFFs menores, com k baixo e de preferência q baixo também. Já para aumentar de forma absoluta a quantidade de blocos modificáveis, é necessário aumentar a diferença entre o número de testes e blocos. Porém, o número de modificações não impacta no desempenho da localização, e não há restrição de número máximo de caracteres no bloco para realizá-la.

Acerca das escolhas de algoritmos criptográficos, o esquema de assinatura tradicional utilizado impacta pouco no desempenho comparativamente, mas o Ed25519 possui desempenho

melhor e mais segurança para a criação de assinaturas. Já a escolha de função de *hash*, junto do t gerado, define o comprimento da assinatura, mas não é tão significativa em desempenho na geração de assinatura para usos comuns (s pequeno). Porém, há impacto no desempenho da correção, relacionado ao comprimento dos resumos gerados. Essas escolhas não afetam a verificação de assinatura de forma significativa.

Sobre a correção, os blocos não podem ser divididos previamente para diminuir o seu comprimento. Para caracteres de 1 byte de comprimento, o maior s viável é igual 3, mas para esse valor não existem colisões para as funções testadas. Para a implementação criada, o tempo de execução aumenta proporcionalmente ao número de modificações devido à execução sequencial e não-paralela das correções. Como a aplicação não considera a existência de colisões, já que não existem para o s viável na prática, o tempo de execução da correção depende dos caracteres sendo corrigidos. Para permitir a correção de caracteres não contidos na codificação ASCII, o ideal é criar codificação própria para preencher os valores binários que não são convertidos para caracteres UTF-8 e manter o comprimento de 1 byte por caractere.

5.1 TRABALHOS FUTUROS

Para aumentar a usabilidade do MTSS, um dos principais fatores é a escolha de construção de CFF, já que ela dita o número de blocos utilizável, qual a capacidade de localização de erros e o comprimento da assinatura. É interessante desenvolver aplicações do esquema utilizando outras construções e comparar as implementações resultantes para desempenho e usabilidade prática.

Outra variação de implementação que pode ser feita é a da utilização do MTSS para manter a validade da assinatura para documentos que precisam remover ou censurar seções, com o propósito de manter a privacidade de certos dados do documento mas ainda ser possível garantir a sua autenticidade. Essa aplicação já é prevista em (IDALINO; MOURA; ADAMS, 2019, Seção 6), mas a implementação realizada neste trabalha não considerou este caso em específico.

Como este trabalho considera os casos mais genéricos, onde não se especifica quais são os blocos modificáveis ou o que pode ser modificado, é possível otimizar mais o uso do MTSS para aplicações específicas. Porém, para permitir a modificação de apenas determinados blocos, é necessário o uso de CFFs mais flexíveis e no momento mais estudos teóricos são necessários.

Mesmo para a construção de CFF já utilizada neste trabalho, é possível criar novas implementações com desempenho superior em relação à aplicação desenvolvida por meio de utilização de linguagens de programação e bibliotecas mais rápidas, computação multiprocesso para a montagem dos testes na geração e verificação de assinatura, e otimizações dos algoritmos do MTSS em si para que funcionem melhor na prática. A partir disso, também seria interessante analisar a variação de desempenho e viabilidade de parâmetros para computadores com capacidades de processamento variadas.

A prova de conceito criada buscou principalmente executar experimentos práticos acerca do esquema, e não possui o melhor desempenho ou usabilidade para o usuário final. Para que o MTSS seja utilizável em larga escala, é necessário não só ter bom desempenho, mas também criar uma aplicação com interface amigável para usuários que não conhecem os algoritmos do esquema, mostrando como variação de parâmetros afeta as funcionalidades do esquema de forma gráfica e de fácil compreensão.

Para permitir a correção de caracteres em português ou outras linguagens e manter o comprimento dos caracteres estabelecido, é necessário criar uma nova codificação de texto a partir da UTF-8 para preencher os valores binários que não possuem caracteres correspondentes para 8 bits. A codificação utilizada poderia ser selecionada pelo usuário, de forma a ter conjuntos de caracteres variados de acordo com o contexto de utilização do arquivo sendo assinado.

Outro fator que melhoraria a usabilidade do esquema é a possibilidade de assinar mais tipos de arquivos. Um formato muito utilizado hoje é o PDF, por exemplo, porém, devido à sua estrutura complexa, não foi implementada separação de blocos para ele, e seria interessante analisar formas diferentes de realizar essa divisão e como as implementações diferentes impactam nas capacidades de localização e correção para esse tipo de documento.

Por fim, um dos fatores que pode aumentar as possibilidades de utilização do esquema é a montagem de blocos, que neste trabalho foi explorada na prática de forma apenas sequencial. Para documentos estruturados em árvore, como XML e JSON, é possível criar os blocos considerando essa estrutura de forma a flexibilizar a modificação da mensagem, por exemplo ao permitir reordenação de certos blocos no caso de mensagens sem ordem garantida. Assim, seria interessante implementar essa abordagem e analisar como ela afetaria a utilização do MTSS.

REFERÊNCIAS

ABEL, P. de O. **Implementação do MTSS em Python**. 2023. Disponível em: https://github.com/paolabel/mtss-signer/releases/tag/v1.0. Acesso em: 12 jun. de 2023.

BARKER, A. R. E. **Size considerations for public and private keys**. NIST, 2019. Disponível em: https://doi.org/10.6028/NIST.SP.800-131Ar2. Acesso em: 11 jun. de 2023.

BERNSTEIN, D. J. et al. High-speed high-security signatures. **J. Cryptogr. Eng.**, v. 2, n. 2, p. 77–89, 2012. Disponível em: https://doi.org/10.1007/s13389-012-0027-1. Acesso em: 11 jun. de 2023.

BIYASHEV, R. G.; NYSSANBAYEVA, S. E. Algorithm for creating a digital signature with error detection and correction. **Cybernetics and Systems Analysis**, Springer, v. 48, n. 4, p. 489–497, 2012. Disponível em: https://doi.org/10.1007/s10559-012-9428-5.

BONIS, A. D.; CRESCENZO, G. D. Combinatorial group testing for corruption localizing hashing. In: **Computing and Combinatorics**. [S.l.: s.n.], 2011. (COCOON'11), p. 579–591. ISBN 978-3-642-22684-7.

BONIS, A. D.; CRESCENZO, G. D. **A Group Testing Approach to Improved Corruption Localizing Hashing**. 2011. Cryptology ePrint Archive, Paper 2011/562. https://eprint.iacr.org/2011/562. Disponível em: https://eprint.iacr.org/2011/562.

DWORKIN, M. J. Sha-3 standard: Permutation-based hash and extendable-output functions. **Federal Inf. Process. Stds. (NIST FIPS)**, National Institute of Standards and Technology, Gaithersburg, MD, p. 23, 2015. Disponível em: https://doi.org/10.6028/NIST.FIPS.202.

FÜREDI, Z. On r-Cover-free Families. **J. Combin. Theory Ser. A**, v. 73, n. 1, p. 172–173, 1996.

GATHEN, J. von zur. **Cryptoschool**. 1. ed. [S.l.]: Springer, 2015. ISBN 3662484234.

HWANG, D.-Z. D. F. K. **Combinatorial Group Testing and its Applications**. 2. ed. [S.l.]: World Scientific, 2000. ISBN 810241070.

IDALINO, D. P. R. F. C. L. M. T. B. Locating modifications in signed data for partial data integrity. **Information Processing Letters, Volume 115, Issue 10**, Elsevier, p. 731–737, 2015. Disponível em: https://doi.org/10.1016/j.ipl.2015.02.014.

IDALINO, T. B.; MOURA, L.; ADAMS, C. Modification tolerant signature schemes: Location and correction. In: HAO, F.; RUJ, S.; GUPTA, S. S. (Ed.). **Progress in Cryptology – INDOCRYPT 2019**. Cham: Springer International Publishing, 2019. p. 23–44. ISBN 978-3-030-35423-7.

IDALINO, T. B. et al. Locating modifications in signed data for partial data integrity. **Information Processing Letters**, Elsevier BV, v. 115, n. 10, p. 731–737, oct 2015. Disponível em: https://doi.org/10.1016

KALISKI, B. **PKCS #1: RSA Encryption Version 1.5**. RFC Editor, 1998. RFC 2313. (Request for Comments, 2313). Disponível em: https://www.rfc-editor.org/info/rfc2313. Acesso em: 07 jun. de 2023.

MENEZES, S. A. V. P. C. van O. A. J. **Handbook of Applied Cryptography**. 1. ed. [S.l.]: CRC Press, 1997. ISBN 0849385237.

PöHLS, H. C. **Increasing the Legal Probative Value of Cryptographically Private Malleable Signatures**. 2018. Ph.D. Thesis, University of Passau.

RUSZINKÓ, M. On the upper bound of the size of the r-cover-free families. **J. Combin. Theory Ser. A**, v. 66, n. 2, p. 302–310, 1994. ISSN 10960899.

STINSON, M. B. P. D. R. **Cryptography: Theory and Practice**. 4. ed. [S.l.]: Chapman and Hall/CRC, 2018. ISBN 1138197017.

The Unicode Consortium. **The Unicode® Standard**. Mountain View, CA: The Unicode Consortium, 2022. Disponível em: https://www.unicode.org/versions/Unicode15.0.0/UnicodeStandard-15.0.pdf. Acesso em: 18 jun. de 2023.

W3C. **Extensible Markup Language (XML) 1.1**. W3C, 2006. Disponível em: https://www.w3.org/TR/2006/REC-xml11-20060816/. Acesso em: 08 jun. de 2023.

W3C. **Canonical XML Version 2.0**. W3C, 2013. Disponível em: https://www.w3.org/TR/xml-c14n2/. Acesso em: 08 jun. de 2023.

W3TECHS. **Usage of character encodings broken down by ranking**. 2023. Disponível em: https://w3techs.com/technologies/cross/character_encoding/ranking. Acesso em: 07 jun. de 2023.

WEI, R. **On Cover-Free Families**. [S.l.], 2006. 1–19 p. Arxiv: arxiv.org/abs/2303.17524.

APÊNDICE A – ARTIGO DO TCC

Esquema de Assinatura Digital Tolerante a Modificações

Paola de O. Abel[1]

[1]Departamento de Informática e Estatística
Universidade Federal de Santa Catarina (UFSC)
Florianópolis – SC – Brasil

paola.o.abel@gmail.com

Resumo. *Tradicionalmente, as assinaturas digitais são geradas de forma a verificar a integridade e autenticidade do arquivo assinado como um todo, não permitindo qualquer tipo de modificação ao documento. Para permitir a localização e correção de modificações em documentos assinados, foi criado o esquema de assinatura digital tolerante a modificações (MTSS). Contudo, o funcionamento do esquema foi definido apenas de forma teórica, já que não houve implementação de seus algoritmos. Assim, o objetivo deste trabalho é estudar o esquema para criar uma aplicação prova de conceito para ele e analisar o seu funcionamento para obter resultados sobre sua viabilidade prática.*

1. Introdução

As assinaturas digitais foram criadas para fornecer a validação de veracidade de documentos dentro do ambiente digital, visando manter as propriedades de segurança das assinaturas em papel e caneta. Tradicionalmente, as assinaturas digitais são geradas de forma a verificar a integridade e autenticidade do arquivo como um todo, não permitindo qualquer tipo de modificação ao documento e retornando apenas respostas de sim ou não para sua validade.

Esse esquema de construção de assinatura funciona para a maior parte dos propósitos, contudo não é possível flexibilizar a forma de validação, permitindo que certas áreas do documento sejam modificáveis, por exemplo. Essa construção também inviabiliza visualizar onde ocorreu a alteração, o que pode ser útil para identificar se houve falha acidental ou tentativa intencional de modificar o conteúdo para benefício próprio. Por fim, sem a localização de erros, também não se pode pensar em corrigi-los, o que tornaria os sistemas que precisam de arquivos assinados mais autônomos e resistentes a falhas caso possam recuperar os documentos originais de forma automatizada.

Para permitir a localização e correção de modificações em documentos assinados, foi criado o esquema de assinatura digital tolerante a modificações (com sigla em inglês MTSS) [Idalino et al. 2019], mas seu funcionamento foi definido apenas de forma teórica, já que não houve implementação prática de seus algoritmos para analisar sua viabilidade em aplicações reais. Dessa forma, este trabalho se propõe a estudar o esquema, implementá-lo e analisar o funcionamento prático.

2. Fundamentação teórica

2.1. Cover-free families

Uma *d-cover-free family* (d-CFF) é uma família de conjuntos onde, ao unir quaisquer d conjuntos, nenhum dos outros conjuntos está contido nessa união. Ou seja, sempre

existirão 1 ou mais elementos em todos conjuntos restantes que não pertencem à essa união. Ela é definida por um par $F = (X, B)$, em que X é um conjunto e B é um conjunto de subconjuntos de X, chamados de blocos. Uma d-CFF é denotada por d-CFF(t,n), sendo que a cardinalidade do conjunto X é representada por $|X| = t$, o número de blocos é representado por $|B| = n$, e ela é construída seguindo $d < t \leq n$, com o objetivo de utilizar o menor t possível para determinados d e n. O limite inferior teórico para t, quando $d \geq 2$ é dado por $t(d, n) \geq c\frac{d^2}{\log d} \log n$ para alguma constante c [Wei 2006, Ruszinkó 1994, Füredi 1996], mas o limite real varia de acordo com o método utilizado para a construção da CFF. Já para $d = 1$, o limite é dado pelo Teorema de Sperner, como será apresentado na Seção 2.2.1.

Uma d-CFF(t,n) pode ser representada como uma matriz binária $t \times n$, também chamada de *d-disjunct matrix*, em que, para qualquer conjunto de $d+1$ colunas, a submatriz composta por essas colunas contém uma matriz de permutação de dimensão $d+1$. Tal representação é igual à matriz de incidência do sistema F correspondente, em que cada linha i corresponde a um ponto X_i de X, cada coluna j corresponde a um bloco B_i de B, e o valor binário a_{ij} indica se o ponto X_i está contido em B_j ou não.

A utilização de *cover-free families* neste trabalho é baseada na sua aplicação em *combinatorial group testing* ("testagem de grupos combinatória"), ou CGT, em que subconjuntos de n itens são testados para presença ou não de defeitos dentre seus elementos. Se assume que no máximo d itens serão defeituosos, de forma que a combinação dos itens dentro dos t testes junto de seus resultados $\{verdadeiro, falso\}$ possibilita determinar exatamente quais itens causaram os erros. Caso existam mais que d itens com defeito, a testagem não será conclusiva.

Para realizar a testagem utilizando CFFs, o conteúdo das colunas da CFF agora representa os itens a serem testados, cada linha representa um teste, B representa o conjunto de itens e X representa o conjunto de testes. Se um teste der resultado verdadeiro, então nenhum dos itens envolvidos no teste tem defeito, o que significa que eles são livres de defeitos não importa em quais outros testes aparecerem. Dessa forma, é possível unir todos os itens contidos em testes verdadeiros e então subtraí-los do conjunto de itens total para encontrar os itens defeituosos.

2.2. Construções

2.2.1. 1-CFFs

Quando $d = 1$, a CFF é considerada um *sistema de conjuntos Sperner* ou *família de Sperner*, que é um sistema de conjuntos com subconjuntos disjuntos entre si. Considerando a sua representação matricial, nenhuma de suas colunas pode conter o conteúdo de qualquer outra coluna [Hwang 2000, Seção 7.5].

Dado n e $d = 1$, o menor número de linhas possível para uma 1-CFF, de acordo com o número de colunas desejado, é determinado pelo Teorema de Sperner, em que t é igual ao menor t possível de forma que $\binom{t}{\lfloor t/2 \rfloor} \geq n$ seja verdadeiro [Hwang 2000, Seção 7.5]. Após t ser determinado, a 1-CFF é construída de forma que cada bloco é igual a um dos subconjuntos distintos de $\lfloor t/2 \rfloor$ elementos do conjunto $X = \{X_1, \ldots, X_t\}$. O conjunto B é formado por todos os subconjuntos de um tamanho fixo, e portanto nenhum $B_i \in B$ cobre nenhum outro $B_j \in B$. Esta construção dá uma 1-CFF(t, n) onde t se

aproxima do limite inferior teórico de número de testes para $d = 1$, já que $t \approx \log n$.

2.2.2. Construção polinomial para d-CFFs

Neste trabalho, a construção escolhida para d-CFFs será a construção polinomial devido à facilidade de se montar os conjuntos necessários em relação a outras construções existentes e bom limite inferior para t considerando sua complexidade.

A construção da d-CFF se baseia no uso de polinômios cujos coeficientes são elementos de um corpo finito, denotado por $\mathbb{F}_q$, sendo que q é o seu número de elementos, chamado de *ordem*, que precisa ser uma potência de número primo.

Para construir a d-CFF, é necessário um corpo finito $\mathbb{F}_q$, sendo q uma potência de primo, e um número inteiro $2 \leq k \leq q$. A partir de $\mathbb{F}_q$, é obtido $\mathbb{F}_q[x]_{<k}$, que será definido como o conjunto de todos os polinômios de uma única variável e grau menor que k: $a_0 + a_1x + a_3x^2 + \ldots + a_{k-1}^{k-1}$, cujos coeficientes $a_0, a_1, a_2, \ldots, a_k$ pertencem a $\mathbb{F}_q$. Por exemplo, se $\mathbb{F}_q = \mathbb{Z}/3$ e $k = 2$, teremos $\mathbb{F}_3[x]_{<2} = \{0, 1, 2, x, x+1, x+2, 2x, 2x+1, 2x+2\}$.

Considere q uma potência de primo, k um inteiro positivo e $\mathbb{F}_q = \{a_1, a_2, \ldots, a_q\}$ um corpo finito. Definimos uma (X, B) como a seguir. $X = \mathbb{F}_q \times \mathbb{F}_q$. Para cada polinômio $p \in \mathbb{F}_q[x]_{<k}$ de grau menor que k, temos um subconjunto associado $B_p = \{(a_1, p(a_1)), \ldots, (a_q, p(a_q))\}$, e por fim definimos $B = \{B_p : p \in \mathbb{F}_q[x]_{<k}\}$. O resultado obtido é uma d-CFF(q^2, q^k), com $d \leq \lfloor (q-1)/(k-1) \rfloor$.

2.3. MTSS

O esquema de assinatura digital tolerante a d modificações (MTSS) apresentado em [Idalino et al. 2019, Seção 4] permite a localização e correção de modificações em documentos assinados ao aplicar conceitos de *group testing* a assinaturas digitais, por meio da combinação de vários resumos criptográficos vindos do mesmo documento.

2.3.1. Localização de modificações

A localização das modificações é realizada ao se juntar a separação da mensagem em blocos com os testes de uma CFF. Ao invés do material a ser assinado ser composto apenas pelo hash da mensagem, agora ele é composto também pelos *hashes* de combinações de blocos da mensagem, de forma que a estrutura do material é montada a partir dos testes de uma d-CFF. Para o funcionamento dos algoritmos de geração e verificação de assinatura desse esquema, é necessário que eles compartilhem o uso de algum esquema de assinatura tradicional, função de resumo criptográfico pública e matriz de incidência de uma d-CFF. O MTSS tem sua segurança limitada pela segurança do esquema de assinatura e da função de resumo criptográfico utilizados [Idalino et al. 2019, Seção 5].

O funcionamento do algoritmo de geração de assinatura tolerante a d modificações é descrito a seguir. Ele recebe de entrada uma mensagem dividida em n blocos. Ele inicia criando os n *hashes* dos blocos da mensagem, que foram organizados anteriormente. A seguir, são realizadas concatenações de blocos de acordo com a matriz de incidência. Considerando que cada concatenação de blocos representa um teste da CFF (ou linha da matriz) e cada bloco da mensagem representa uma coluna da matriz, os blocos são

concatenados de forma que cada c_i tenha o conteúdo de um teste, ou seja, contenha os mesmos blocos na mesma combinação da matriz de incidência. Por fim, é criado um *hash* para cada concatenação, assim como o resumo da mensagem original, e a sequência de todos os resumos dos testes junto do *hash* da mensagem são identificados no algoritmo pelo vetor T, que é então assinado, gerando σ'. A assinatura final é composta por $\sigma = (T, \sigma')$ para que seja possível a localização de erros na verificação da assinatura. Sendo a o comprimento da saída do algoritmo de assinatura tradicional em bits, h o comprimento da saída da função de *hash* em bits e t o número de testes da CFF, o tamanho da assinatura do MTSS será igual a $a + h(t + 1)$ bits.

O funcionamento do algoritmo de verificação de assinatura tolerante a d modificações é descrito a seguir. Ele começa verificando a assinatura da junção de *hashes* dos testes e da mensagem, ao invés de verificar a assinatura da mensagem como um todo como no algoritmo tradicional. A verificação não irá funcionar caso algum resumo criptográfico em T, enviado junto da assinatura, tenha sido alterado. Se a assinatura é válida, então é possível afirmar se a mensagem foi alterada ou não. Como o *hash* h_m da mensagem foi assinado em conjunto aos testes, ele já foi verificado, então basta comparar o resumo enviado na assinatura com o *hash* da mensagem a ser verificada para determinar se a mensagem se manteve completamente íntegra. Caso exista diferença entre os dois resumos, então houve alteração na mensagem, e os passos seguintes do algoritmo servem para reconstruir as concatenações de blocos e compará-los com os enviados junto da assinatura. A localização das modificações é feita ao encontrar os índices dos blocos presentes em testes não modificados, e então subtraí-los do conjunto total de índices de blocos para encontrar os restantes, de forma similar aos procedimentos descritos na Seção 2.1.

2.3.2. Correção de modificações

O esquema base MTSS pode ser modificado para permitir a correção de erros ao limitar o tamanho dos blocos do documento. A limitação de comprimento dos blocos em s bits é necessária para o tempo de execução do algoritmo não ser inviável, já que, no pior caso, são necessárias 2^s operações para se encontrar todas as possibilidades de conteúdo de um bloco de s bits de comprimento [Idalino et al. 2019].

O funcionamento do algoritmo de verificação de assinatura tolerante a d modificações com correção de erros é descrito a seguir. Assim como no algoritmo de verificação sem correção, ele começa verificando a assinatura da tupla de *hashes* dos testes e da mensagem, mas fazendo o uso do algoritmo do MTSS, para encontrar os índices dos blocos modificados, ao invés de fazer a verificação de assinatura usando o esquema tradicional. Também há mudança no retorno do algoritmo, que agora inclui a mensagem corrigida. Caso o resultado da verificação seja falso, a correção retornada é vazia, para representar o erro na computação. O restante do procedimento é executado para corrigir todos os blocos contidos em testes que falharam.

3. Trabalhos relacionados

A localização de alterações em partes determinadas dos dados já foi proposta no contexto de funções de resumo criptográfico por [Bonis and Crescenzo 2011a] e [Bonis and Crescenzo 2011b] e no contexto de assinaturas digitais por

[Biyashev and Nyssanbayeva 2012] e [Idalino et al. 2015]. A correção de modificações é proposta por [Biyashev and Nyssanbayeva 2012], mas permite a correção de um único erro. Os outros trabalhos não a propõem, porém utilizam técnicas utilizando *cover-free families* para gerar a informação redundante necessária para localizar modificações nos dados. Em [Pöhls 2018], a divisão da mensagem em blocos é utilizada na criação de assinaturas em que blocos podem ser editados para esconder informações confidenciais.

Porém, nenhum dos trabalhos acima contém implementações de prova de conceito dos esquemas propostos, e portanto a contribuição desse trabalho para a literatura consiste na análise de viabilidade do esquema proposto em [Idalino et al. 2019], através da implementação de um esquema de localização e correção de até d blocos modificados de um documento assinado, para um limite superior d escolhido no momento da assinatura.

4. Implementação

Para realizar a análise de viabilidade prática do esquema apresentado anteriormente, ele foi implementado como prova de conceito usando Python 3.10. O código desenvolvido está disponível em repositório público no GitHub da autora [de Oliveira Abel 2023].

O desenvolvimento da aplicação foi realizado utilizando implementações pré existentes de um esquema de assinatura tradicional e função de *hash* comumente utilizados. Para gerar assinaturas tradicionais utilizando chaves RSA, foi escolhido o esquema PKCS#1 v1.5 [Kaliski 1998]. Também há a opção de criar assinaturas utilizando o algoritmo Ed25519. Já para a criação de resumos criptográficos, podem ser utilizadas como parâmetro as funções SHA2-256 (ou SHA256), SHA2-512 (ou SHA512), SHA3-256 e SHA3-512. As implementações utilizadas de tais algoritmos vieram da biblioteca de código aberto PyCryptodome.

Como parâmetros para controlar a criação da CFF que moldará a assinatura, é possível inserir o tamanho máximo em bytes desejado para a assinatura ou o parâmetro k da CFF, sendo que o número de blocos (n) é obtido diretamente do documento que se deseja assinar, dependendo do tipo de arquivo. Caso o número de blocos gerado para a mensagem não seja uma potência de primo ou os parâmetros inseridos gerem uma 1-CFF de construção polinomial, optou-se por gerar a assinatura baseada em 1-CFF utilizando famílias de Sperner devido à maior eficiência de armazenamento, geração e verificação da assinatura, já que gera um número de testes (*hashes*) menor. Para usar essa construção, também implementou-se a possibilidade de inserir $k = 1$ como atalho, já que é um valor inválido para criação de CFFs polinomiais. O tamanho mínimo válido para gerar uma assinatura depende do n gerado para a mensagem de entrada e do comprimento da assinatura necessário para o algoritmo de assinatura e chave privada escolhidos. Já a utilização do parâmetro k define o tamanho da assinatura gerada e a capacidade de localização de erros da assinatura, e pode ser usado para avaliar mudanças na viabilidade das operações do MTSS de acordo com a diferença ou variação dos parâmetros q ou k para criação de CFFs, porém esse parâmetro também determina quais valores de n podem ser utilizados, já que $n = q^k$.

A aplicação foi projetada para receber arquivos de texto (extensão .txt) ou arquivos XML, sendo que a extensão .txt pode ser utilizada para qualquer documento em texto plano (por exemplo, arquivos de código, CSV, e também XML). Os arquivos em formato PEM contendo as chaves precisam estar em formato PKCS#1 para chaves RSA

ou PKCS#8 para chaves Ed25519.

5. Análise de viabilidade e resultados

5.1. Análise de desempenho para *t* variado

O número de testes (t) para a assinatura é produto dos valores para q e k, conforme detalhado acima, então o crescimento em tempo de execução para aumento de t mas com n fixo será diferente do crescimento para variação de apenas q ou k, e também permite testar todas as possibilidades de assinatura MTSS para o mesmo arquivo.

A Tabela 1 mostra os resultados obtidos na execução do Algoritmo **??** para o mesmo arquivo de 4096 blocos mas com q e k variado, de forma a testar todas as possibilidades de CFFs polinomiais para o mesmo arquivo, o que gera números de testes (t) diferentes mas consegue manter o n fixo.

n	$\|m\|$	t	q	k	d	t. de execução	$\|\sigma\|$
4096	8191 B	15*	-*	-*	1*	0,3100 s*	0,755 kB
		64	8	4	2	0,3660 s	2,28 kB
		256	16	3	7	0,6663 s	8,28 kB
		4096	64	2	63	3,9154 s	128,28 kB

Table 1. Tempo médio de execução (tempo médio para 100 execuções) para assinatura de arquivo de texto com t variado, assinado usando chave RSA de módulo 2048.

*- Resultado para 1-CFF gerada via famílias de Sperner.

A Tabela 2 mostra os resultados obtidos na localização de 1 modificação (denotada por $|I|$, ou seja, o número de índices de blocos modificados para as assinaturas geradas na Tabela 1, utilizando em todas as análises um arquivo de texto similar ao assinado, mantendo o tamanho e número de blocos, mas com o mesmo caractere modificado.

n	$\|m\|$	$\|I\|$	t	k	$\|\sigma\|$	d	t. de execução
4096	8191 B	1	15*	-*	736 B*	1*	0,1937 s*
			64	4	2,25 kB	2	0,2151 s
			256	3	8,25 kB	7	0,3379 s
			4096	2	128,25 kB	63	3,0888 s

Table 2. Tempo médio de execução (para 100 execuções) da localização de 1 modificação no mesmo arquivo a partir das assinaturas geradas na Tabela 1.

*- Resultado para 1-CFF gerada via famílias de Sperner.

Os resultados mostram que o tempo de execução parece estar diretamente relacionado ao número de testes da assinatura, o que é esperado considerando o funcionamento do algoritmo, e é intensificado pela implementação sem multiprocesso da avaliação de T'. Dessa forma, para melhorar o desempenho dessa funcionalidade, o ideal é não criar assinaturas com t muito alto em relação a n.

5.2. Análise de desempenho para *s* variado

A Tabela 3 mostra os resultados obtidos para a correção da mesma linha de um arquivo de texto, mas para comprimento variado dos blocos na mensagem original, sendo que todos

os caracteres corrigidos são exclamações (U+0021) para minimizar o tempo de execução. Os arquivos assinados contém o mesmo conteúdo em todos os blocos, mas para cada experimento, a eles foram alterados antes da assinatura de forma a repetir caracteres e assim incrementar s de forma homogênea. Os tempos médios de execução (para 100 execuções) para a localização dos blocos são mostrados na coluna 't. verif.' para que os valores obtidos sejam diminuídos do resultado final na última coluna.

n	t	$\|\sigma\|$	d	$\|I\|$	char	$\|m\|$	s	t. verif.	t. de execução
4096	256	8,28 kB	7	1	! (U+0021)	8 kB	1	0,3379 s	0,0242 s
						12 kB	2	0,3260 s	0,6180 s
						16 kB	3	0,3176 s	2 min 33 s*
						20 kB	4	0,3228 s	10 h 24 min**
						24 kB	5	0,3211 s	107,5 d***

Table 3. Tempo médio de execução (para 100 execuções) de correção do mesmo bloco mas com comprimento em caracteres variável.

*- Resultado para 50 execuções.
**- Resultado para 1 execução.
***- Valor estimado com base na progressão da execução após 14,5 horas.

Os resultados mostram que há incremento drástico de tempo de execução para o aumento de s, mesmo com a utilização de computação paralela na implementação. Considerando que esse são os resultados para o melhor caso, onde todos os caracteres a serem corrigidos são U+0021, é perceptível a inviabilidade de correção de blocos com $s > 3$. De forma secundária, a tabela também mostra que valores pequenos para s não influenciam o tempo de localização para o mesmo número de modificações, mesmo que a correção aumente exponencialmente.

6. Conclusão

Este trabalho conclui que o MTSS funciona na prática para a prova de conceito desenvolvida. O esquema não é tão eficiente quanto esquemas tradicionais, porém ele permite localizar e até corrigir modificações ao invés de apenas descartar o documento caso ele tenha sido alterado, e a aplicação desenvolvida foi a primeira que permite mensurar na prática a escolha entre eficiência (de desempenho ou armazenamento) e capacidade de identificar e corrigir modificações. Contudo, devido às estruturas utilizadas para implementá-lo, suas vantagens vêm junto de certas restrições nos valores de entrada possíveis.

Devido à necessidade de gerar a mesma CFF para a geração e verificação de assinatura, ou seja, o número de blocos (n) precisa ser o mesmo para ambos os procedimentos, ele funciona melhor para arquivos mais rígidos em estrutura, ou com divisão de blocos bem demarcada. De forma geral, o esquema funciona melhor na compressão da assinatura para arquivos que geram n, porém a capacidade de localizar modificações é melhor para n menor. Para o tamanho final das assinaturas não ser maior que o documento de entrada, a quantidade de caracteres por bloco (s) não pode ser muito baixa, porém a correção dos blocos exige s pequeno.

Sobre a geração de CFFs, as 1-CFFs utilizando famílias de Sperner são muito mais eficientes em desempenho e não tem restrição de n de entrada, mas permitem localizar

apenas $d = 1$ modificações. Já para CFFs de construção polinomial, ela é fácil e consistente para gerar e permite localizar $d = \frac{q-1}{k-1}$ modificações. Porém, essa construção não é muito flexível, pois para tirar maior proveito da relação $n = q^k$ e $t = q^2$, é preciso considerar q como sempre sendo uma potência de primo. Assim, ela é restrita acerca dos números de blocos utilizáveis, já que precisam também ser potências de primos, e n não pode ser arredondado para cima caso não siga essa propriedade. Dessa forma, o ideal seria buscar outras construções para criar implementações menos restritas.

References

Biyashev, R. G. and Nyssanbayeva, S. E. (2012). Algorithm for creating a digital signature with error detection and correction. *Cybernetics and Systems Analysis*, 48(4):489–497.

Bonis, A. D. and Crescenzo, G. D. (2011a). Combinatorial group testing for corruption localizing hashing. In *Computing and Combinatorics*, COCOON'11, pages 579–591.

Bonis, A. D. and Crescenzo, G. D. (2011b). A group testing approach to improved corruption localizing hashing. Cryptology ePrint Archive, Paper 2011/562. `https://eprint.iacr.org/2011/562`.

de Oliveira Abel, P. (2023). Implementação do mtss em python.

Füredi, Z. (1996). On r-Cover-free Families. *J. Combin. Theory Ser. A*, 73(1):172–173.

Hwang, D.-Z. D. F. K. (2000). *Combinatorial Group Testing and its Applications*. World Scientific, 2 edition.

Idalino, T. B., Moura, L., and Adams, C. (2019). Modification tolerant signature schemes: Location and correction. In Hao, F., Ruj, S., and Sen Gupta, S., editors, *Progress in Cryptology – INDOCRYPT 2019*, pages 23–44, Cham. Springer International Publishing.

Idalino, T. B., Moura, L., Custódio, R. F., and Panario, D. (2015). Locating modifications in signed data for partial data integrity. *Information Processing Letters*, 115(10):731–737.

Kaliski, B. (1998). PKCS #1: RSA Encryption Version 1.5. RFC 2313.

Pöhls, H. C. (2018). Increasing the legal probative value of cryptographically private malleable signatures. Ph.D. Thesis, University of Passau.

Ruszinkó, M. (1994). On the upper bound of the size of the r-cover-free families. *J. Combin. Theory Ser. A*, 66(2):302–310.

Wei, R. (2006). On Cover-Free Families. Technical report, Lakehead University. arxiv: `arxiv.org/abs/2303.17524`.