Practical algorithms and parameters for modification-tolerant signature scheme

Anthony B. Kamers[1], **Paola de Oliveira Abel**[1], **Thaís B. Idalino**[1], **Gustavo Zambonin**[1], **Jean E. Martina**[1]

Departamento de Informática e Estatística – Universidade Federal de Santa Catarina (UFSC)
Caixa Postal 476 – 88040-370 – Florianópolis – SC – Brazil

{anthony.kamers,gustavo.zambonin}@posgrad.ufsc.br,
paola.abel@grad.ufsc.br, {thais.bardini,jean.martina}@ufsc.br

Abstract. *Traditional digital signature schemes are insufficient to identify exactly which part of a signed document had its integrity compromised. In INDOCRYPT '19, Idalino et al. presented an efficient modification-tolerant signature scheme (MTSS) framework using group testing techniques, enabling the detection and correction of modified parts. However, the authors did not give ideal parameters for real use case scenarios. We implement the framework, discuss the practical consequences of the effort, give several parameter sets, and compare the performance of MTSS against traditional signature schemes. We additionally propose a novel use case of the framework, which allows for the integrity of any part of a signed document to be verified without ownership of the whole message.*

1. Introduction

A traditional digital signature scheme enables a user to claim the integrity and authenticity of some signed data. The signature verification algorithm has a boolean output, which allows for the *detection* of modifications; it reports failure if a single bit of a signed document is modified. In practical terms, the signer and the information contained within the document may not be trusted. However, if additional properties such as *location* and *correction* of possible modifications [Idalino et al. 2019] are considered, digital signatures can be employed in a variety of new scenarios.

In [Idalino et al. 2015], the authors cite several important use cases: (i) modification discovery in fillable forms, allowing one to sign the empty form and others to fill it in without invalidating the original signature; (ii) crime investigations by data forensics, where the investigator may retrieve more information about the attacker by knowing what was modified [Goodrich et al. 2005a]; (iii) improving the efficiency of computer systems: for instance, if we detect where a large database was modified, we do not invalidate it; (iv) privacy protection, where parts of a signed document can be intentionally redacted without invalidating the original signature (cf. content extraction signatures [Steinfeld et al. 2002] and redactable signatures [Johnson et al. 2002, Haber et al. 2008]).

In the literature, the location property was addressed by [De Bonis and Di Crescenzo 2011a, De Bonis and Di Crescenzo 2011b] in the context of hash functions, by [Di Crescenzo et al. 2004, Goodrich et al. 2005b] in the context of message authentication codes, and by [Idalino et al. 2015, Idalino et al. 2019] in the context of digital signatures. In general, the authors propose to compute extra integrity information that can be used later for the location of modifications.

There also exist situations where portions of the data can be intentionally removed, redacted, or modified. This is the case of malleable signature schemes, which are studied under the name of *redactable* and *sanitizable* signatures [Bilzhause et al. 2017]. They allow the modification of the signed data in a controlled way (in some cases, by a third party) while the signature is still successfully verified. Such schemes are commonly applied in privacy settings, in which portions of the target data are required to be redacted [Johnson et al. 2002, Haber et al. 2008, Lim and Lee 2011].

Notably, the *modification-tolerant signature scheme* (MTSS) framework employs combinatorial group testing techniques, using *cover-free families* (CFFs), to locate modified parts of a signed document [Idalino et al. 2019]. Intuitively, a document is split into blocks and grouped according to a CFF on signature generation. When verifying the signed document, the framework allows for the location of modified blocks via the underlying CFF. If the blocks are small enough, it is possible to recover the original file, depending on the parameter choices of the scheme. MTSS can be used with any signature scheme, such as RSA, elliptic curves, or post-quantum algorithms.

We address some open questions of the proposal as laid out in [Idalino et al. 2019]. Namely, we concretely implement MTSS in a high-level programming language and use different underlying signature schemes to measure its performance. We debate which parameters are feasible for constructing CFFs in many scenarios and measure the performance of signature algorithms with such constructions. We also discuss strategies to divide to-be-signed documents into blocks, considering different types of documents and the criteria that affect error identification during signature verification. Additionally, we propose a novel usage of a variation of MTSS that verifies the authenticity and integrity of some parts of a signed document without having access to the entire signed data. Our contributions rely on modification-tolerant signature schemes regarding authentication and data integrity. In this work, we do not explore applications regarding privacy (such as redactable and sanitizable signatures).

This work is organized as follows. In Sect. 2, we briefly give the necessary background to understand MTSS and our contributions. In Sect. 3, we present our specific objectives and give technical details about the practical implementation of MTSS. In Sect. 4, we discuss the selection of parameters and other implementation choices for MTSS; we also provide several performance and storage measurements. In Sect. 5, we propose a variation of MTSS that verifies the integrity and authenticity of a single block of the document without having access to the whole document. Finally, in Sect. 6, we summarize our contributions and suggest further contributions as future work.

2. Definitions

2.1. Cover-free families

A set system is a tuple $(X, \mathcal{B})$, where X is a set of points and $\mathcal{B} = \{B_1, \ldots, B_n\}$ is a collection of subsets of X, called blocks. Intuitively, a d-cover-free family is a set system where the union of any d blocks does not cover any other block in $\mathcal{B}$; a formal definition is given as follows.

Definition 1 **(d-CFFs)** *Let d, t, n be positive integers, with $d < t \leq n$. A d-cover-free family, denoted d-CFF(t, n), is a set system $(X, \mathcal{B})$ with $|X| = t$, $|\mathcal{B}| = n$, where for any block B_{i_0} and any other d blocks $B_{i_1}, \dots, B_{i_d}$ in $\mathcal{B}$ we have $|B_{i_0} \setminus \cup_{j=1}^{d} B_{i_j}| \geq 1$.*

We may also define a d-CFF(t, n) in terms of its incidence matrix $\mathcal{M}$: a $t \times n$ binary matrix with $\mathcal{M}_{i,j} = 1$ if $x_i \in B_j$, and 0 otherwise. In the literature, this matrix representation is also known as a d-disjunct matrix. We hereafter say a binary matrix is d-CFF if its corresponding set system is d-CFF. Figure 1a shows an example of a 2-CFF$(9, 12)$ incidence matrix where $X = \{1, \dots, 9\}$, the collection $\mathcal{B} = \{B_a, B_b, \dots, B_l\}$, and $B_a = \{1, 2, 3\}, B_b = \{4, 5, 6\}, \dots, B_l = \{3, 5, 7\}$. CFFs are used in combinatorial group testing, where n items must be tested, and we can detect at most d items as "defective". The items are represented by the columns of the incidence matrix $\mathcal{M}$, and the rows identify the groups to be tested.

The d-CFF property allows us to perform only t tests and identify up to d defects among the n items in the following way: groups that pass the test contain only non-defective items; the remaining ones are considered defective. Figure 1b shows how we can identify up to 2 defective items using a 2-CFF$(9, 12)$. Groups $3, 5, 8, 9$ passed the test, and therefore the items in these groups are all non-defective (in green). Groups that fail the test must contain at least one defective item, and so after identifying the non-defective items, the remaining ones are the defective ones (in red). If there are more than d defective items, some non-defective items might be erroneously considered defective due to the group testing technique.

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

a. Incidence matrix of a 2-CFF$(9, 12)$.

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

b. Procedure to identify defects.

Figure 1. Matrix representation of a CFF and defect location procedure.

Hence, given n and d, we aim to build d-CFFs that minimize the number of rows t. In the case $d = 1$, an optimal construction is given by Sperner set systems. For a given n, we set $t = \min\{s : \binom{s}{\lfloor s/2 \rfloor} \geq n\}$. Then, let $X = \{1, 2, \dots, t\}$ and $\mathcal{B}$ be the collection of all distinct subsets of X of size $\lfloor t/2 \rfloor$ [Sperner 1928, Wei 2006]. Since all subsets are distinct and have the same size, no subset is contained in any other, yielding a 1-CFF(t, n).

For $d \geq 2$, it is known that $t \geq c\frac{d^2}{\log d} \log n$ for a constant c [Füredi 1996, Ruszinkó 1994, Wei 2006]. The best existence results are given by probabilistic methods, achieving $t = \Theta(d^2 \log n)$ [Bshouty 2015, Gargano et al. 2020, Porat and Rothschild 2011, Rescigno and Vaccaro 2023]. We hereafter consider a polynomial construction based on finite fields, first presented in [Erdös et al. 1985].

Consider a finite field $\mathbb{F}_q$, with q a prime power and $2 \leq k \leq q$. We define $\mathbb{F}_q[x]_{<k}$ to be the set of all polynomials with degrees less than k and coefficients in $\mathbb{F}_q$. For $d \leq \lfloor \frac{q-1}{k-1} \rfloor$, a d-CFF(q^2, q^k) defined by the set system $(X, \mathcal{B})$ is constructed as follows. Let $X = \mathbb{F}_q \times \mathbb{F}_q$; then, for each polynomial $p \in \mathbb{F}_q[x]_{<k}$ we have an associated subset $B_p = \{(a_1, p(a_1)), \ldots, (a_q, p(a_q))\}$, and consequently $\mathcal{B} = \{B_p : p \in \mathbb{F}_q[x]_{<k}\}$. For more details on cover-free families, applications, and other constructions for the case $d \geq 2$, see [Idalino and Moura 2024].

2.2. The modification-tolerant signature scheme framework

As previously mentioned, the MTSS framework [Idalino et al. 2019] is able to locate d modifications in a signed document using a d-CFF(t, n). On signature generation, a to-be-signed document must be split into n blocks of s bits to fit the underlying CFF. For blocks of small enough size, the authors also propose correcting such modifications. The signature must carry extra information in the form of cryptographic hashes of blocks in each row (or test). From now on, we may refer to *test* as the document hashes computed using a CFF. Thus, signature size depends on t, d, and n, and it is crucial to provide parameter sets compatible with several use cases.

Let Σ be a digital signature scheme, with the usual key generation (KEYGEN), signature generation (SIG), and signature verification (VER) algorithms; let $\mathcal{H}$ be a cryptographic hash function, and $\mathcal{M}$ a d-CFF(t, n) incidence matrix. We hereafter refer to a complete instantiation of the framework as MTSS$(\Sigma, \mathcal{H}, \mathcal{M})$. MTSS has 4 algorithms: KEYGEN, SIG, VER and "verify-and-correct" (VCOR); a brief description of each one is given as follows. We also define the set of modified blocks identified by the framework as I. Let $\top$ be a valid output, and $\bot$ otherwise.

KEYGEN(λ). Let $\lambda \in \mathbb{N}$ be the security parameter. The algorithm proceeds as follows.

1. Set $(\mathsf{sk}, \mathsf{pk}) \leftarrow \Sigma.\text{KEYGEN}(\lambda)$.
2. Output $(\mathsf{sk}, \mathsf{pk})$.

SIG(m, sk). Let sk be a private key of Σ, and $m = (m_1, \ldots, m_n)$ any message divided into n blocks. The algorithm proceeds as follows.

1. For $1 \leq i \leq t$ and $1 \leq j \leq n$, set $h_j \leftarrow \mathcal{H}(m_j)$, c_i the concatenation of all h_j such that $\mathcal{M}_{i,j} = 1$, and $T_i \leftarrow \mathcal{H}(c_i)$.
2. Set $T \leftarrow (T_1, \ldots, T_t, \mathcal{H}(m))$.
3. Set $\sigma' \leftarrow \Sigma.\text{SIG}(T, \mathsf{sk})$.
4. Output (σ', T).

VER(m, σ, pk). Let $m = (m_1, \ldots, m_n)$ be any message divided into n blocks, $\sigma = (\sigma', T)$ a MTSS signature, and $T = (T_1, \ldots, T_t, h_m)$, and pk a public key of Σ. The algorithm proceeds as follows.

1. Set $r \leftarrow \Sigma.\text{VER}(T, \sigma', \mathsf{pk})$. If $r = \bot$, output $(\bot, \{\})$. Otherwise, go to step 2.
2. If $\mathcal{H}(m) = h_m$, output $(\top, \{\})$. Otherwise, go to step 3.
3. For $1 \leq i \leq t$, compute T_i' analogously to Step 1 of SIG.
4. Set $V \leftarrow \{\}$. For $1 \leq i \leq t$, if $T_i = T_i'$, set $V \leftarrow V \cup \{j : \mathcal{M}_{i,j} = 1\}$.
5. Set $I \leftarrow \{1, \ldots, n\} \setminus V$. If $|I| \leq d$, output $(\top, I)$; otherwise, output $(\bot, I)$.

VCOR(m, σ, pk). Let $m = (m_1, \ldots, m_n)$ be any message divided into n blocks, $\sigma = (\sigma', T)$ a MTSS signature, and pk a public key of Σ. The algorithm proceeds as follows.

1. Set $(r, I) \leftarrow \textsc{Ver}(m, \sigma, \mathsf{pk})$. If $r = \bot$, output $(\bot, \{\}, \{\})$. Otherwise, go to step 2.
2. If $|I| = 0$, go to step 7. Otherwise, set $b \leftarrow \min(I)$, $I \leftarrow I \setminus b$ and go to step 3.
3. Set i to be the index of any row $\mathcal{M}_i$ such that $\mathcal{M}_{i,b} = 1$ and $\mathcal{M}_{i,j} = 0$ for all $j \in I$.
4. For all j such that $\mathcal{M}_{i,j} = 1$ and $j \neq b$, set $h_j \leftarrow \mathcal{H}(m_j)$. Set $c_b = \bot$.
5. For every possible bit string q of size $\leq s$:
   (a) Set $h_b \leftarrow \mathcal{H}(q)$.
   (b) For $1 \leq j \leq n$, compute T'_i analogously to Step 1 of Sig.
   (c) If $T'_i = T_i$ and $c_b = \bot$, set $c_b = \top$ and $m_b \leftarrow q$; otherwise, if $T'_i = T_i$ and $c_b = \top$, output $(\top, I, \varepsilon)$, where ε is the empty string.
6. Go to step 2.
7. Output $(\top, I, m)$.

We refer the reader to [Idalino et al. 2019] for a detailed algorithm explanation and proof of the security and correctness of the framework. We remark on an interesting characteristic of MTSS: given only a signature σ and the number of blocks n, we can reconstruct $\mathcal{M}$. This may be used for implementation purposes. In the following, we address the practical consequences of the algorithms above, such as the division of m into n blocks, the construction of $\mathcal{M}$, and the impact of Σ, $\mathcal{H}$, and the aforementioned procedures in overall performance and signature sizes.

3. Discussion on MTSS parameters

As previously mentioned, several practical considerations were not given or deeply explored in [Idalino et al. 2019]. We present the following questions as a guideline for our discussion about parameters and their consequences on implementations of MTSS.

$\mathbf{Q_1}$ *How to efficiently implement message division, and what are the consequences for* Sig*,* Ver*, and* VCor*?*

We recall that all algorithms except KeyGen expect to receive a message m already separated in n blocks; however, in practice, a user expects to simply input the entire to-be-signed message, and an implementation of MTSS performs the appropriate division. Hence, we define a DivideBlocks(m') algorithm, which separates the input message m' into blocks according to some criteria depending on the file type. Its output is a tuple (m, n), where $m = (m_1, \ldots, m_n)$. Our implementation does not consider n as a free input because of possible issues in locating errors using Ver algorithm; rather, we infer n from the type of document chosen. We further address this question in Sections 3.1 and 4.1.

$\mathbf{Q_2}$ *Which parameters are suitable for* $\mathcal{M}$ *and what is their effect on performance?*

As per Sect. 2.1, a CFF is defined via parameters d, t, and n, with t minimized. We recall that n is the number of blocks of a message m' given by DivideBlocks(m'), as discussed in $\mathbf{Q_1}$. It remains to calculate d and t; we recall that the parameter k can be used to yield d and t, and also q for $k \geq 2$, via the constructions presented in Sect. 2.1. Hence, we define the CreateCFF(n, k) algorithm as follows: if $k = 1$, the Sperner construction is used, where $d = 1$; otherwise, we use the polynomial over finite fields construction, in which d depends on q and k, i.e., given n and k, we choose the smallest q such that $n \leq q^k$. Its output is the incidence matrix $\mathcal{M}$. We further address this question in Sections 3.2 and 4.1.

$\mathbf{Q_3}$ *How does the choice of* $\mathcal{H}$ *impact overall performance?*

We observe that SIG and VER require $nt + 1$ evaluations of $\mathcal{H}$. Thus, the choice of the underlying cryptographic hash function greatly influences the performance of operations on signatures. We evaluate several choices of $\mathcal{H}$, and give performance results and recommendations in Sections 4.2 and 4.3.

$\mathbf{Q}_4$ *How does the choice of* Σ *impact overall performance?*

We observe that the authors of [Idalino et al. 2019] roughly estimate the performance of MTSS depending on Σ. We explore other choices of signature schemes and give actual performance and signature size results in Sections 4.2 and 4.3.

3.1. Documents and blocks division

We remark that CFF blocks need not be homogeneous in size; this is useful for digital documents since each file type has specific syntactic characteristics. However, as per Step 5 of the VCOR algorithm, a large block implies a slower error correction procedure since all possible bit strings need to be searched to correct the block. Hence, efficient block division criteria are relevant in this context. Several authors [Idalino et al. 2019, Pöhls 2018] consider strategies such as sequential ordering considering some delimiter, non-sequential ordering using complex structural data, a header at the beginning of the file informing the blocks, or even a combination of these.

The easiest way to separate the content into blocks is to divide a message into sequential sections of fixed size. However, this approach is not free of problems [Idalino et al. 2015]. Without loss of generality, for any bit string that is divided into blocks, if any bit is added to or removed from any block but the last, a cascade effect happens to the subsequent blocks. To prevent this problem, we must define document representations and block delimiters for the DIVIDEBLOCKS algorithm.

We first consider plain text files using the ASCII encoding; we set the line break (`0x0a`) character as the default delimiter. Hence, the number of lines must remain the same to detect changes, but any individual lines are modifiable. We also discuss how to interpret XML files (also encoded in ASCII) as an example of a more complex document type. As canonicalization algorithms usually remove indentation and spaces [W3C 2008], such characters cannot be used as block delimiters.

The $<$ (`0x3c`) character can be used as a delimiter, but we would have redundancy among different types of tags (opening and closing). Hence, we choose to pair opening and closing tags in the same block. The parent tag is kept in a separate block if any child elements exist. One disadvantage of this approach is the excess of usual tag delimiter characters since they are always put in the blocks. This approach can also be applied to other similar markup systems, such as HTML.

A more natural way for hierarchy file types could be using a tree as structural data. This way, we could separate each node as a tag and their children as connected to this parent node. This separation method could also be used for other hierarchy-like files. Nevertheless, we show hereafter that the DIVIDEBLOCKS algorithm has poor efficiency using the simple pair opening and closing tags, and using a tree would add more complexity to that. Besides, we would have the override of implementing a search algorithm to access a particular node, which would also affect the performance of MTSS algorithms.

3.2. CFF parameters

We remark some initial observations from the algorithms described in Sect. 2.2: (i) the signature size grows with the number of blocks n; (ii) to efficiently correct modifications, it is suggested to limit block size to a small enough size s. However, this minimization leads to more blocks; (iii) larger blocks lead to less accuracy in error location and fewer tests t executed. In this context, we aim to find ideal parameters for polynomial constructions, i.e., $d \geq 2$. From Sect. 2.1, we recall that $n = q^k$ or $n = t^{k/2}, t = q^2$, and $d \leq \lfloor \frac{q-1}{k-1} \rfloor$. As previously mentioned, we calculate q from n and k via the CREATECFF algorithm.

To generate smaller CFFs and improve efficiency, we increase the gap between the number of blocks and tests to minimize t. The proportion between n and t is given by q^{k-2}, which gives us the signature compression rate; hence, larger values for k lead to further gains in compression. However, due to the upper bound of d, we observe that smaller k are useful for achieving larger error location abilities. On the other hand, $q \leq 5$ are not generally useful because they generate 1-CFFs, which are already optimally built using Sperner set systems. The same occurs for $k \geq 7$ since most constructions with this parameter can be achieved via Sperner families or are meant for larger n.

A d-CFF(q^2, q^k) allows locating up to d modifications. The ratio between d and n is given as follows: $\frac{d}{n} = \frac{(q-1)/(k-1)}{q^k}$. This leads us to a conclusion: for fixed n, smaller values of q have a better proportion of modifiable blocks d. Furthermore, the aforementioned relation between d, q, and k shows that d grows with q. These statements help us answer $\mathbf{Q_2}$. In summary, smaller values for k have more advantages: they optimize the modification localization while obtaining modifiable blocks at a higher number and proportion. Hence, we suggest $3 \leq k \leq 7$ and $q > 5$ to yield efficient constructions when $d \geq 2$.

4. Experiments

We provide an open-source implementation[1] of the MTSS framework as described in the original work [Idalino et al. 2019]. We use Python 3.10 and run the tests on an Intel Core i7-13700H @ 5.0 GHz with CPU performance scaling disabled. Except when otherwise indicated, the performance results presented in this section were calculated by executing each algorithm 100 times and taking the mean. For the choices of Σ, we consider (i) RSA-2048 and RSA-4096, which are known to be widely used in public-key cryptographic applications [Cao and Fu 2008]; (ii) Ed25519, an elliptic-curve signature scheme which is a recent alternative to RSA, with smaller signatures and higher overall performance [Bernstein et al. 2012]; (iii) and all instances of ML-DSA, a post-quantum algorithm based on lattices which is currently in the process of standardization by the U.S. National Institute of Standards and Technology [of Standards and Technology 2023].

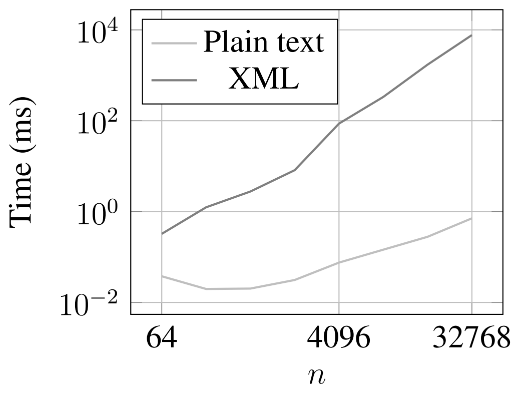
We also consider several cryptographic hash functions for the choice of $\mathcal{H}$: SHA2-256, SHA2-512, SHA3-256, SHA3-512, BLAKE2s, and BLAKE2b. We argue that these choices of Σ and $\mathcal{H}$ represent a wide range of use cases and allow for easier reproducibility of our results while also considering state-of-the-art algorithms. We note that $\mathcal{H}$ may vary according to Σ, except in the case of Ed25519, where it is required that $\mathcal{H}$ = SHA2-512 [Josefsson and Liusvaara 2017]). Otherwise, we test all $\mathcal{H}$ against all Σ to compare the performance of signature generation and verification in the context of MTSS.

[1]`https://github.com/AnthonyKamers/mtss-signer/`

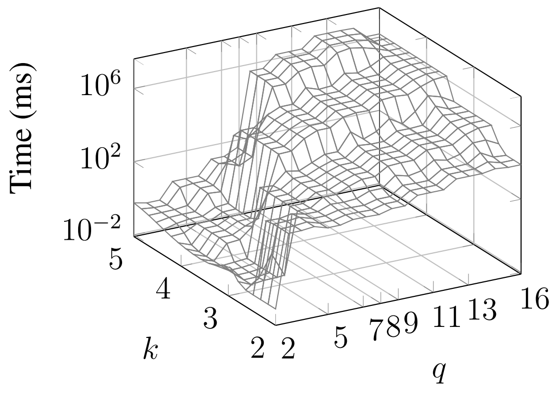
We follow the guidelines of the original authors of MTSS in our implementation. Particularly, we parallelize hash computations whenever possible in VCOR. We anticipate that CFF generation is costly when on-demand, i.e., in every computation of SIG, and if errors are detected by VER. Therefore, we implement a simple cache for d-CFF(t, n) with distinct d, t, n to prevent such overhead. In our experiments, we assume that all CFFs used are already cached and previously serialized to memory, except when otherwise noted. Finally, we execute DIVIDEBLOCKS before every computation of SIG, VER, and VCOR.

4.1. Auxiliary algorithms

As previously mentioned, auxiliary algorithms for MTSS create overhead in signature generation and verification. In the following, we answer $\mathbf{Q_1}$ and $\mathbf{Q_2}$ by respectively discussing the practical consequences of implementing DIVIDEBLOCKS and CREATECFF. Due to time constraints, both algorithm's performance tests were executed only once.



a. Performance of DIVIDEBLOCKS.



b. Performance of CREATECFF.

Figure 2. General performance of auxiliary algorithms for several choices of k, q, n.

Document division. Figure 2a summarizes the performance of DIVIDEBLOCKS considering plain text and XML documents of several sizes. As n increases, XML files require more time to separate into blocks due to the additional overhead of canonicalization, syntactic validation, and structure parsing. However, according to [Mlynkova et al. 2006], the average size of XML documents is around 4.6 kB, containing a small number of nodes; moreover, 99% of XML documents have fewer than eight levels. This suggests that, despite larger XML documents taking longer to separate into blocks, most real-world XML documents are relatively short and simple, making them suitable for use with MTSS in practical applications. Nonetheless, since parsing plain text files is considerably simpler, our subsequent experiments will focus on this file type.

CFFs. We first note that, for a fixed number of blocks n, the relative number of tests t is $\approx 173.39\%$ larger when employing polynomial constructions and takes $\approx 2000\times$ longer to execute. Thus, for the case $d = 1$, Sperner families are the most efficient solution. However, we recall that the polynomial construction is required if $d \geq 2$. Figure 2b shows the performance of creating CFFs with polynomial constructions for several choices of k and q. We observe that as q increases, the execution becomes exponentially slower since $n = q^k$ and, as n grows, the entire CFF grows. A particularly efficient point is shown between $5 \leq q \leq 8$, and $k = 5$, where the parameters are among the ones proposed in Sect. 3.2, generates a large CFF and in a reasonable time ($<$ 40ms).

While $q > 5$ produces more valuable CFFs, performance tends to decline for values greater than this threshold. Techniques such as pre-constructing the CFFs can mitigate this issue and improve the framework's overall efficiency. If we consider using MTSS in a central resourceful server, CFFs could be made in advance and stored locally, which would not disturb other algorithms' performance.

4.2. Signing

We divide the overall signature generation procedure into two separate stages, *pre-sign* and *sign* (SIG proper), for ease of discussion. In the *pre-sign* stage, we handle I/O operations, segment the message into blocks based on the document type using DIVIDEBLOCKS, and create or parse the cached CFF with CREATECFF. We integrate both algorithms in the *pre-sign* stage. As a practical consequence, the implementation of SIG takes an additional parameter k so that auxiliary algorithms need not be invoked by the user.

During the *pre-sign* stage, the document m' (not parsed into blocks yet) is read from the disk, processed by DIVIDEBLOCKS(m'), and subsequently by CREATECFF(n, k). The secret key sk is also retrieved from the disk, preparing all necessary components for the *sign* stage. Given the potential computation cost of these operations, we provide examples illustrating the execution time of both stages in the following.

The hash function $\mathcal{H}$ and the signature scheme Σ used in the framework significantly impact the signing performance. Table 1 shows a performance comparison of different signature schemes and hash functions in the context of our implementation. We do not consider the *pre-sign* stage in performance measurements. The last two columns represent the size of the resulting MTSS signature in bytes, depending on the output size (in bits) of $\mathcal{H}$. This comparison addresses $\mathbf{Q_3}$, demonstrating that the choice of the hash function $\mathcal{H}$ plays a crucial role in MTSS performance. Furthermore, the signing time of MTSS-augmented Σ remains consistent with traditional Σ, which answers $\mathbf{Q_4}$.

Based on these findings, we hereafter consider $\Sigma =$ RSA-2048 and $\mathcal{H} =$ BLAKE2b for our next experiments, as RSA-2048 is commonly used in practice and BLAKE2b showed the best performance among the evaluated hash functions. The performance of *sign* is influenced by the number of tests t in $\mathcal{M}$. We identify that increasing the number of tests results in longer signing times and larger signatures, as shown in Table 2. The upper group demonstrates how performance is affected by the size of the input file for fixed k. On the other hand, the lower group shows how different k affects the same file. It is important to note that the same file can have varying numbers of tests t, depending on the chosen values for k. The framework efficiency is reduced as tests or the number of blocks increases. However, MTSS creates stronger signatures that make it easier to identify document alterations, which is crucial in the discussed scenarios.

The *pre-sign* stage can sometimes take longer than *sign*, depending on the input file (and subsequent number of blocks). As discussed in Sect. 4.1, parsing large XML files can be time-consuming. This is an important consideration when using the MTSS framework. Figure 3 compares the performance of the *pre-sign* and *sign* stages, showing the total time to sign different files with $k = 4$. For simplicity, each file is named `n.ext`, where n is

Table 1. Performance and output size of MTSS($\Sigma, \mathcal{H}, \mathcal{M}$).Sɪɢ for several choices of Σ, $\mathcal{H}$ and security parameter λ, considering $\mathcal{M} = 2$-CFF($25, 125$), $k = 3$ and a 1.19MB plain text file as input.

			Sɪɢ time (ms)						$\|\sigma\|$ (bytes)	
			SHA-2		SHA-3		BLAKE			
	λ	Σ	256	512	256	512	2s	2b	256	512
Raw Σ	128	RSA-2048	4.83	3.63	3.93	6.42	2.49	3.32	256	256
	128	ML-DSA-44	3.93	2.67	2.96	5.49	1.54	2.36	2360	2360
	128	Ed25519		3.08						64
	256	RSA-4096	8.83	7.64	7.96	10.44	6.54	7.38	512	512
	256	ML-DSA-65	3.98	2.71	2.98	5.55	1.60	2.49	3220	3220
	512	ML-DSA-87	4.10	2.73	3.02	5.58	1.62	2.41	4490	4490
MTSS	128	RSA-2048	27.35	19.42	21.76	36.85	10.86	15.63	1088	1880
	128	ML-DSA-44	26.44	18.85	21.27	36.04	10.36	15.2	3180	3990
	128	Ed25519		19.99						1690
	256	RSA-4096	31.32	23.37	25.74	40.87	14.9	19.66	1310	2120
	256	ML-DSA-65	26.63	18.83	21.1	35.93	10.22	15.29	4030	4840
	512	ML-DSA-87	26.76	18.98	21.12	36.17	10.02	15.05	5300	6110

the number of blocks and `ext` represents plain text and XML files. When considering complex files, we note that *pre-sign* can overtake the *sign* stage time.

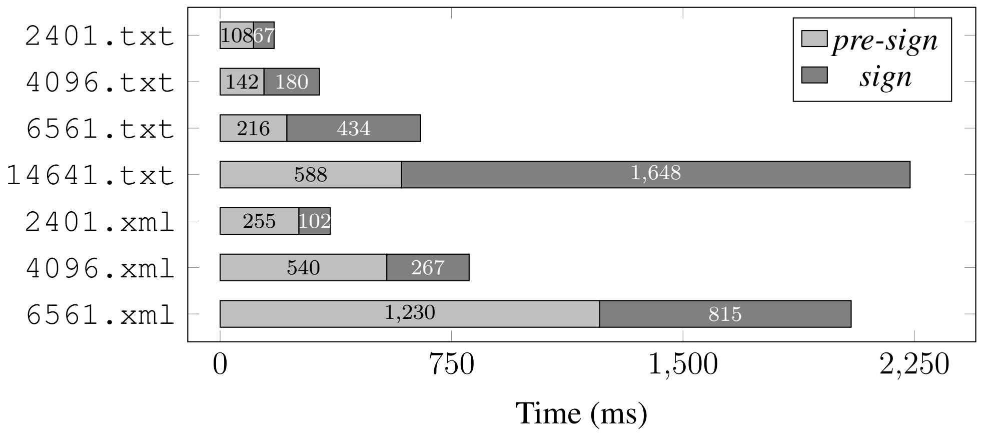


Figure 3. Performance of the signature generation procedure stages for MTSS($\Sigma, \mathcal{H}, \mathcal{M}$), where $\mathcal{M}$ changes according to each file.

4.3. Verifying the signature and locating errors

Analogously, we divide the signature verification procedure into *pre-verify* and *verify*. The former handles I/O operations by reading the message m, signature σ, and public key pk from the disk; the latter follows the MTSS Vᴇʀ algorithm. In the framework, signature verification performance depends on whether the signed message was modified or not. If its integrity is preserved, the efficiency of Vᴇʀ is comparable to Σ.Vᴇʀ, albeit with the

Table 2. Performance of signature operations of MTSS($\Sigma, \mathcal{H}, \mathcal{M}$) for several choices of $\mathcal{M}$, plain text files of different sizes, and $|I| = 1$.

Parameters of $\mathcal{M}$					Size (kB)		Time (ms)	
k	q	t	n	d	m	σ	SIG	VER
4	7	49	2401	2	4.69	3.38	56.98	33.41
4	9	81	6561	2	12.8	5.38	156.27	185.95
4	11	121	14641	3	28.6	7.88	471.47	777.91
4	13	169	28561	4	55.8	10.9	1318.92	2609.34
-	-	15	4096	1	8	1.25	96.21	88.41
4	8	64		2		4.31	91.30	82.93
3	16	256		7		16.3	187.37	191.41
2	64	4096		63		256	1720.94	2238.13

overhead of the MTSS signature being larger (cf. $|\sigma|$ in Table 1). Therefore, we focus on the performance of locating modifications in signed messages.

We discuss the influence of the number of tests t in modified signed messages in Table 2. We infer that the verification time is directly proportional to t. In order to become practical, fewer tests are desired; nevertheless, depending on the framework's application, more tests and a higher number of modifications detected are expected. This tradeoff is necessary for the more powerful signatures MTSS produces. Another experiment we conducted was changing the number of modified characters $|I|$ in the same document from the bottom part of Table 2, using $k = 3$, i.e., generating a 7-CFF$(256, 4096)$; we noted that the performance for different $|I|$ is very small, since step 4 from VER recognizes all modified blocks in a loop, not stopping until reaches all tests.

Table 3 shows the performance overhead of MTSS.VER with $|I| = 0$ and $|I| = 1$ against verifying using only Σ.VER. We remark that locating errors increases the verifying time by up to $2\times$, depending on the number of tests t. However, we still consider it as efficient and practical in real scenarios. This comparison also addresses $\mathbf{Q_3}$ and $\mathbf{Q_4}$ in the context of signature verification.

4.4. Correcting

The VCOR algorithm supersets VER: modified blocks are first located, and then corrections are tried. As mentioned in Sect. 2.2, we need to brute-force all possibilities from some block considering some character encoding. Aspects such as the number of blocks, number of modifications, content to be corrected, and the cryptographic hash function used affect the performance of the algorithm. Particularly, the chosen $\mathcal{H}$ has a greater impact on correcting modified blocks since a block with s bits requires 2^s hash calculations. We note that our implementation has already been performed using multiprocessing.

In our experiment, we varied the parameter $|I|$ and s, using configurations $\Sigma =$ RSA-2048, $\mathcal{H} =$ BLAKE2b, and $\mathcal{M}$ as 7-CFF$(126, 4096)$ and $k = 3$. We observed that the overall performance for locating and correcting errors is linearly proportional to $|I|$. Increasing $|I|$ linearly scales the time, so doubling $|I|$ would double the time required. From that, we can infer that the time complexity T of the algorithm is $T(|I|, s) = \mathcal{O}(|I| \times f(s))$,

Table 3. Performance and output size of MTSS($\Sigma, \mathcal{H}, \mathcal{M}$) VER for several choices of Σ, $\mathcal{H}$ and security parameter λ, considering $\mathcal{M} = 2$-CFF($25, 125$), $k = 3$ and a 1.19MB plain text file as input.

	λ	Σ	VER time (ms)					
			SHA-2		SHA-3		BLAKE	
			256	512	256	512	2s	2b
Raw Σ	128	RSA-2048	4.09	2.87	3.15	5.66	3.21	4.84
	128	ML-DSA-44	3.88	2.61	2.90	5.47	1.51	2.30
	128	Ed25519		3.84				
	256	RSA-4096	4.37	3.17	3.44	5.94	3.50	5.12
	256	ML-DSA-65	3.91	2.62	2.92	5.47	1.51	2.32
	512	ML-DSA-87	3.91	2.64	2.94	5.49	1.53	2.34
MTSS ($\lvert I\rvert = 0$)	128	RSA-2048	4.16	2.95	3.25	5.80	1.81	2.64
	128	ML-DSA-44	3.94	2.69	3.00	5.55	1.55	2.37
	128	Ed25519		3.90				
	256	RSA-4096	4.45	3.23	3.54	6.08	2.10	2.91
	256	ML-DSA-65	3.95	2.70	3.08	5.58	1.56	2.38
	512	ML-DSA-87	3.95	2.72	3.05	5.57	1.58	2.40
MTSS ($\lvert I\rvert = 1$)	128	RSA-2048	159.95	174.53	162.22	162.29	153.15	162.94
	128	ML-DSA-44	156.93	177.55	159.49	165.35	163.51	154.37
	128	Ed25519		165.41				
	256	RSA-4096	169.32	153.97	166.26	161.66	167.44	160.91
	256	ML-DSA-65	160.05	191.31	165.23	162.47	154.52	160.33
	512	ML-DSA-87	158.34	162.69	167.37	158.83	162.81	152.57

where $f(s)$ represents the super-linear growth rate concerning s. Some performance examples are given next, for $|I| = 1$: i) for $s = 1$, the time was 50ms; ii) for $s = 2$, the time increased to 200ms; iii) For $s = 3$, the time significantly increased to 130,000ms; iv) we estimated for $s = 5$, it would take ≈ 80.2 days.

5. Ensuring data integrity of individual blocks

We now consider a novel approach to the MTSS framework, framed as another question.

$\mathbf{Q_5}$ *Is it possible to verify the integrity and authenticity of only one block of the original signed data without having access to the whole data?*

We were inspired by big data applications where a large signed dataset is stored on a server, and the challenge is to verify the integrity of a small portion of this data without downloading the entire set. We aim to verify whether a given small portion of data belongs to the original dataset and ensure its authenticity. For instance, we want to verify that a single page from a large signed PDF document belongs to the whole document and check its integrity without accessing the entire file. This is relevant to documents in the Brazilian Federal Register (Diário Oficial da União), where each page currently needs a separate digital signature. The approach we propose in this section allows the entire document to be signed once, enabling verification of each page's integrity and authenticity without multiple signatures. Here, we solely focus on integrity and authenticity verification, and we do not explore applications of this scheme for privacy protection or correction of modifications.

We positively answer the question by defining a protocol between two parties: the server S, which stores the original document m, its corresponding MTSS signature σ, and the system parameters of the framework, and the client C, which requests whether some block m_j belongs to the original document m, i.e., $m_j \in m$.

The server S stores a tuple $Z = (m, \mathcal{H}(m), \sigma, \Sigma, \mathcal{H}, n)$, where $m = (m_1, \dots, m_n)$ is the document already split into n blocks; $\mathcal{H}(m)$ is the cryptographic hash of m; $\sigma = (\sigma', T)$ is a MTSS signature, and $\Sigma, \mathcal{H}, n$ are MTSS parameters as shown in Sect. 2.2. Without loss of generality, S uses $\mathcal{H}(m)$ as a unique index to identify the corresponding message m. We assume C holds $\mathcal{H}(m)$, made publicly available by S. We recall that S can reconstruct the d-CFF $\mathcal{M}$, given the necessary parameters. First, we provide an auxiliary algorithm that allows for the verification of the integrity and authenticity of a single block.

BLOCKVER(m_j, pk, Y). Let m_j be a block of a message m, pk a public key of Σ, and $Y = (\sigma, i, M, k)$, where $\sigma = (\sigma', T)$ is a MTSS signature, $i \in \mathbb{N}$ is an index such that T_i is a hash of the MTSS signature in which block m_j appears, $M = (m_\ell \in m : \mathcal{M}_{i,\ell} = 1, \ell \neq j)$, and k is the index of $m_j \in T_i$. The algorithm proceeds as follows.

1. Set $r \leftarrow \Sigma.\text{VER}(\sigma', T, \mathsf{pk})$. If $r = \perp$, output $\perp$. Otherwise, go to step 2.
2. Set $M \leftarrow (M_1, \dots, M_{k-1}, m_j, M_k, \dots)$.
3. For $1 \leq x \leq |M|$, set $h_x \leftarrow \mathcal{H}(m_x)$.
4. Set $h' \leftarrow \mathcal{H}(h_1 || h_2 || \cdots || h_{|M|})$.
5. If $h' = T_i$, output $\top$. Otherwise, output $\perp$.

Then, we propose an iterative protocol between C and S as follows:

1. C sends a tuple $X = (\mathcal{H}(m), j)$ to S, where j is the index of the desired block; S keeps this information in memory until C ends the protocol or step 5 is reached.
2. S sends $Y = (\sigma, i, M, k)$ to C; the contents of Y are described in BLOCKVER.
3. C sets $r \leftarrow \text{BLOCKVER}(m_j, \mathsf{pk}, Y)$.
4. If $r = \perp$, C sends i to S, so it decides on a new i and goes back to step 2. Otherwise, C sends $\top$ to S to end the protocol.
5. If S has no more item i, this terminate the protocol.

Now, we discuss how the protocol and algorithm solve the proposed problem. First, note that even though we don't need the entire message m to perform the verification, we still need some extra blocks from m, i.e., the ones concatenated to m_j in hash T_i, denoted by the tuple M.

The first step in BLOCKVER ensures that the set T of hashes from σ is authentic and can be used to verify m_j. The next steps recreate the particular hash h' using both m_j and the other blocks from M. Finally, in step 5, we compare T_i, which is the i-th hash of the signature, with the computed hash h'. If they match, it means that m_j is authentic and belongs to the expected original message m. Otherwise, we have two possibilities: i) m_j does not belong to m or has integrity issues; ii) some other $m_s \in M$ has integrity issues and is the reason why step 5 failed.

We can not be sure about i) since we always depend on other blocks to perform the verification. However, we can still surpass case ii) for up to d invalid accompanying blocks m_s since σ comes from a d-CFF $\mathcal{M}$. Note that $\mathcal{M}$ has several other rows i for which $\mathcal{M}_{i,j} = 1$, so we can ask the server for the next index i and the corresponding set of accompanying blocks M. For each new set of parameters, we can perform BLOCKVER again. If step 5 outputs $\top$ for one of them, we know m_j is authentic and belongs to m.

We observe that the number $|M|$ of extra blocks necessary for the verification depends on the number of 1s per row in $\mathcal{M}$. For instance, if $\mathcal{M}$ came from the polynomial construction, we have $|M| = q^{k-1}$. In this case, we are interested in constructions that minimize the 1s per row of $\mathcal{M}$, which is an open problem, as mentioned in [Idalino and Moura 2024]. Assuming no storage limitations, the server can minimize t in exchange for larger signatures.

Also, restarting the protocol with the next index i and tuple M in case BLOCKVER outputs $\perp$ can be performed several times, which is equal to the number of 1's in column j of $\mathcal{M}$. For the polynomial construction, this value equals q; for the Sperner construction, it equals $\lfloor \frac{t}{2} \rfloor$. Finally, our solution uses the same signature algorithm of the MTSS scheme, but we execute a partial verification instead of the one from MTSS. We claim that our protocol signature is secure under the same assumptions proved in [Idalino et al. 2019].

Moreover, our proposal assumes that C has access to the block index j, which may not be practical in real-world applications. A more feasible approach would involve C sending a tuple (h_m, m_j) to S, or, to optimize network resources, $(h_m, \mathcal{H}(m_j))$. Given that S is a resource-rich server in terms of processing and storage, it would be possible to create structural data that efficiently correlate $\mathcal{H}(m_j)$ with tests T_i in $\mathcal{M}$, allowing S to send the tuple Y back to C. Although this would require additional algorithms and storage, it would simplify the process for C, which would only need to have h_m and its block m_j.

6. Conclusion

We implement the modification-tolerant signature scheme (MTSS) framework, first introduced by [Idalino et al. 2019], in a high-level programming language. With that, we test its overall performance for its different algorithms, such as signing, verifying, and locating errors and correcting them. We demonstrated how choosing a traditional signature scheme Σ and a hash function $\mathcal{H}$ can affect the performance of its algorithms. Additionally, we showed different arguments to give ideal parameters for constructing CFFs, depending on how many modified blocks d are desired.

We also showed how difficult it is to separate different digital documents into blocks so we can locate changes in them afterward; complex structure files like XML take more time to parse and separate into blocks. We analyzed the performance details using different parameters in our implementation of Sperner sets and polynomial constructions; this led us to take some techniques to soften CFF construction, such as caching. Finally, we present a novel approach to using MTSS, where we ensure partial data integrity and authenticity of a single block without access to the whole signed message m, using an iterative protocol between a client C and a server S.

We leave as a future work incorporating efficient implementations of CFFs with $d \geq 2$ within MTSS. Also, it is important to explore how to properly divide more complex and common files into blocks, such as PDF or hierarchical documents in different data structures. A general performance overview of our new protocol over the MTSS framework would be advisable, validating it in a real-life example.

References

Bernstein, D. J., Duif, N., Lange, T., Schwabe, P., and Yang, B.-Y. (2012). High-speed high-security signatures. *Journal of cryptographic engineering*, 2(2):77–89.

Bilzhause, A., Pöhls, H. C., and Samelin, K. (2017). Position paper: the past, present, and future of sanitizable and redactable signatures. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*, pages 1–9.

Bshouty, N. H. (2015). Linear Time Constructions of Some d-Restriction Problems. In *Algorithms and Complexity. CIAC 2015. Lecture Notes in Computer Science*, volume 9079, pages 74–88.

Cao, Y.-y. and Fu, C. (2008). An efficient implementation of rsa digital signature algorithm. In *2008 International Conference on Intelligent Computation Technology and Automation (ICICTA)*, volume 2, pages 100–103.

De Bonis, A. and Di Crescenzo, G. (2011a). Combinatorial group testing for corruption localizing hashing. In *Computing and Combinatorics: 17th Annual International Conference, COCOON 2011, Dallas, TX, USA, August 14-16, 2011. Proceedings 17*, pages 579–591. Springer.

De Bonis, A. and Di Crescenzo, G. (2011b). A group testing approach to improved corruption localizing hashing. *Cryptology ePrint Archive*.

Di Crescenzo, G., Ge, R., and Arce, G. R. (2004). Design and analysis of dbmac, an error localizing message authentication code. In *IEEE Global Telecommunications Conference, 2004. GLOBECOM'04.*, volume 4, pages 2224–2228. IEEE.

Erdös, P., Frankl, P., and Füredi, Z. (1985). Families of finite sets in which no set is covered by the union of r others. *Israel Journal of Mathematics*, 51(1):79–89.

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

Gargano, L., Rescigno, A. A., and Vaccaro, U. (2020). Low-weight superimposed codes and related combinatorial structures: Bounds and applications. *Theoret. Comput. Sci.*, 806:655–672.

Goodrich, M. T., Atallah, M. J., and Tamassia, R. (2005a). Indexing information for data forensics. In *Applied Cryptography and Network Security: Third International Conference, ACNS 2005, New York, NY, USA, June 7-10, 2005. Proceedings 3*, pages 206–221. Springer.

Goodrich, M. T., Atallah, M. J., and Tamassia, R. (2005b). Indexing information for data forensics. In *Applied Cryptography and Network Security: Third International Conference, ACNS 2005, New York, NY, USA, June 7-10, 2005. Proceedings 3*, pages 206–221. Springer.

Haber, S., Hatano, Y., Honda, Y., Horne, W., Miyazaki, K., Sander, T., Tezoku, S., and Yao, D. (2008). Efficient signature schemes supporting redaction, pseudonymization, and data deidentification. In *Proceedings of the 2008 ACM symposium on Information, computer and communications security*, pages 353–362.

Idalino, T. B. and Moura, L. (2024). A survey of cover-free families: Constructions, applications, and generalizations. In Colbourn, C. J. and Dinitz, J. H., editors, *New*

Advances in Designs, Codes and Cryptography, pages 195–239, Cham. Springer Nature Switzerland.

Idalino, T. B., Moura, L., and Adams, C. (2019). Modification tolerant signature schemes: location and correction. In *International Conference on Cryptology in India*, pages 23–44. Springer.

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.

Johnson, R., Molnar, D., Song, D., and Wagner, D. (2002). Homomorphic signature schemes. In *Cryptographers' track at the RSA conference*, pages 244–262. Springer.

Josefsson, S. and Liusvaara, I. (2017). Edwards-Curve Digital Signature Algorithm (EdDSA). RFC 8032.

Lim, S. and Lee, H.-S. (2011). A short and efficient redactable signature based on rsa. *ETRI Journal*, 33(4):621–628.

Mlynkova, I., Toman, K., and Pokornỳ, J. (2006). Statistical analysis of real xml data collections. In *COMAD*, volume 6, pages 20–31.

of Standards, N. I. and Technology (2023). Module-lattice-based digital signature standard. Technical Report Federal Information Processing Standards Publications (FIPS PUBS) 204 (Draft), August 24, 2023, U.S. Department of Commerce, Washington, D.C.

Pöhls, H. C. (2018). *Increasing the Legal Probative Value of Cryptographically Private Malleable Signatures*. PhD thesis, Universität Passau.

Porat, E. and Rothschild, A. (2011). Explicit nonadaptive combinatorial group testing schemes. *IEEE Trans. Inf. Theory*, 57(12):7982–7989.

Rescigno, A. A. and Vaccaro, U. (2023). Bounds and Algorithms for Generalized Superimposed Codes Bounds and Algorithms. *Inform. Process. Lett.*, 182.

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.

Sperner, E. (1928). Ein Satz über Untermengen einer endlichen Menge. *Mathematische Zeitschrift*, 27:544–548.

Steinfeld, R., Bull, L., and Zheng, Y. (2002). Content extraction signatures. In *Information Security and Cryptology—ICISC 2001: 4th International Conference Seoul, Korea, December 6–7, 2001 Proceedings 4*, pages 285–304. Springer.

W3C (2008). Canonical xml version.

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