Journal of the Brazilian Computer Society, 2026, 32:1, doi: 10.5753/jbcs.2026.5565
This work is licensed under a Creative Commons Attribution 4.0 International License.

Partial integrity, authenticity and belongingness using modification-tolerant signature schemes

Anthony Bernardo Kamers [**Federal University of Santa Catarina** | ***anthony.kamers@posgrad.ufsc.br***]
Gustavo Zambonin [**Federal University of Santa Catarina** | ***gustavo.zambonin@posgrad.ufsc.br***]
Thaís Bardini Idalino [**Federal University of Santa Catarina** | ***thais.bardini@ufsc.br***]
Paola de Oliveira Abel [**Federal University of Santa Catarina** | ***paola.abel@grad.ufsc.br***]
Jean Everson Martina [**Federal University of Santa Catarina** | ***jean.martina@ufsc.br***]

Departamento de Informática e Estatística, Universidade Federal de Santa Catarina (UFSC), R. Eng. Agronômico Andrei Cristian Ferreira, s/n, Trindade, Florianópolis, SC, 88040-900, Brazil.

Received: 15 February 2025 • **Accepted:** 20 June 2025 • **Published:** 16 March 2026

Abstract. Digital signatures allow us to ensure that the signed digital data is authentic and has not been modified. However, even a single bit modification in the data invalidates the entire signature. In INDOCRYPT '19, Idalino et al. presented an efficient modification-tolerant signature scheme (MTSS) framework using combinatorial group testing techniques, allowing the location and correction of modified parts of the signed data. In this work, we implement their framework and discuss the practical performance of the solution. We also propose various necessary auxiliary algorithms not explored in the initial work, such as the division of data into blocks and the generation of the underlying combinatorial structure needed for the signature generation. Moreover, we propose a novel use case of the framework, which we call the *belongingness framework*. This scheme allows the verification of the integrity and authenticity of a subset of the signed data without having access to the whole data. This is particularly interesting in big data applications, where access to the whole signed data is prohibitive due to storage limitations.

Keywords: Digital signatures, cryptography, modification-tolerant signatures, partial integrity, big data

1 Introduction

A traditional digital signature scheme allows a user to verify the integrity and authenticity of digitally signed data. The signature verification algorithm has a boolean output, which allows for the *detection* of modifications; if a single bit of a digitally signed document is modified, the respective digital signature is considered invalid. In practical terms, the signer and/or the information contained within the document may not be trusted.

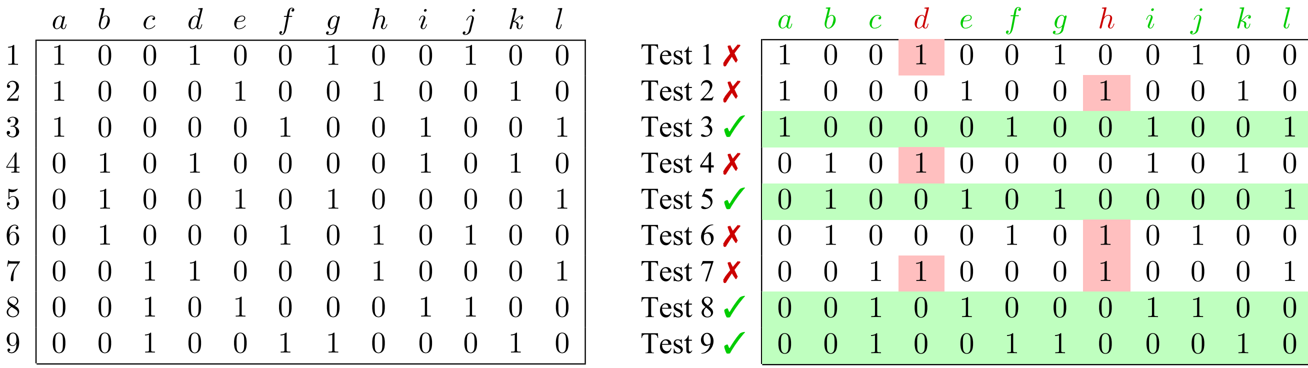
However, if additional properties such as *location* and *correction* of possible modifications are considered, digital signatures can be employed in a variety of new scenarios. Idalino *et al*. [2015] cites several use cases:

(i) modification discovery in fillable forms, allowing one to sign an 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*., 2005];
(iii) improving the efficiency of computer systems: for instance, by pinpointing where a large database was modified, without invalidating the entire database;
(iv) privacy protection, where parts of a signed document can be intentionally redacted without invalidating the original signature (e.g., content extraction [Steinfeld *et al*., 2001] 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,b] in the context of hash functions, by di Crescenzo *et al*. [2004]; Goodrich *et al*. [2005] in the context of message authentication codes, and by Idalino *et al*. [2015, 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 to be signed is split into blocks, and these blocks are combined and signed according to a CFF. 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 even possible to recover the original signed document, depending on the parameter choices of the scheme.



	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. Adapted from [Idalino and Moura, 2022].

Contributions. We address some open questions of the proposal as laid out by 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 suggest parameters 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. Our contributions focus on data authenticity and integrity; we do not explore privacy applications, such as redactable and sanitizable signatures.

In the context of *partial* data integrity and authenticity, we also explore the possibility of guaranteeing such properties in a *subset* of the signed data. We make clear definitions for this use case and name it *belongingness framework*. This is especially interesting in the context of big data, where the verifier may not have access to the full data to perform the verification of the digital signature. This subject has been somewhat explored in the literature using different techniques [Goyal and Vaikuntanathan, 2022; Lu *et al*., 2024; Yan *et al*., 2023]. We propose a variation of MTSS that can be used to verify the authenticity and integrity of some parts of signed data without having access to the entire data. Then, we instantiate our belongingness framework using MTSS signatures.

This work is an extended version of [Kamers *et al*., 2024], particularly in the following aspects:

- (i) we consider a much larger variety of document types and corresponding block division strategies;
- (ii) we improve the CFF construction to ensure compatibility with documents with an arbitrary number of blocks;
- (iii) we improve the MTSS signature procedure by decreasing the number of hash concatenations required;
- (iv) we perform new experiments considering the influence of different CFFs on MTSS signature operations;
- (v) we give a definition of the subset verification problem (without access to the entire signed data), namely, the belongingness framework;
- (vi) we give an overview of possible instantiations of the belongingness framework, implement the proposed protocol using MTSS, and present a more detailed analysis of its performance.

Organization. In Section 2, we briefly give the necessary background to understand MTSS and our contributions. In Section 3, we give technical details about the practical implementation of MTSS, such as document division into blocks and construction of suitable CFFs. In Section 4, we discuss the selection of parameters and other implementation choices for MTSS; we also provide several performance and storage measurements. In Section 5, we introduce the belongingness framework, we define the framework and its requirements, and then we instantiate the protocol with a variation of MTSS. Finally, we compare this implementation with other suitable candidates in the literature. Finally, in Section 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 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 [Idalino and Moura, 2022]) *Let $d, t, n \in \mathbb{N}$, 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 $B_{i_1}, \ldots, B_{i_d}$ in $\mathcal{B}$ we have that B_{i_0} is not contained in the union $B_{i_1} \cup B_{i_2} \cup \ldots \cup B_{i_d}$.*

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 more detail, rows are indexed by $x_i \in X$ and each column j is the incidence vector of the respective B_j. In the literature, this matrix representation is also known as a d-disjunct matrix [Du and Hwang, 2000]. 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 $t = 9, n = 12$, $X = \{1, \ldots, 9\}$, $\mathcal{B} = \{B_a, B_b, \ldots, B_l\}$, where $B_a = \{1, 2, 3\}$, $B_b = \{4, 5, 6\}$, …, $B_l = \{3, 5, 7\}$. In this example, all blocks (columns)

have a cardinality of 3, which is a consequence of the specific d-CFF construction used here (via Proposition 1), and not a general requirement.

Combinatorial group testing (CGT) is a testing technique that aims at identifying up to d defective items from a pool of n items. Instead of testing each item individually (performing n tests), the idea consists of combining items into groups and testing the groups instead. By doing so, we save on tests and resources while being able to identify up to d defects.

Notably, a d-CFF(t, n) can be used in CGT. In this context, each column of $\mathcal{M}$ corresponds to an item being tested, and each row of $\mathcal{M}$ represents a group of items that will be tested together. In other words, if $\mathcal{M}_{i,j} = 1$, then the j-th item belongs to the i-th group of items, which will be tested in the i-th test. The d-CFF property allows us to perform only t tests (one per row of $\mathcal{M}$) and easily identify up to d defects among the n items in the following way: groups that pass the test contain only non-defective items; remaining items are considered defective.

By using a d-CFF in a CGT scenario, we have that the union of d columns does not cover any other column, which guarantees that for any combination of up to d defective items, the non-defective ones can always be identified by the groups that pass the test. For more details on algorithms for group testing and results on CFFs, we refer the reader to [Du and Hwang, 2000; Idalino and Moura, 2022; Wei, 2006].

Figure 1b shows the procedure with $t = 9$ tests (rows), which are sufficient to test $n = 12$ items (columns a to l). The figure also shows how we can identify two defective items using only 9 tests of the 2-CFF$(9, 12)$. We assume in this example that items d and h (in red) are defective and, therefore, the groups of items represented by rows $1, 2, 4, 6$, and 7 will fail their corresponding tests. However, 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, items d and h). The fact that this is a 2-CFF guarantees that any 2 defective items can be identified. If there are more than d defective items, some non-defective items might be erroneously considered defective due to the CGT technique.

In a CGT context, given the number n of items to be tested and the upper bound d on defective items, we are interested in performing the smallest possible number of tests t while being able to identify up to d defects. In the context of CFFs, given values n and d, we aim to build d-CFFs that minimize the number of rows t. This objective has been central to the study of bounds and constructions for d-CFFs. For $d = 1$, an optimal construction is given by Sperner set systems [Sperner, 1928].

Construction 1 (1-CFF(t, n) [Sperner, 1928]) *Let $n \in \mathbb{N}$ and $t = \min\{s : \binom{s}{\lfloor \frac{s}{2} \rfloor} \geq n\}$. Consider $X = \{1, 2, \ldots, t\}$ and take $\mathcal{B}$ to be the collection of all distinct subsets of X of size $\lfloor \frac{t}{2} \rfloor$.*

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 constructive 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. For a finite field $\mathbb{F}_q$, consider $\mathbb{F}_q[x]_{<k}$ as the set of all polynomials with degrees less than k and coefficients in $\mathbb{F}_q$.

Proposition 1 (d-CFF(t, n) [Erdős *et al.*, 1985]) *Let $n \in \mathbb{N}$, q be a prime power, $2 \leq k \leq q$, $d \leq \lfloor \frac{q-1}{k-1} \rfloor$, and $\mathbb{F}_q$ a finite field with q elements. 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 $\mathcal{B} = \{B_p : p \in \mathbb{F}_q[x]_{<k}\}$. Thus (X, B) is a d-CFF(q^2, q^k).*

For more details on cover-free families, applications, and other constructions for the case $d \geq 2$, we refer the reader to the survey of Idalino and Moura [2022].

2.2 The MTSS framework

The MTSS framework is able to locate d modifications in a signed document using a d-CFF(t, n). During the signature generation, a to-be-signed document must be split into n blocks of s bytes to fit the underlying CFF. For blocks of small enough size, the authors also propose correcting such modifications. The signature must carry extra information to provide the location of modified blocks of text. This extra information is given in the form of t hashes, which are computed according to the rows of the d-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 traditional digital signature scheme, with the usual key generation (Σ.KeyGen), signature generation (Σ.Sig), and signature verification (Σ.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 $\mathrm{MTSS}(\Sigma, \mathcal{H}, \mathcal{M})$. MTSS has four algorithms: KeyGen, Sig, Ver, and "verify-and-correct" (VCor); a brief description of each one is given next. We also define the set of modified blocks identified by the framework as I. Let $\top$ be a valid output, and $\perp$ otherwise; when applied to verification, $\perp$ means a verification failure.

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

1. Set $(\mathsf{sk}, \mathsf{pk}) \leftarrow \Sigma.\mathrm{KeyGen}(\lambda)$.
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 c_i to be the concatenation of all m_j such that $\mathcal{M}_{i,j} = 1$, and $T_i \leftarrow \mathcal{H}(c_i)$.
2. Set $h_m \leftarrow \mathcal{H}(m)$.
3. Set $T \leftarrow (T_1, \ldots, T_t, h_m)$.
4. Set $\sigma' \leftarrow \Sigma.\mathrm{Sig}(T, \mathsf{sk})$.
5. Output (σ', T).

$\text{Ver}(m, \sigma, \mathsf{pk})$. Let $m = (m_1, \dots, m_n)$ be any message divided into n blocks, $\sigma = (\sigma', T)$ an MTSS signature, with $T = (T_1, \dots, T_t, h_m)$, and pk the corresponding public key of Σ. The algorithm proceeds as follows.

1. Set $r \leftarrow \Sigma.\text{Ver}(T, \sigma', \mathsf{pk})$. If $r = \perp$, output $(\perp, \{\})$. Otherwise, go to Step 2.
2. If $\mathcal{H}(m) = h_m$, output $(\top, \{\})$.
3. For $1 \leq i \leq t$, compute $\widetilde{T}_i$ as in Step 1 of Sig.
4. Set $V \leftarrow \{\}$. For $1 \leq i \leq t$, if $T_i = \widetilde{T}_i$, set $V \leftarrow V \cup \{j : \mathcal{M}_{i,j} = 1\}$.
5. Set $I \leftarrow \{1, \dots, n\} \setminus V$. If $|I| \leq d$, output $(\top, I)$. Otherwise, output $(\perp, I)$.

$\text{VCor}(m, \sigma, \mathsf{pk})$. Let $m = (m_1, \dots, m_n)$ be any message divided into n blocks, $\sigma = (\sigma', T)$ an MTSS signature, and pk a public key of Σ. The algorithm proceeds as follows.

1. Set $(r, I) \leftarrow \text{Ver}(m, \sigma, \mathsf{pk})$. If $r = \perp$, output $(\perp, \{\}, \{\})$.
2. If $|I| = 0$, go to Step 7. Otherwise, set $b \leftarrow \min(I)$, $I \leftarrow I \setminus b$, and proceed.
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 $c_b = \perp$.
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 $\widetilde{T}_i$ as in Step 1 of Sig.
   (c) If $\widetilde{T}_i \neq T_i$, continue to the next q.
   (d) If $c_b = \perp$, set $c_b \leftarrow \top$ and $m_b \leftarrow q$; continue to the next q.
   (e) Output $(\top, I, \varepsilon)$, where ε is the empty string.
6. Go to Step 2.
7. Output $(\top, I, m)$.

We observe that, given only a signature σ and the number of blocks n, one can reconstruct $\mathcal{M}$. This may be used for implementation purposes. 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 highlight that constructing a proof of security under an adversarial model, such as an adaptive chosen-message adversary, is outside the scope of this work; moreover, our work hereafter concentrates its efforts on the location and correction of modifications on a signed document, not exploring redactable signatures, which were also proposed in [Idalino *et al.*, 2019].

Remark 1 *In contrast to Kamers* et al. *[2024], Step 1 of* Sig *is slightly modified. Particularly, no blocks m_j are hashed prior to their concatenation in c_i, as they do not need to be uniquely identified. Consequently, the number of hashes calculated in* Sig, Ver, *and* VCor *is reduced by a factor of n. This was independently realized by Luo [2024], and the author shows that this modification does not compromise the security of MTSS. We briefly comment about this performance improvement in Section 4.2.*

In the following section, 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 on overall performance and signature sizes.

3 Discussion on MTSS parameters

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 MTSS algorithms except KeyGen expect to receive a message m already separated into 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 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 $m = (m_1, \dots, m_n)$. Our implementation does not consider n as a free input because of possible issues in locating errors using the 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 Section 2.1, a CFF is defined via parameters d, t, and n; our goal is to minimize t, so we have smaller CFFs. We recall that n is the number of blocks of a message m' given by DivideBlocks, as discussed in $\mathbf{Q_1}$. The user is expected to choose d, controlling how many modifications can be located after the signature. Hence, we define the CreateCFF algorithm as follows: if $d = 1$, the Sperner construction (cf. Construction 1) is used; otherwise, we use the polynomial over finite fields construction, which considers parameters q, k, n (cf. Proposition 1). The output of CreateCFF 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 algorithms Sig and Ver require $t + 1$ evaluations of the hash function $\mathcal{H}$. Thus, the choice of the underlying cryptographic hash function greatly influences the performance of operations on signatures. We evaluate several choices of $\mathcal{H}$, discuss the necessity of these hash calculations, and give performance results and recommendations in Section 4.2.

$\mathbf{Q_4}$ How does the choice of Σ impact overall performance?

We observe that Idalino *et al.* [2019] roughly estimates the performance of MTSS depending on Σ. We give actual performance and signature size results in Section 4.2.

3.1 Document types 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. Idalino *et al*. [2019]; Pöhls [2018] consider strategies such as sequential ordering considering some delimiter, non-sequential ordering using complex structural data, a header in the file informing the blocks, or 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]. 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 all subsequent blocks. To prevent this problem, we must define document representations and block delimiters for DivideBlocks. We note that each document type has its own specificity.

In other words, we must define rules to divide a to-be-signed document that: (i) consumes little computational resources; (ii) avoid attacks that explore block division to invalidate the whole signature; (iii) prevents the cascade effect. Another problem is how we represent a block after we have parsed the document. A block must clearly represent a part of the document; blocks should be simple and effective, making it easy to compute hashes of their content. We abstract a block as a record data structure able to represent the contents of different file formats homogeneously.

Definition 2 (Block record) *A block is a record composed of the following fields:*

(i) an identifier*, represented by a string, containing the block name;*
(ii) a level*, represented by a non-negative integer, containing the hierarchical level of the block relative to the original file;*
(iii) the contents*, represented by a string, containing the actual data;*
(iv) and attributes*, represented by an associative array of strings to strings, representing additional information about the block.*

A null object is defined as ε. An empty block is defined by the symbol $\square$. All fields are optional, but each document type uses its configuration. We output the given parameters separated by a vertical bar (U+007C) as the block representation. Our considerations include documents of the type plain text, CSV, XML, JSON, PDF, and images (PGM, BMP, and PNG). We will explain them in different categories: block delimiters, hierarchical files, and fixed-size blocks. Hereafter, we demonstrate effective ways to separate different file types; in Figure 6 and Figure 7, we give examples of Block records for XML and JSON documents.

3.1.1 Block delimiters

We define block delimiters for plain text and CSV files. For plain text, we set the line break character (U+000A) as the default delimiter. Hence, the number of lines must remain the same to allow the location of modifications, but any individual lines are modifiable. With this strategy, content can be added or removed within a line, but creating or deleting lines will cause a cascade effect on locating errors with MTSS.

When considering CSV files, we define three possible delimiters: line breaks (hereafter CSV-b), commas (U+002C, hereafter CSV-c), or semicolons (U+003B). The choice depends on how the document is structured. For instance, a row of a CSV file is composed of cells, and a line break separates each row; each cell is separated by either a comma or a semicolon. Simply put, if we choose to delimit blocks by line breaks, each block will correspond to a row of the file; otherwise, to a cell.

By dividing the rows into blocks, we can detect if more cells were added to a line, considering that the number of lines remains the same. By contrast, if we consider the cells as blocks, we will be able to locate modifications with higher granularity; however, this strategy will work only in cases where modifications were made to already existing cells. Modifications in which new cells are added or cells are removed will cause a cascade effect. Although the second alternative gives a more precise location capability, dividing into rows is a more generic approach that tells us that at least one cell in that row was modified.

3.1.2 Hierarchical files

Some file formats have their structure organized as a tree, where child nodes inherit the properties and attributes of their parent node. Usually, such files are meant to be human-readable, facilitating the understanding of the whole document. In this context, we address the XML, JSON, and PDF file formats, but our approach can be adapted to other hierarchical file formats.

The block record of Definition 2 is precisely constructed to support such tree structures. Without loss of generality, we scan the document tree from top to bottom and separate each element into blocks following a *preorder* traversal. We build blocks with the properties and attributes presented in the element and their level in the corresponding tree.

However, depending on the file format, an element can be represented by different data structures. For instance, each key in a JSON file can be mapped to a string, a sequence, an associative array, etc. To clarify our proposed block division for hierarchical files, we present the DivideBlocksHier and DivBlocksHierRec algorithms.

DivBlocksHierRec(e, ℓ, b, m, P). Let e be an element of the hierarchical file, $\ell \in \mathbb{N}$ be the current depth of recursion (on the hierarchical content), b a block (cf. Definition 2), m a sequence of blocks (our final output), and P a set of blocks. The algorithm proceeds as follows.

1. If $e \in P$, return $\perp$. Otherwise, go to Step 2.
2. Set $P \leftarrow P \cup e$.
3. If e is represented as an associative array, go to Step 4. Otherwise, go to Step 5.
4. For all key-value pairs $(k, v) \in e$:
   (a) Initialize a block b_k with $(k, \ell, \varepsilon, v.\mathtt{attributes})$.
   (b) Set $m \leftarrow m \mid\mid b_k$.
   (c) Call DivBlocksHierRec$(v, \ell + 1, b_k, m, P)$.

5. If e is represented as a sequence, go to Step 6. Otherwise, go to Step 7.
6. For all $i \in e$, call $\text{DivBlocksHierRec}(i, \ell + 1, b, m, P)$.
7. If e is represented as a primitive type, set $b.\mathsf{contents} \leftarrow e$.
8. Return $\perp$.

$\text{DivideBlocksHier}(m')$. Let m' be a hierarchical file. We assume the existence of a $\text{GetRoot}(m')$ procedure that returns the root element of m', according to the syntax of its file format. The algorithm proceeds as follows.

1. Set $r \leftarrow \text{GetRoot}(m')$.
2. Set $b \leftarrow \square$.
3. Set $m \leftarrow ()$.
4. Set $P \leftarrow \{\}$.
5. Call $\text{DivBlocksHierRec}(r, 1, b, m, P)$.
6. Return $(m, |m|)$.

DivBlocksHierRec is able to process some rather generic data structures; nevertheless, some file formats have defined additional structures. For instance, PDF incorporates any encodable content into *streams* [ISO/TC 171/SC 2, 2008] as raw bytes, which, when interpreted, may be decoded to images, audio, etc. Decoding these streams is a complex task, as each encodable content differs; thus, we proceed differently. We calculate a cryptographic hash of the raw bytes and put it inside the contents field of a block. Therefore, if a stream is changed by a malicious party, one is able to locate the modification due to a failed integrity check on the corresponding block.

We need to consider the PDF from two points of view: its *visual content* and its *internal content*. We note that the visible written contents of a PDF file can also be encoded into a stream. In other words, our algorithm divides the PDF internal content into blocks, not the visual content per se. This procedure is necessary since any component can overlay the original written content due to features of the PDF file format [ISO/TC 171/SC 2, 2008, Section 7.5.6]. Moreover, the modification of a PDF file entails changes to several internal structures and, therefore, blocks. Hence, users intending to sign PDF documents using MTSS should select a higher d.

3.1.3 Sequential fixed-size blocks

Although we have discussed the negative implications of sequential fixed-size blocks, some file formats, such as images, benefit from this perspective. We consider dividing a to-be-signed image into fixed-size blocks of the same width and height. Also, some of these formats allow the inclusion of arbitrary information in the header. However, we only consider applying the block division to the image pixels and identifying whether a "tile" was altered.

To expand the range of individual blocks, the user inputs the desired size of a block into the application. We note this approach is useful for images but not text-written formats. When attackers modify images, the content is normally altered but not added to or removed. The cascade effect only happens when we modify the file structure per se. In text-focused documents, attackers can make any of the previous since they aim to modify the original signed content.

We consider Portable Gray Map (PGM), raster (bitmap), and Portable Network Graphics (PNG) images in our discussion and implementation. We choose such image formats as they are representative of a wide variety of use cases, as the formats have one, three, and four channels, respectively. The division of blocks is similar to the one proposed by Luo [2024]. Let $b' \in \mathbb{N}$ be a block size, and i an image with width and height $w, h \in \mathbb{N}$. The number of blocks is $n = \lceil \frac{w}{b'} \rceil \times \lceil \frac{h}{b'} \rceil$. If i has multiple channels, for each pixel, channel values are concatenated and hashed.

3.2 CFF parameters

We remark on some initial observations from the algorithms described in Section 2.2:

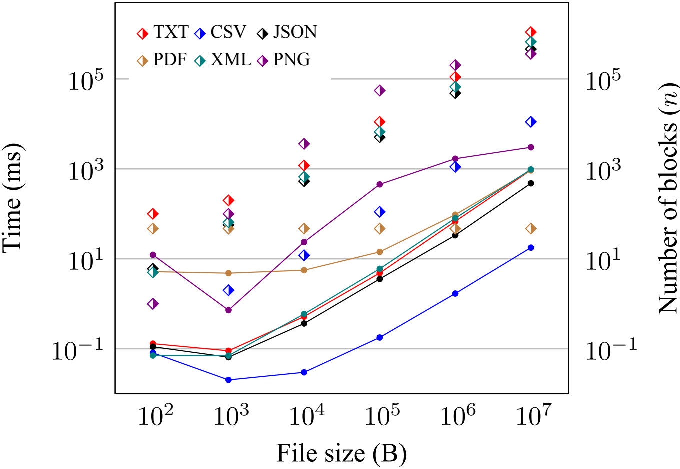
(i) the signature size grows with the number of blocks n;
(ii) to efficiently correct modifications, it is suggested to limit the block size to a small enough size s. However, this minimization leads to more blocks;
(iii) larger blocks lead to smaller n and consequently smaller t, which implies smaller signatures and consequently faster verification.

In this context, we aim to find ideal parameters for the polynomial construction of CFFs, i.e., $d \geq 2$. From Section 2.1, we recall that $n = q^k$ or $n = t^{k/2}, t = q^2$, and $d \leq \lfloor \frac{q-1}{k-1} \rfloor$.

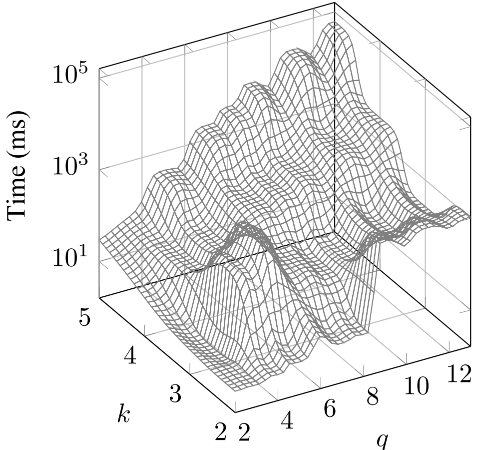
To generate smaller CFFs and improve efficiency, we increase the gap between the number of blocks and tests to minimize t. The proportion between n and t is given by q^{k-2}, which gives us the signature compression rate; hence, larger values for k lead to further gains in compression. However, due to the upper bound of d, we observe that a smaller k is 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 very large and out of the scope of most applications.

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 k 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 $d \geq 2$.

We observe that the number of blocks $n' = |m|$ obtained from DivideBlocks may not fit the $n = q^k$ parameter restriction of Proposition 1 for a chosen $d \geq 2$. To solve this, we must choose a d-CFF(t, n) with $n \geq n'$ and reduce the number of columns without changing its capability of identifying up to d modifications. Since the d-CFF property from Definition 1 is true for any $d + 1$ blocks considered, it remains true as long as $n \geq d + 1$. Therefore, without loss of generality, we simply remove the last $n - n'$ columns for the chosen



(a) Performance of DivideBlocks. Lines represent the time spent, and diamonds represent the number of blocks.



(b) Performance of CreateCFF.

Figure 2. General performance of auxiliary algorithms.

d-CFF. Furthermore, we define an auxiliary data structure to speed up the process of creating CFFs without the need to recompute parameters.

Definition 3 (CFF parameter associative array) *Let q be a prime power in {7, 8, ..., 61, 64} and $2 \leq k \leq 18$, with $k < q$. For all q, k, calculate d, n and t according to Proposition 1. We define A to be an associative array whose keys are all possible values of d, and the corresponding values are sequences of tuples in the form (q, k, n, t).*

For a given d, the associated sequence of tuples in A is ordered in the usual lexicographic order (i.e., smaller values of q come first; if there is a tie, smaller k come first, and so on). We remark that the limitations on q and k, which produce CFFs with $d \leq 31$, are imposed solely for performance reasons. We now describe the CreateCFF algorithm.

CreateCFF**(d, m', A).** Let $d \in \mathbb{N}$ be the maximum number of modifications to be located, m' a document, and A an associative array constructed via Definition 3. The algorithm proceeds as follows.

1. Set $(m, n') \leftarrow \text{DivideBlocks}(m')$.
2. Set $c \leftarrow \mathbf{0}$.
3. If $d = 1$, set c to be a 1-CFF(t, n) with $n \geq n'$, constructed according to Construction 1 and go to Step 6. Otherwise, go to Step 4.
4. For all key-value pairs $(d, (q, k, n, t)) \in A$, if $n \geq n'$, set c to be a d-CFF(t, n) constructed according to Proposition 1 and go to Step 6.
5. If $c = \mathbf{0}$, return $\bot$. Otherwise, go to Step 6.
6. Set $\mathcal{M}$ to the incidence matrix of c. If $d \neq 1$, go to Step 7. Otherwise, go to Step 8.
7. Set $\mathcal{M} \leftarrow (\mathcal{M}_{i,j})_{1 \leq i \leq t, 1 \leq j \leq n-n'}$.
8. Return $\mathcal{M}$.

We observe that, at Step 5, if $n < n'$ for all parameters in A, the algorithm aborts. This is also due to performance reasons, as q and k grow rapidly.

Remark 2 *Per our description of* CreateCFF*, we now require that d is appended to σ at the end of the signature generation algorithm, as it is otherwise impossible to reconstruct $\mathcal{M}$ given the message m' and σ. In other words, we must know the difference $n - n'$ to correctly rebuild the CFF. Further information on the performance of the algorithm is given in Section 4.2.*

4 Experiments

We provide an open-source implementation 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.

We start by addressing $\mathbf{Q_4}$. The choice of Σ does not affect the overall performance of MTSS; it remains consistent with the traditional Σ.Sig operation, as explored in the earlier version of this paper [Kamers *et al.*, 2024]. We hereafter choose to set Σ to ML-DSA with NIST security level I (ML-DSA-I), except when otherwise noted, as it is a recently standardized post-quantum signature scheme [NIST, 2023].

Security levels are those given by NIST in the post-quantum cryptography standardization process [NIST, 2016]. 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.

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.

We assume that all CFFs used are already cached and pre-

viously serialized to memory, except when otherwise noted. Finally, we execute $\mathrm{DivideBlocks}$ before every computation of Sig and VCor; for the Ver algorithm, it depends on whether the original message has changed or not.

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 $\mathrm{DivideBlocks}$ and $\mathrm{CreateCFF}$.

4.1.1 Document division into blocks

Intrinsically, some files take longer to separate into blocks than others due to their syntax and different parser implementations. For instance, XML files have the additional overhead of canonicalization, and PNG images may have long headers and the necessity of manipulating many layers of images (and colors). It is important to recall that the performance of $\mathrm{DivideBlocks}$ affects signature generation (Sig) and signature localization (Ver with the number of blocks altered $|I| \geq 1$).

To assess the practical impact of using MTSS in real-world scenarios, it is essential to understand how different types of documents are consumed. According to Mlynkova *et al.* [2006], the average size of XML documents is approximately 4.6 kB and typically contains a small number of nodes, with 99% of XML documents having fewer than eight levels. Similarly, Rodríguez *et al.* [2016] provides an overview of file types used and accessed through REST APIs, reporting that the median size of JSON files is 1.5 kB; this work also addresses that images (JPG, GIF, and PNG), JSON, and XML are among the ten most commonly used resources.

Furthermore, reports from Crawl [2024] and Johnson [2021] indicate that PDFs are the second most widely used document format on the internet, with usage steadily increasing over time [Xiong, 2020]. On the other hand, CSV files are predominantly used for importing/exporting datasets, creating backups, or training machine learning models. While CSV files tend to be larger in size, there is currently no precise data available regarding their average size.

Figure 2a summarizes the performance of $\mathrm{DivideBlocks}$ considering plain text, CSV-b, XML, JSON, PDF, and PNG images of several sizes, separated into 20×20 squares. We remark that the algorithm is more affected as the number of blocks increases, not the file size necessarily; as the number of blocks grows, the algorithm becomes slower. However, as the size of the document grows, some types of documents keep the same number of blocks n, particularly PDF files.

One may wonder how PDF files are so fast using our algorithm; we have two considerations: PDFs use more references to already existing objects as they grow exponentially, and the number of blocks remains almost fixed as the size of the file grows. Nevertheless, other documents explored do not have these concerns, as they have specific delimiters (block delimiters, key-value mappings, or sequential fixed-size) and do not have references to already existing objects. With the performance results obtained, we can assert that the performance of MTSS is not harmed by using $\mathrm{DivideBlocks}$ as long as the file is not too large (exceeds 10^4 blocks). More information will be provided in Section 4.2.

4.1.2 CFFs construction

We recall that, for the case of $d = 1$, the Sperner construction gives the best values for t. For the case $d \geq 2$, we consider the construction based on polynomials over finite fields, with $n = q^k$ and $t = q^2$ for q a prime power and k a positive integer. Figure 2b shows the performance of creating CFFs with this construction 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 Section 3.2, and we can generate a large CFF 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 overall efficiency of the framework.

4.2 Signature generation, verification and location of modifications

Evidently, MTSS signature operations cause additional overhead on top of traditional signature schemes, particularly due to $\mathrm{DivideBlocks}$ and $\mathrm{CreateCFF}$. We discuss how the choice of Σ, $\mathcal{H}$, and $\mathcal{M}$ impacts the overall performance of each MTSS operation and other implementation details.

As per Remark 1, we performed modifications to the original MTSS description, so we did not need to hash each block while concatenating on Step 1 of Sig algorithm. While keeping the same security implications, we have highly improved its performance. Specifically, this modification lowers the number of hash computations in Sig and Ver by a factor of n. Empirically, our implementation demonstrates that this optimization improves signature generation time by an average factor of $55\times$, and verification time (with $|I| \geq 1$) by up to $25\times$. In our current work, all experiments are performed using this improvement.

Considerations about Sig**.** We divide the overall signature generation procedure into two separate stages, `PreSign` and Sig, for ease of discussion. In the `PreSign` stage, we handle I/O operations, segment the message into blocks based on the document type using $\mathrm{DivideBlocks}$, and create or parse the cached CFF with $\mathrm{CreateCFF}$. As a practical consequence, the implementation of signature generation takes an additional parameter d so that auxiliary algorithms need not be invoked by the user. The Sig stage implements the procedure of MTSS Sig as explained in Section 2.2.

When compared to the Sig algorithm presented in Section 2.2, some small alterations were performed due to Remark 2; we recall we are not able to reconstruct the CFF for verification using the $\mathrm{CreateCFF}$ algorithm, because we are not aware of the parameter d used for signing. We change Step 3 of algorithm Sig to "Set $T \leftarrow (T_1, \ldots, T_t, \mathcal{H}(m), d)$". As T is signed in Step 4, we can guarantee the integrity and

Table 1. Performance and signature size of MTSS(ML-DSA, $\mathcal{H}, \mathcal{M}$) Sig for several choices of $\mathcal{H}$ and NIST security levels, using files with different n. Values in parentheses are Σ.Sig. PreSign stage is not considered.

$\mathcal{M}$	NIST	Sig time (ms)						$\|\sigma\|$ (bytes)	
		SHA-2		SHA-3		BLAKE			
		256	512	256	512	2s	2b	256	512
2-CFF (25, 125)	I	3.98 (0.12)	0.63 (0.08)	0.59 (0.09)	0.43 (0.16)	0.29 (0.07)	0.28 (0.15)	3254 (2420)	4086 (2420)
	II	0.80 (0.15)	0.57 (0.07)	0.60 (0.10)	0.68 (0.17)	0.39 (0.20)	0.28 (0.10)	4127 (3293)	4959 (3293)
	III	0.53 (0.28)	0.54 (0.13)	0.48 (0.12)	0.44 (0.11)	0.36 (0.12)	0.31 (0.15)	5429 (4595)	6261 (4595)
2-CFF (49, 2401)	I	4.05 (0.14)	3.92 (0.10)	3.80 (0.19)	4.09 (0.12)	3.67 (0.07)	3.59 (0.06)	4022 (2420)	5622 (2420)
	II	4.01 (0.23)	3.90 (0.14)	3.91 (0.15)	4.09 (0.19)	3.83 (0.10)	3.69 (0.15)	4895 (3293)	6495 (3293)
	III	3.99 (0.15)	3.98 (0.12)	4.01 (0.13)	4.16 (0.17)	3.79 (0.17)	3.69 (0.12)	6197 (4595)	7797 (4595)
3-CFF (121, 14641)	I	72.04 (0.42)	70.40 (0.36)	71.35 (0.33)	72.92 (0.53)	69.80 (0.40)	69.40 (0.20)	6326 (2420)	10230 (2420)
	II	72.11 (0.44)	70.95 (0.34)	71.16 (0.41)	72.75 (0.75)	70.01 (0.38)	69.23 (0.58)	7199 (3293)	11103 (3293)
	III	74.83 (0.50)	70.84 (0.48)	71.53 (0.55)	73.26 (0.58)	70.52 (0.36)	70.01 (0.31)	8501 (4595)	12405 (4595)

authenticity of its data. We use d to reconstruct the necessary blocks in the Ver algorithm, as it will be explained next.

Considerations about Ver**.** Analogously, we divide the signature verification procedure into PreVer and Ver. The PreVer stage handles I/O operations by reading the message m, signature σ, and public key pk from the disk; the PreVer stage follows the Ver algorithm. Again, we set $T = (T_1, \ldots, T_t, h_m, d)$ as explained above. We note that varying the number of altered blocks $|I|$ does not greatly impact location since Step 4 of Ver recognizes all modified blocks in a loop, not stopping until it reaches all tests; for this reason, our experiments consider $|I| = 1$ for simplicity.

In the context of MTSS, the usual signature verification algorithm is performed when $|I| = 0$; we refer to this process hereafter as Ver_0. Otherwise, the *location* procedure is able to identify how the original message was modified, referred to as Ver_1. In particular, the division of a message into blocks and the construction of a CFF are only necessary if we are locating errors. Consequently, we opt to perform experiments focusing on the performance of the location.

Discussion. We start by providing good choices of Σ (variations of ML-DSA), $\mathcal{H}$ and $\mathcal{M}$, subsequently answering $\mathbf{Q_4}$, $\mathbf{Q_3}$ and $\mathbf{Q_2}$. We first show how the hash function affects MTSS operations and then present how increasing the number of blocks n and tests t on a CFF $\mathcal{M}$ affects the overall process. Table 1 shows the performance of MTSS signature generation for various use cases. We remark that Sig has an influence on the underlying CFF used, besides including a call to traditional Σ.Sig; naturally, Sig is supposed to have a slower performance than a traditional signature generation.

We point out the growth in signature size and lower performance for documents with larger n and t. Larger n and d require larger t, which is directly related to the number of hash calculations performed during the signature generation. Consequently, the choice of $\mathcal{H}$ plays a crucial role in the performance of signature operations. We hereafter set $\mathcal{H} =$ BLAKE2b, as it showed the best performance among the evaluated functions. We note that the choice of Σ does not affect the overall performance, remaining consistent compared to a call to the traditional signature generation.

The right-hand side of Table 2 shows similar consequences to the performance of verification and location. Verifying the signature with the original message, i.e., with $|I| = 0$, is comparable to Σ.Ver, albeit with the overhead of the MTSS signature being larger (cf. $|\sigma|$ in Table 1). On the other hand, the location performance relies on different factors. The influence of the number of tests t and blocks n in modified messages is apparent; we recall that in matrix $\mathcal{M}$, we have t rows and n columns. Therefore, the more these values increase, the more time is required to identify alterations to the original message.

Table 2 also complements the answers regarding the CFF parameters on $\mathbf{Q_2}$ and gives an overview of different message sizes. The upper group demonstrates how performance is affected by using different files (increasing the number of blocks n); on the other hand, the lower group shows how the same message m behaves, expanding the number of possible error localizations d.

We point out the execution time for the Sig and the Ver_1, highlighting how the framework's efficiency decreases as n, d, and t increase. This growth in parameters enhances the precision in error localization but comes at the cost of increased computational time for signature and location, as well as signature size. More specifically, for a fixed document, increasing n by dividing it into smaller blocks also increases t, improving error localization granularity. However, with more blocks, a higher d is also needed, which further impacts t. We observe the following tradeoff: a smaller t gives a more efficient location, while the tolerance to a larger number of modifications d requires a larger t.

We highlight that the Sig performance time from Table 1 and Table 2 only considers the Sig stage. This allows us to observe the overall CFF impact over the procedure, as well as analyze the choice of Σ and $\mathcal{H}$. The total time of Sig must be added to the time of PreSign. To give a more general overview of the impact of MTSS applied to different types of documents and a variation of the number of blocks, we present Figure 3. We highlight the various stages of the algorithms to facilitate our demonstration of results.

We observe how PreSign easily overcomes the Sig stage for files with greater n (some formats are more efficient than others); in other words, the DivideBlocks impact over Sig algorithm is greater proportionally to bigger files. As mentioned earlier, Ver is rather efficient for simple verification. As n grows, and subsequently, the CFF parameters and the

Table 2. Performance of signature operations of MTSS(ML-DSA-I, BLAKE2b, $\mathcal{M}$) for several choices of $\mathcal{M}$, plain text files of different sizes. For ease of notation, we use Ver_0 to refer to Ver with $|I| = 0$, and Ver_1 to refer to Ver with $|I| = 1$. PreSign stage is not considered.

Parameters of $\mathcal{M}$					Size (bytes)		Time (ms)			
d	k	q	t	n	m	σ	Sig	Ver_1	Ver_0	Σ.Ver
2	4	7	49	2401	4801	5622	8.49	29.06	0.05	0.08
2	4	9	81	6561	13121	7670	32.16	154.84	0.08	0.09
3	4	11	121	14641	29281	10230	94.93	627.10	0.11	0.09
4	4	13	169	28561	57121	13302	247.25	2047.66	0.19	0.13
1	-	-	15	4096	8191	3446	8.49	47.75	0.11	0.07
2	4	8	64			6582	16.40	70.01	0.07	0.06
7	3	16	256			18870	47.77	172.08	0.10	0.08
63	2	64	4096			264630	570.53	2201.24	0.44	0.06

size of the signature, the location algorithm is greatly impacted, taking more than 60% of the overall process (using all algorithms) for most formats.

In view of this discussion, we argue that MTSS is applicable to real-life scenarios, and its performance is not prohibitive. Naturally, signature sizes grow with d, n, and t grow; however, we remark that this is a reasonable trade-off, considering the location and correction features obtained by using an MTSS signature.

4.3 Correcting modifications

The VCor algorithm supersets Ver: modified blocks are first located, and then corrections are performed. As mentioned in Section 2.2, we need to brute-force all possibilities of the modified block, considering some character encoding. Aspects such as the number of blocks, the number of modifications, the 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 consider the MTSS(ML-DSA-I, BLAKE2b, 7-CFF$(126, 4096)$) parameter set and define $s \in \mathbb{N}$ as the size of each block in bytes; we remark that block sizes do not need to be homogeneous in size, but for our experiments, we will always consider blocks of the same size. The error correction process operates at the block level, where at most $|I| \leq d$ blocks may require correction. Consequently, the total number of bytes to be corrected is $|I| \times s$. We observe that the overall performance for locating and correcting errors is linearly proportional to $|I|$. From this, we infer that the time complexity of the algorithm is $\mathcal{O}(|I| \times f(s))$, where $f(s)$ represents the super-linear growth rate concerning s.

Remark 3 *Due to the modification provided and commented on Remark 1, Step 6 of* VCor *needs to provide all the r-length combinations of elements while allowing individual elements to have successive repeats. This procedure makes* VCor *algorithm performance to become poorer. While the procedure of hashing individual blocks makes the performance* Sig *and* Ver *worse, not making it for* VCor *becomes even worse due to the required combinatorial algorithms.*

We give some simple time measurements for $|I| = 1$: (i) for $s = 1$, the algorithm took 576.28 ms; (ii) for $s = 2$, the algorithm took 2141.63 ms; (iii) for $s = 5$, we estimate that the algorithm would take approximately 80 days. We also provide simple time measurements for $s = 1$: (i) for $|I| = 1$, the algorithm took 68.78 ms; (ii) for $|I| = 4$, the algorithm took 307.12 ms; (iii) for $|I| = 7$, the algorithm took 527.88 ms. This is not surprising due to the nature of the brute-force algorithm for the correction of modifications. We conclude that this is only viable for very small block sizes s and a very small number of modifications $|I|$.

5 Ensuring data integrity of individual blocks

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

> $\mathbf{Q_5}$ Is it possible to verify the integrity and authenticity of a subset 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 signed dataset, ensuring its integrity and 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. Many applications are considered within this scenario, such as IoT, certificate transparency, Blockchain, and countries' federal registers. The approach we propose in this section allows the entire document to be signed once, enabling the verification of integrity and authenticity for each subset of the original signed data 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.

5.1 Definition and requirements

We formalize what we call the *belongingness* framework by a tuple of algorithms $(\text{Sign}, \text{PartVer})$ as follows. Let S be a server that keeps the information about a signed message m and corresponding signature σ. Let C be a client with access

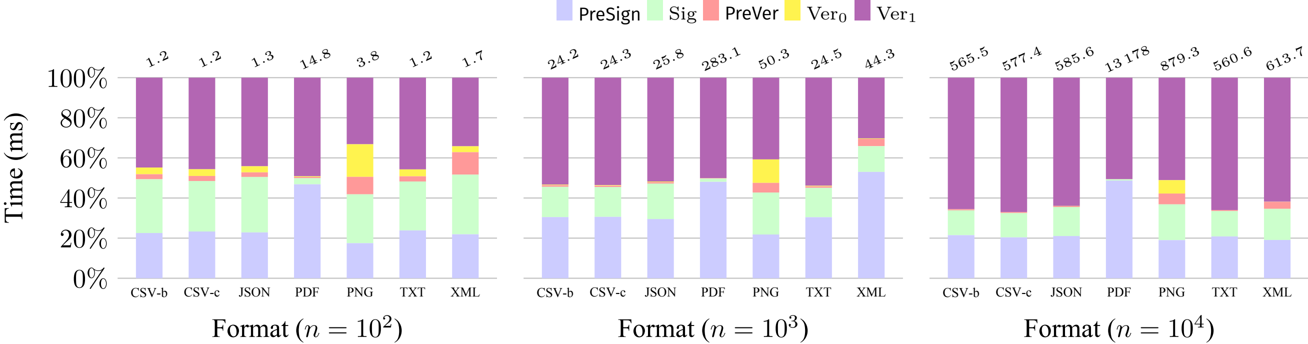


Figure 3. Relative time spent in signature operations for MTSS(ML-DSA-I, BLAKE2b, $\mathcal{M}$), for different number of blocks n (and CFFs $\mathcal{M}$). The numbers at the top of each bar represent the total time spent.

to σ and a piece of data m_j, and is interested in verifying whether $m_j \in m$. When performing this verification, the client is not only convinced that m_j is a subset of m, but it holds its integrity and authenticity by σ.

Let $m = (m_1, \ldots, m_n)$ be a message and Sign be an algorithm that takes as input m and a secret key sk and produces a digital signature $\sigma = \text{Sign}(m, sk)$. The partial verification algorithm PartVer takes as input a message m_j, a public key pk and σ, and outputs $\top$ only when $m_j \in m$.

We highlight that PartVer should not require the entire signed message m to verify that $m_j \in m$. Ideally, σ should carry enough information to guarantee integrity, authenticity, and non-repudiation of m_j, as well as to guarantee that m_j is part of m. In practice, we may need some extra information in order to provide such properties (as shown in Section 5.5).

A trivial instantiation of this scheme would consist of signing each m_i in m individually as $\sigma_i = \text{Sign}(m_i, sk)$ and creating $\sigma = (\sigma_1, \ldots, \sigma_n)$. The verification PartVer would then verify m_i with each $\sigma_j \in \sigma$ and, if one of them is successful, it outputs $\top$. In this case, the verification requires only m_i and the list of n signatures. We note that the trivial solution has a complexity that grows linearly with n and requires us to generate, store, and potentially verify n digital signatures, which can be infeasible for applications in big data.

We acknowledge that there might be several ways to approach this problem in a non-trivial way, and in order to analyze and compare possible solutions, we consider the following desirable (and possibly competing) objectives:

(a) minimize time consumption for creating σ;
(b) minimize the size of σ that needs to be stored by the remote server;
(c) minimize the amount of data that a client/verifier needs to download to run PartVer;
(d) minimize the overall amount of data transmitted between client and server.

5.2 Practical example

To provide an intuitive explanation of the belongingness framework, consider the example depicted in Figure 4. In the framework, we have a client C, which has only a single fragment of the original document and aims to verify its integrity and authenticity without requiring access to the entire signed message, which is located in S.

Let S be the server and let $m = (a, b, c, \ldots, l)$ denote the original message composed of multiple blocks, which is signed by S using the Sign algorithm from the belongingness framework; this process yields a digital signature $\sigma = \text{Sign}(m, \mathsf{sk})$ under the author's private key sk. The resulting signature is made publicly available alongside any auxiliary data necessary to support partial verification, i.e., the PartVer algorithm.

Suppose a client, denoted as C_1, holds block b and wishes to verify whether this block is part of the original message m, was authored by the original author, and remains unaltered. To do so, client C_1 initiates a verification query to the server S, sending the tuple (b, pk). The server S responds with auxiliary data required to reconstruct the relevant verification context, here denoted as $\{\text{Aux}\}$. Using this information, C_1 runs the PartVer algorithm, and obtains output $\top$, indicating that block b is indeed part of the signed message and maintains both its integrity and authenticity.

On the other hand, client C_2 holds a block z, which does not belong to the original signed message m. Client C_2 likewise communicates with the server S, sending the tuple (z, pk). The server processes the request and returns the necessary verification information. However, executing the PartVer algorithm, the output is $\bot$, meaning that block z does not belong to the signed message m.

This example concretely demonstrates how the belongingness framework must work so clients can verify the integrity and authenticity of specific blocks of a signed message without requiring access to the entire content. Notably, the Sign and PartVer algorithms must be independent of the underlying signature scheme, and may be instantiated using various constructions.

5.3 Applications for the belongingness framework

Many applications shall benefit from using the belongingness framework. For instance, applications that require the generation and verification of several digital signatures could especially benefit from it by saving on verification time and the amount of data transmitted. In this section, we survey some potential applications to motivate our framework.

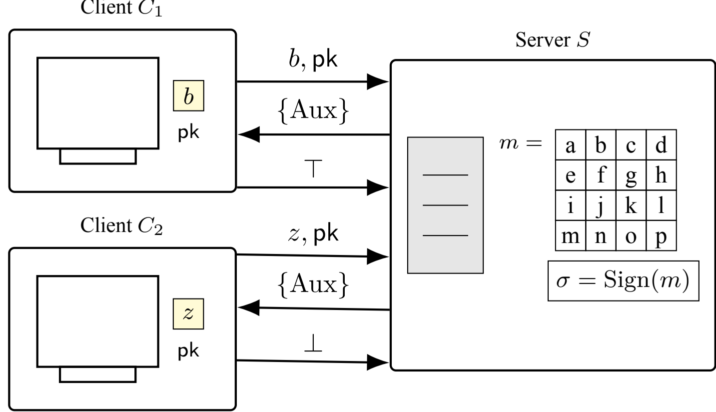


Figure 4. Belongingness framework verification process. The server holds the original signed message $m = (a, b, \ldots, l)$ and the signature σ. A client C_1 positively verifies whether block b belongs to m, and holds its integrity and authenticity, while client C_2 negatively verifies block z; in other words, we verify that $b \in m$, but $z \notin m$, while not requiring access to the entire message m.

When we discuss IoT scenarios, we consider embedded devices whose primary goal is to perform their specific task (usually communicating with other devices) at the highest performance and the lowest energy cost, and communicate to a central server to share their findings. Such communication often needs to be secured and verified to check if the integrity and authentication of the data are maintained. In situations where multiple devices need to authenticate specific contents, the belongingness framework allows efficient verification of the integrity and authenticity of a portion of the transmitted data. This is particularly useful in resource-constrained IoT environments [Yan *et al.*, 2023; Chen *et al.*, 2021].

Certificate transparency [Laurie *et al.*, 2021] creates public logs to determine whether certificate authorities have correctly issued digital certificates. Traditionally, each certificate in the log must be individually signed, leading to significant storage overhead and requiring users to download and verify all logged certificates to check for a specific one. In contrast, the belongingness framework optimizes storage by removing the need to store an individual signature for each certificate. Instead, it computes and stores a compact cryptographic structure that can attest to the integrity and authenticity of multiple certificates in the log. This eliminates the need for full log access during the signature verification.

Distributed ledgers in blockchain provide strong security guarantees but can be inefficient in certain verification processes, particularly those requiring multiple lookups. For instance, consider the case where a payer wants to later prove to an auditor that a specific signed transaction exists within the set of all transactions they have issued. In a naive approach, this verification would require checking and validating multiple digital signatures, e.g., for n signed transactions, we potentially require n separate verifications, which can be computationally expensive. The belongingness framework can potentially improve performance in this scenario. Instead of verifying each transaction individually, the auditor can efficiently check the presence of a specific transaction using $\mathrm{PartVer}$ [Han *et al.*, 2023].

Finally, one particular application inspired the creation of the belongingness framework: federal registers. Nations are required to publish daily records of legislative activities to ensure transparency and keep citizens informed about legal developments; some examples include the United States Federal Register and the Brazilian *Diário Oficial da União* (DOU). The digital version of the federal registers is often signed, typically in PDF format, and shared online. However, individual legislative records should be verifiable in terms of integrity and authenticity without requiring access to the entire document. In order to achieve this, each record is signed separately, leading to n signatures for a PDF with n pages. The belongingness framework offers an alternative: instead of signing each record individually, a single signature on the entire daily federal register allows verification of any specific record while preserving its authenticity and integrity.

5.4 Literature Review

The definition of belongingness presented above is generic and open, and many possible solutions exist, addressing several of the desirable aspects. Here, we present some existing schemes in the literature that could be further explored as an instantiation of the proposed belongingness framework.

5.4.1 Locally Verifiable Aggregate Signatures (LVAS)

Consider multiple pairs (m_i, σ_i) of messages and their corresponding signatures. Aggregate signatures allow the compression of the individual signatures σ_i into one single aggregate signature σ', saving storage, transmission, and verification time. However, to determine whether a message m_j belongs to the set of messages m signed by the aggregate signature σ' (or, in other words, to verify σ'), we need access to all individual messages m_i.

To overcome the challenge of needing to access all $m_i \in m$, Goyal and Vaikuntanathan [2022] propose a Locally Verifiable Aggregate Signature (LVAS), which introduces an auxiliary "hint" that enables efficient verification of individual messages. Given an aggregate signature over a set of signed messages m, a short hint h is generated for any specific message $m_i \in m$. This hint preserves the integrity

and authenticity properties of m_i while allowing verification without requiring access to the entire message set.

If we consider $m = (m_1, \ldots, m_n)$ and σ' as the aggregate digital signature of m, we can consider the local verifiable signature scheme as a possible instance of a belongingness framework; with that, we are able to answer whether $m_i \in m$ using both σ' and the hint h. In the context of our framework, this mechanism aligns with the $\mathrm{PartVer}$ algorithm. The verification process of local signatures follows a similar approach: the verifier takes as input the public key pk, the specific message m_i, the aggregated signature σ', and the corresponding hint h. Notably, the hint can be precomputed and stored. However, the procedure of hint generation is costly (the author does not provide measurements) and must be stored somewhere (whether in the client or the server).

By using LVAS as a solution for the belongingness problem, we can observe the following aspects: (a) for n individual messages, n digital signatures need to be previously generated; (b) regarding storage, only the aggregated signature σ' must be kept in the server S; (c) the client C must have access to the four parameters mentioned in the verification algorithm: $(\mathsf{pk}, m_i, \sigma', h)$; (d) finally, the only data transmitted between the parties consists of the four parameters mentioned.

The hint generation algorithm is computationally intensive (the authors do not provide specific performance measurements). Additionally, it remains unclear where the hint h would be located or how it could be securely stored and easily accessed by the client C. Managing the hint h could become challenging if the client is required to maintain a different hint for each different message $m_i \in m$. Furthermore, the process still involves generating n individual signatures and subsequently aggregating them, which may not be efficient for big data scenarios.

5.4.2 Partial verification using MACs in industrial IoT (PV-MAC)

In short, industrial IoT fog-based systems have three parties: the data source (sensors), the fog node (which is resourceful and processes most of the data), and the data user (IoT devices). Data users require messages from data sources, but do not have enough resources to verify the integrity of all message cells, so this workload is divided among the fog nodes. However, the fog node has other responsibilities, such as data analysis and training, and delays can vary depending on the available resources at the moment.

Traditional Message Authentication Code (MAC) schemes require full-message verification, which is inefficient in such applications, particularly when dealing with high-frequency and large-volume data transmissions. Yan *et al*. [2023] propose a Secret Sharing Message Authentication Code (SS-MAC) scheme that leverages secret sharing to enable partial verification, or PV-MAC. In this approach, a message is divided into k cells, each acting as a secret sharing key. It combines with a Prefix-verifiable Message Authentication Code (PMAC) so the data user can assure the integrity of the cells it should know.

These keys allow the message to be reconstructed only when a sufficient subset of cells is verified. The fog node, which has greater computation resources than individual IoT devices, selects a subset of at least k cells for verification. If this subset meets the predefined threshold, the regenerated MAC from the selected cells will match the MAC of the entire message, ensuring its integrity without requiring full message verification.

This method naturally integrates with the $\mathrm{PartVer}$ algorithm. Here, the fog node generates and verifies the MAC over a selected subset of k cells, and if the threshold is met, the authenticity of the entire message is ensured. This enables efficient verification while reducing computational overhead for resource-constrained IoT devices.

Nevertheless, this approach is better suited for lighter applications due to the inherent overhead associated with managing secret sharing keys. While it may be optimal for industrial IoT in fog-based environments, its scalability limitations make it less suitable for Big Data scenarios, which is our primary focus.

5.5 Our proposal using MTSS

We positively answer $\mathbf{Q_5}$ by instantiating our *belongingness* scheme using a variation of MTSS. We consider the signed message to be split into n blocks, Sign is the MTSS signature algorithm (Sig) as defined in Section 2.2, and $\mathrm{PartVer}$ is here referred to as $\mathrm{BlockVer}$, and aims to identify if a piece of message m_j is part of the full message m. In other words, our belongingness instantiation is given by (Sig, $\mathrm{BlockVer}$) as presented next.

We define 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 $m = (m_1, \ldots, m_n)$ is the message already split into n blocks; $\mathcal{H}(m)$ is the cryptographic hash of m; $\sigma = (\sigma', T)$ is an MTSS signature, and $\Sigma, \mathcal{H}, n$ are MTSS parameters as shown in Section 2.2. Without loss of generality, S uses $\mathcal{H}(m)$ as a unique index to identify the corresponding message m. We assume that the client C knows $\mathcal{H}(m)$, which is made publicly available by S. We recall that S can reconstruct the d-CFF $\mathcal{M}$, given the necessary parameters.

First, we provide an algorithm called $\mathrm{BlockVer}$ that allows for the verification of the integrity and authenticity of a single block from the MTSS scheme. Note that the following algorithm requires an MTSS signature, a block m_j that may or may not be part of the signed message, but also some extra blocks m_ℓ in order to perform this verification. This is because the signature is composed of hash values T_i, created by concatenating some blocks of the message m according to a d-CFF $\mathcal{M}$.

$\mathrm{BlockVer}$**(**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 an 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 (in such a row over an underlying CFF), $M = (m_\ell \in m : \mathcal{M}_{i,\ell} = 1, \ell \neq j)$, and k is the relative position (index) of $m_j \in T_i$. The algorithm proceeds as follows.

1. Set $r \leftarrow \Sigma.\mathrm{Ver}(\sigma', T, \mathsf{pk})$. If $r = \perp$, output $\perp$. Otherwise, go to Step 2.
2. Insert m_j in the k-th position of M, so that $M = (m_{\ell_1}, \ldots, m_{\ell_{k-1}}, m_j, m_{\ell_k}, \ldots, m_{\ell_{|M|}})$.
3. Set $h' \leftarrow \mathcal{H}(m_{\ell_1}||m_{\ell_2}|| \cdots ||m_{\ell_{|M|}})$.
4. If $h' = T_i$, output $\top$. Otherwise, output $\perp$.

Step 1 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 4, 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 4 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 is an MTSS signature constructed from a d-CFF $\mathcal{M}$. Note that $\mathcal{M}$ has several other rows (of index) 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 4 outputs $\top$ for one of them, we know m_j is authentic and belongs to m. Then, we propose an iterative protocol between C and S as follows:

Belongingness protocol using MTSS.

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 until there are no more index i (of row in $\mathcal{M}$).
2. If S has no more index i, S sends $\perp$ to C, which terminates the protocol. Otherwise, S sends $Y = (\sigma, i, M, k)$ to C; the contents of Y are described in BlockVer.
3. C sets $r \leftarrow \mathrm{BlockVer}(m_j, \mathsf{pk}, Y)$.
4. If $r = \perp$, C sends index i (of row in $\mathcal{M}$) to S, so it decides on a new index i and goes back to Step 2. Otherwise, C sends $\top$ to S to end the protocol.

We highlight some observations regarding the chosen d-CFF $\mathcal{M}$. We observe that the number $|M|$ of extra blocks necessary for the verification of m_j depends on the number of 1s per row in $\mathcal{M}$. For instance, if $\mathcal{M}$ is constructed using the construction based on polynomials, we have $|M| = q^{k-1} - 1$. In this case, we are interested in constructions that minimize the 1s per row of $\mathcal{M}$, which is an open problem described in [Idalino and Moura, 2022]. Assuming no storage limitations, the server can focus on minimizing the number of 1s per row in exchange for a larger t and, consequently, larger signatures. This approach requires different CFF constructions than those we used in this work. Finally, it is important to observe that 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 1s in column j of $\mathcal{M}$. For the polynomial construction, this number is q; for the Sperner construction, it is $\lfloor \frac{t}{2} \rfloor$.

Table 3. Performance overview for the MTSS belongingness protocol. The network delay and the network bandwidth are not considered; these results are obtained offline. The CFFs for each file are the same as those used in Table 1. Plain text files were used for simplicity. Signed with Σ as ML-DSA level I and $\mathcal{H}$ as BLAKE2b. $|M|$ represents the number of blocks transmitted from S to C; the representation of $|I|$ is the number of modified blocks in the server data structure.

	n	$\lvert M \rvert$	Time (ms)		
			Step 1	Step 2	Step 3
$\lvert I \rvert = 0$	10^2	19	0.01	0.18	0.04
	10^3	142	0.01	5.35	0.07
	10^4	909	0.04	84.48	0.19
$\lvert I \rvert = 1$	10^2	38	0.01	0.2	0.07
	10^3	284	0.01	5.24	0.11
	10^4	1817	0.04	81.66	0.27
$\lvert I \rvert = 2$	10^2	76	0.01	0.2	0.12
	10^3	426	0.01	5.37	0.17
	10^4	2725	0.04	83.37	0.35

We emphasize that our solution uses the same signature algorithm as the MTSS scheme, but we execute a partial verification instead of the one from MTSS. We claim that our signature used in the protocol σ resulting from Sig is secure under the same assumptions proved by 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 $(\mathcal{H}(m), m_j)$ to S, or, to optimize network resources, $(\mathcal{H}(m), \mathcal{H}(m_j))$. Given that S is a resource-rich server in terms of processing and storage, it would be possible to create structural data that efficiently correlates $\mathcal{H}(m_j)$ 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 $\mathcal{H}(m)$ and its block m_j.

Our MTSS implementation for the belongingness framework builds upon the existing MTSS codebase. For simplicity, our focus is on verifying whether a single block $m_j \in m$ satisfies the belongingness properties. However, the algorithm could be extended to handle multiple consecutive blocks, enabling the identification of whether any subset of the original document preserves these properties. We remark that this procedure is out of the scope of this work.

Performance considerations The trivial solution presented in Section 5.1 needs to perform n signatures and store them in a list. Due to the MTSS properties, the Sig allows us to sign it only once. Thus, we minimize the time consumption for creating σ, regarding objective (a). As per objective (b), $|\sigma|$ grows according to the CFF parameters but is very compact, as discussed in Section 4.2.

The largest data the client C needs to download is the set of blocks M, which has size $q^{k-1} - 1$ for CFFs constructed via the polynomial construction. Depending on the construction of the d-CFF, this can be potentially large, which addresses the performance objective (c) and can be further explored and improved. The transmitted data from client to server

is very small since it is only a message hash and an index. However, the server-to-client data transmitted is larger due to M. These considerations are important for the performance objective (d).

Security considerations Hereafter, we consider a broad explanation of how the cryptographic assumptions under our protocol are secure. Nevertheless, we do not cover possible threats and attacks to the overall protocol.

The security definition of the belongingness protocol using MTSS ensures that any subset of data verified against an MTSS-signed message genuinely belongs to the original data, providing integrity, authenticity, and non-repudiation. Formally, security is assessed in a model where an adversary can adaptively attempt modifications, removals, or reordering of individual message blocks, aiming to forge a valid subset or compromise the authenticity of the message. Under these assumptions, the protocol is secure if an adversary has only a negligible probability of successfully verifying an unauthorized or modified subset.

Our belongingness proposal relies on d-CFFs and cryptographic techniques to achieve partial message verification. The correctness of the scheme and its security depend fundamentally on the properties of collision resistance and unforgeability provided by the MTSS signature; the MTSS signature security is based on the original work of [Idalino *et al.*, 2019]. Any successful adversarial strategy against the protocol implies the compromise of these cryptographic primitives.

In practical terms, our proposal tolerates controlled and predefined modifications consistent with its d-CFF structure, preventing unauthorized manipulations. Even if an attacker could make changes in random blocks in the server, the inherent location property of MTSS is able to identify the modification; if more than d modifications were performed, we are unable to complete the protocol, but we are able to identify the forgery.

As future work, we intend to formalize the security of the belongingness protocol in a game-based framework [Bellare and Rogaway, 2004] and over network control. In other words, we intend to prove that our belongingness is resistant to adaptive chosen message attacks, replay attacks, and over Auxiliary Information Leakage.

5.5.1 Implementation

Due to the usage of the MTSS signature, our protocol was implemented by adapting the existing MTSS codebase with punctual modifications. One key adjustment was the ability to store the document in pre-divided blocks and index, both the structured data and the corresponding MTSS signature, prefixed by the document's hash $\mathcal{H}(m)$.

To facilitate organization and testing, we store this information in a structured JSON format. The JSON file includes a mapping of each block index i to its corresponding message m_i and records the CFF parameters (t, n, d). This design choice simplifies access and verification within the protocol. Additionally, we store m_i directly rather than its hash, mining compatibility with the $\mathrm{BlockVer}$ algorithm, which operates on full message blocks.

Furthermore, all the procedure occurs as an iterative protocol between C and S as shown in Section 5.5. To evaluate the overall performance of such a protocol, we use the first three steps shown in the belongingness protocol using MTSS, where Steps 1 and 3 are client C attributions and calculations; and Step 2 is a server S calculation.

Table 3 summarizes the performance overview between the protocol steps. We remark that the Y parameter sent from S to C is composed of multiple items; however, the item that impacts the most is M, which is the set of extra blocks that appear together with m_j in the same row of the CFF. For performance measurements, we only considered the execution time in C and S, disregarding network bandwidth. We have provided experiments considering at least 2-CFF, i.e., we can provide integrity and authenticity for some block m_j even when two other blocks were altered on the server.

Table 3 considers elements with varied $|I|$, which, in this case, we consider as altered blocks in the server; we perform these experiments in order to test the iterative protocol of the belongingness framework using MTSS signatures. We can check how much slower Step 2 gets as we increase n and, consequently, the CFF; this happens because the server searches through the CFF, which possible rows can be used to check the integrity and authenticity of m_j. The impact of $|I|$ is not so relevant for the server, as it puts data in the cache until it finishes the protocol. Similarly, we can observe how much slower Step 3 gets ($\mathrm{BlockVer}$ algorithm) in the client as we grow the number of modified blocks in the server.

Each time we increase the number of altered blocks in the server $|I|$, the overall $|M|$ transmitted between C and S increases drastically. The behavior is due to the CFF properties. However, we can provide integrity, authenticity, non-repudiation, and the certainty that any block m_j belongs to m, even when there is corruption on the server side.

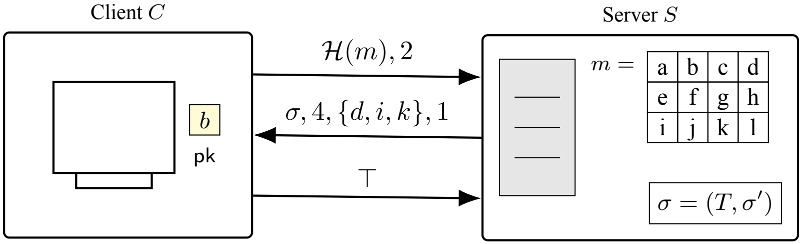
5.5.2 Practical example using MTSS

MTSS is particularly well-suited to instantiate the belongingness framework, due to its capacity to verify subsets of a signed message with tolerance to modifications. In this subsection, we describe interactively how our MTSS proposal applies to the belongingness situation, leveraging the $\mathrm{PartVer}$ algorithm, or, in this case, $\mathrm{BlockVer}$ algorithm, because of its CFF structure.

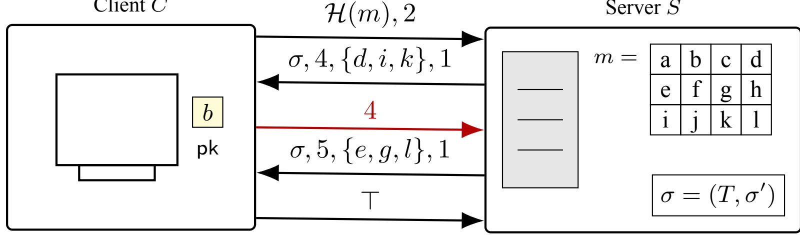
Figure 5 illustrates two scenarios in which MTSS is used as a belongingness instantiation. We note that server S holds a message $m = (a, b, \dots, l)$, its MTSS signature $\sigma = (T, \sigma')$, and its underlying CFF $\mathcal{M}$; for simplicity, we use $\mathcal{M}$ as a $2-\mathrm{CFF}(9, 12)$ as in Figure 1a.

In the first scenario (Figure 5a), client C holds block b and sends $X = (\mathcal{H}(m), 2)$, where 2 indicates b is the 2nd block in m. The server replies with a tuple $Y = (\sigma, i, M, k) = (\sigma, 4, \{d, i, k\}, 1)$, containing, respectively: the MTSS signature, the index i of a test row in the underlying CFF $\mathcal{M}$ such that block b appears, the set M of accompanying blocks in that row excluding b, and the relative position k of b in the concatenation order. The client then performs the partial verification $\mathrm{BlockVer}$ and obtains a positive result ($\top$), which confirms that b belongs to the original signed message m.

Since MTSS is inherently tolerant to modifications, the



(a) Practical example of MTSS being used for the belongingness framework.



(b) Practical example of MTSS being used for the belongingness framework, using its tolerant property.

Figure 5. MTSS-belongingness

belongingness verification can still be successfully performed even if some blocks of the original signed message m have been altered by an adversary on the server side. In the second scenario of our example (Figure 5b), client C initially receives a tuple whose verification fails, due to the presence of altered or corrupted blocks within M (either d, i, or k). Thanks to the tolerant nature of MTSS, the server can retry with a different row of $\mathcal{M}$ that also includes b but excludes the previously faulty blocks. This second attempt leads to a successful verification, returning $\top$. If the b block was altered on the server, it would take w iterations to find out, where w is the number of 1s in column b.

5.6 Comparison with literature review

Next, we perform a detailed comparison between the strategies used for instantiating the belongingness framework: our proposed approach using MTSS, the technique of LVAS, and the PV-MAC mechanism. We perform comparisons around the requirements stated in Section 5.1, covering the cost of signature generation, storage efficiency, partial verification cost, and overall scalability.

Signature generation In terms of signature generation, MTSS exhibits a significant efficiency advantage in our scenario of multi-block documents. During the signature process, our MTSS Sig algorithm automatically divides a to-be-signed document into blocks and encodes it using a CFF matrix. Only one cryptographic signature is required over a vector of hash digests; although MTSS results grow linearly with the number of CFF tests t (as demonstrated in Section 4), they grow logarithmically with the number of blocks n (as discussed in Section 2.1). Moreover, it ensures full compatibility with the belongingness framework.

The LVAS strategy incurs a higher generation cost for our use case, i.e., multi-block documents. Specifically, a traditional signature must be computed for each individual block before they are aggregated into a single signature. As such, the number of signatures performed grows linearly with the number of blocks n. While aggregating n signatures is efficient, the cumulative cost of performing multiple signatures exceeds that of MTSS.

PV-MAC introduces a dual-layer signature process, including the generation of a hierarchical secret sharing MAC and a prefix-verifiable MAC. This procedure, which is optimal for fog-based architectures, results in higher overall signing latency. Thus, although the mechanism enables a great workload distribution, its signature generation cost is high.

Storage efficiency The LVAS scheme exhibits the smallest signature storage required among the strategies analyzed. Regardless of the number of signed blocks n, the aggregation procedure produces a single constant-size signature. However, partial verifications in LVAS additionally require verification hints to be provided to the verifier; such hints are generated via a specific algorithm, extracting an inclusion proof of such a message. Hints can be generated at query time, which allows minimal storage; nevertheless, such an algorithm grows linearly with n, which can be leveraged by a caching mechanism.

Belongingness instantiation with MTSS produces a self-contained signature structure not depending on external proofs for partial verification. The MTSS signature encompasses a collection of hash values, derived from the CFF structure, thus growing linearly with t; its size grows logarithmically to n using Sperner or the construction based on polynomials over finite fields. The absence of correlating m_j to a specific hint generation simplifies both storage and retrieval.

Unlike MTSS and LVAS, which are digital signature schemes, PV-MAC operates in different settings. It is structured on operating on ephemeral MACs in a fog-based architecture. The goal is to split the storage responsibility between the fog node and the data user. Although the MACs themselves are lightweight, the use of SS-MAC introduced additional memory management complexity. Following this logic, PV-MAC authentication data is designed to be dis-

carded after verification, with transient storage, which cannot be compared with LVAS and MTSS approaches.

Cost of partial verification The belongingness execution for MTSS should be distinguished by the server and client responsibilities, as clearly demonstrated in Table 3. On the server side (Step 2), it must identify a suitable row in the CFF where the target block m_j appears; the bigger the CFF, the greater the impact of this step. After identifying the corresponding row, the server transmits some information for the client, whose most important parameter is the set of blocks M (blocks involved in the same test as m_j). The bottleneck of the communication is the set M, whereas it can be diminished by reducing the number of 1s per row in CFFs.

On the client side, Step 3, i.e., the execution of $\mathrm{BlockVer}$ algorithm, is the most demanding. This process is lightweight, as it involves a Ver execution and hash checking. Also, the greater the CFF, the greater the impact; however, the procedure is very feasible for most parameters. By contrast, partial verifications in LVAS at the client are constant-time, requiring only a single operation (assuming the hint has already been provided by the server); if not cached, the server must execute the hint generation algorithm. In PV-MAC, partial verification is distributed; while the fog node verifies a subset of cells using MACs, the data user completes the verification by checking the PMAC suffix. This procedure results in a cost that scales linearly with the number of unverified blocks (which is automatically handled by the algorithm).

Scalability The scalability of MTSS depends on the number of blocks sent over through the set M. In other words, depending on the number of 1s per row in such a CFF, this number depends on the construction used, and there is no known construction to leverage it yet [Idalino and Moura, 2022]. However, we demonstrated how large documents with large CFFs behave for the belongingness as per Table 3, and we observed that the practical performance of MTSS belongingness remains efficient.

In opposition, the LVAS scheme is more scalable, where the aggregate signature remains constant in size, and client-side verifications are independent of n. If hints must be generated on demand, the server has a slight overhead; we achieve a trade-off by server-side caching in advance, or server-side computation on demand. For the PV-MAC strategy, we have a different scalability dimension, as it distributes verification between the fog node and data users. Although each individual verification is lightweight, the overall overhead of SS-MAC and PMAC makes it scalable for medium-sized industrial IoT deployments, but not as good for large-scale datasets compared to MTSS and LVAS.

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, locating errors, and correcting them. We demonstrated how the traditional signature scheme Σ and hash function $\mathcal{H}$ affect the performance of their algorithms. Additionally, we showed different arguments to give ideal parameters for constructing CFFs, depending on how many modified blocks d the scheme is able to handle.

We explored the challenges of dividing digital documents into blocks, which is necessary to instantiate the MTSS framework. We show that different approaches are necessary for different types of blocks and present suggestions for PDFs, images, CSV, JSON, plain text, and XML. We note that complex structured files, such as hierarchical ones, take more time to parse and separate into blocks. We analyzed the performance details using different parameters in our implementation of the construction of CFFs by considering two different constructions: Sperner and polynomial; this led us to apply some techniques to soften CFF construction, such as caching.

Finally, we introduced and formalized the concept of *belongingness* in digital signatures, addressing the challenge of verifying whether a data subset is part of a signed document without requiring access to the entire message while guaranteeing the properties of integrity and authenticity. This is particularly relevant in Big Data applications where efficient verifications are required. As part of this work, we surveyed existing techniques, identifying approaches such as Locally-Verifiable Aggregate Signatures (LVAS) and Partial Verification MACs (PV-MAC) that partially or completely address the *belongingness* definition.

However, the belongingness solutions rely on multiple signature generation or industrial IoT environments, making them not completely suitable for decentralized, agnostic, verifiable settings. To overcome these limitations, we proposed a new solution based on the MTSS signature and properties, integrating it into an iterative client-server protocol. Different from existing methods, our MTSS-based belongingness framework enables long-term authenticity verification from a single self-contained signature.

We leave it as future work to explore alternative implementations to improve the construction time of CFFs. Specifically, investigate designs that are not constrained by attributes such as prime power structures. Moreover, the $\mathrm{DivideBlocks}$ algorithm for PDF documents could be improved. We examine the possibility of dividing the written content (on the visual content) of PDFs into blocks, utilizing a specific semantic analyzer that does not allow any content to overlap with others; this restricts both the user and the attacker's resources, but modifications could be located clearly. Also, we could divide the PDF visual content into graphic regions [Wang *et al.*, 2023; Koreeda and Manning, 2021] of separate chunks of some width and height that continuously comprehend a graphical part of the document and analyze the overall impact of such a solution.

Although the MTSS framework has demonstrated practical viability, certain performance limitations persist, particularly in the VCor algorithm, whose brute-force nature leads to high computational cost. Our current implementation relies on concurrent execution due to Python's constraints, but we believe that more advanced parallelization strategies could yield significant improvements. In particular, Step 1 of the Sig algorithm — which constructs the vector T by hashing combinations of blocks according to each row of the incidence matrix $\mathcal{M}$ — is inherently parallelizable, as each row can

be processed independently. Consequently, Step 3 of the Ver algorithm is also parallelizable. Furthermore, alternative correction strategies, such as heuristic-guided or dictionary-based methods, could reduce the cost of VCor.

Regarding the belongingness framework, we propose investigating constructions that minimize the number of 1s per row to reduce the number of messages exchanged between client and server in our belongingness protocol. Another direction is minimizing the effects of reading and storing the entire data structure to keep the messages on the server, potentially using unranking techniques to compute all relevant rows i for a given m_j.

Although MTSS signatures are secure, as proved in [Idalino *et al.*, 2019], it would be valuable to analyze the security of our modified protocol against potential attacks. The current work provides only a preliminary study in this regard, and further research is needed to assess its resilience comprehensively. Additionally, evaluating the impact of network interference on the performance of the protocol would be advisable.

Declarations

Funding

GZ was partially supported by the Fundacao de Amparo a Pesquisa e Inovacao do Estado de Santa Catarina (FAPESC), Edital 62/2024.

Authors' Contributions

PA contributed to the conception of this study. JM contributed to the overall supervision of the work. GZ and TI contributed to the writing, reviewing, and editing process, as well as the investigation of evidence collection. AK is the main contributor, performing the experiments and writing this manuscript. All authors read and approved the final manuscript.

Competing interests

The authors declare that they have no competing interests.

Availability of data and materials

The datasets and software generated during the current study are available in https://github.com/AnthonyKamers/mtss-belongingness.

References

Bellare, M. and Rogaway, P. (2004). Code-based game-playing proofs and the security of triple encryption. Cryptology ePrint Archive, Paper 2004/331. Available at:https://eprint.iacr.org/2004/331.

Bilzhause, A., Pöhls, H. C., and Samelin, K. (2017). Position Paper: The Past, Present, and Future of Sanitizable and Redactable Signatures. In *International Conference on Availability, Reliability and Security (ARES '17)*, pages 87:1–87:9. DOI: 10.1145/3098954.3104058.

Bshouty, N. H. (2015). Linear Time Constructions of Some d-Restriction Problems. In Paschos, V. T. and Widmayer, P., editors, *Algorithms and Complexity (CIAC 2015)*, volume 9079 of *Lecture Notes in Computer Science*, pages 74–88. DOI: 10.1007/978-3-319-18173-8_5.

Chen, H., Zhou, H., Yu, J., Wu, K., Liu, F., Zhou, T., and Cai, Z. (2021). Trusted audit with untrusted auditors: A decentralized data integrity crowdauditing approach based on blockchain. *International Journal of Intelligent Systems*, 36:6213–6239. DOI: 10.1002/int.22548.

Crawl, C. (2024). Common Crawl - Open Repository of Web Crawl Data. Available at:https://archive.is/l1IwV.

de Bonis, A. and di Crescenzo, G. (2011a). A Group Testing Approach to Improved Corruption Localizing Hashing. Available at:https://eprint.iacr.org/2011/562.pdf.

de Bonis, A. and di Crescenzo, G. (2011b). Combinatorial Group Testing for Corruption Localizing Hashing. In Fu, B. and Du, D.-Z., editors, *Computing and Combinatorics (COCOON 2011)*, volume 6842 of *Lecture Notes in Computer Science*, pages 579–591. DOI: 10.1007/978-3-642-22685-4_50.

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 (GLOBECOM 2004)*, pages 2224–2228. DOI: 10.1109/GLOCOM.2004.1378404.

Du, D. Z. and Hwang, F. K. (2000). *Combinatorial group testing and its applications*. World Scientific, 2 edition. Book.

Erdős, 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(1–2):79–89. DOI: 10.1007/BF02772959.

Füredi, Z. (1996). On r-cover-free families. *Journal of Combinatorial Theory, Series A*, 73(1):172–173. DOI: 10.1006/jcta.1996.0012.

Gargano, L., Rescigno, A. A., and Vaccaro, U. (2020). Low-weight superimposed codes and related combinatorial structures: Bounds and applications. *Theoretical Computer Science*, 806:655–672. DOI: 10.1016/j.tcs.2019.10.032.

Goodrich, M. T., Atallah, M. J., and Tamassia, R. (2005). Indexing Information for Data Forensics. In Ioannidis, J., Keromytis, A., and Yung, M., editors, *Applied Cryptography and Network Security (ACNS 2005)*, number 3531 in Lecture Notes in Computer Science, pages 206–221. DOI: 10.1007/11496137_15.

Goyal, R. and Vaikuntanathan, V. (2022). Locally Verifiable Signature and Key Aggregation. In Dodis, Y. and Shrimpton, T., editors, *Advances in Cryptology (CRYPTO 2022)*, volume 13508 of *Lecture Notes in Computer Science*, pages 761–791. DOI: 10.1007/978-3-031-15979-4_26.

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 Abe, M. and Gligor, V. D., editors, *ACM Symposium on InformAtion, Computer and Communications Security (ASIACCS '08)*, pages 353–362. DOI: 10.1145/1368310.1368362.

Han, H., Shiwakoti, R. K., Jarvis, R., Mordi, C., and Botchie, D. (2023). Accounting and auditing with blockchain technology and artificial intelligence: A literature review. *International Journal of Accounting Information Systems*, 48:100598. DOI: 10.1016/j.accing.20222.100598.

Idalino, T. B. and Moura, L. (2022). 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 (NADCC*

2022), volume 86 of *Fields Institute Communications*, pages 195–239. DOI: 10.1007/978-3-031-48679-1_11.

Idalino, T. B., Moura, L., and Adams, C. (2019). Modification Tolerant Signature Schemes: Location and Correction. In Hao, F., Ruj, S., and Gupta, S. S., editors, *Progress in Cryptology (INDOCRYPT 2019)*, volume 11898 of *Lecture Notes in Computer Science*, pages 23–44. DOI: 10.1007/978-3-030-35423-7_2.

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. DOI: 10.1016/j.ipl.2015.02.014.

ISO/TC 171/SC 2 (2008). Document management — Portable document format — Part 1: PDF 1.7. Standard 32000-1:2008, International Organization for Standardization. Available at:https://opensource.adobe.com/dc-acrobat-sdk-docs/pdfstandards/PDF32000_2008.pdf.

Johnson, D. (2021). PDF's popularity online. Available at:https://archive.is/6ZqKK.

Johnson, R., Molnar, D., Song, D., and Wagner, D. (2002). Homomorphic signature schemes. In Preneel, B., editor, *Topics in Cryptology (CT-RSA 2002)*, number 2288 in Lecture Notes in Computer Science, pages 244–262. DOI: 10.1007/3-540-45760-7_17.

Kamers, A. B., de Oliveira Abel, P., Idalino, T. B., Zambonin, G., and Martina, J. E. (2024). Practical algorithms and parameters for modification-tolerant signature scheme. In Santin, A. and Machado, R., editors, *Brazilian Symposium on Information and Computational Systems Security (SBSeg 2024)*, pages 522–537. DOI: 10.5753/sbseg.2024.241677.

Koreeda, Y. and Manning, C. D. (2021). Capturing Logical Structure of Visually Structured Documents with Multimodal Transition Parser. pages 144–154. DOI: 10.18653/v1/2021.nllp-1.15.

Laurie, B., Messeri, E., and Stradling, R. (2021). Certificate Transparency Version 2.0. (9162). DOI: 10.17487/RFC9162.

Lim, S. and Lee, H.-S. (2011). A Short and Efficient Redactable Signature Based on RSA. *ETRI Journal*, 33(4):621–628. DOI: 10.4218/etrij.11.0110.0530.

Lu, Z., Xue, Q., Zhang, T., Cai, J., Han, J., He, Y., and Li, Y. (2024). Locally verifiable approximate multi-member quantum threshold aggregation digital signature scheme. *Computer Communications*, 228:107934:1–107934:13. DOI: 10.1016/j.comcom.2024.107934.

Luo, D. (2024). Modification-Tolerant Signature Schemes using Combinatorial Group Testing: Theory, Algorithms, and Implementation. Master's thesis, University of Ottawa. Available at:https://ruor.uottawa.ca/items/aa615ffa-bb91-4f99-92ba-abb54501f9e6.

Mlynkova, I., Toman, K., and Pokornỳ, J. (2006). Statistical Analysis of Real XML Data Collections. In Lakshmanan, L. V. S., Roy, P., and Tung, A. K. H., editors, *International Conference on Management of Data (COMAD 2006)*, pages 15–26. Available at:https://scispace.com/papers/statistical-analysis-of-real-xml-data-collections-1b8kqz1zio.

NIST (2016). Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process. Technical report, National Institute of Standards and Technology. Available at:https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf.

NIST (2023). Module-Lattice-Based Digital Signature Standard. (204). DOI: 10.6028/nist.fips.204.

Porat, E. and Rothschild, A. (2011). Explicit Nonadaptive Combinatorial Group Testing Schemes. *IEEE Transactions on Information Theory*, 57(12):7982–7989. DOI: 10.1109/TIT.2011.2163296.

Pöhls, H. C. (2018). *Increasing the Legal Probative Value of Cryptographically Private Malleable Signatures*. PhD thesis, Universität Passau. Available at:https://opus4.kobv.de/opus4-uni-passau/files/582/Poehls_Thesis_Final.pdf.

Rescigno, A. A. and Vaccaro, U. (2023). Bounds and algorithms for generalized superimposed codes. *Information Processing Letters*, 182:106365:1–106365:5. DOI: 10.1016/j.ipl.2023.106365.

Rodríguez, C., Baez, M., Daniel, F., Casati, F., Trabucco, J. C., Canali, L., and Percannella, G. (2016). REST APIs: A Large-Scale Analysis of Compliance with Principles and Best Practices. In Bozzon, A., Cudre-Maroux, P., and Pautasso, C., editors, *Web Engineering (ICWE 2016)*, volume 9671 of *Lecture Notes in Computer Science*, pages 21–39. DOI: 10.1007/978-3-319-38791-8_2.

Ruszinkó, M. (1994). On the upper bound of the size of the r-cover-free families. *Journal of Combinatorial Theory, Series A*, 66(2):302–310. DOI: 10.1016/0097-3165(94)90067-1.

Sperner, E. (1928). Ein Satz über Untermengen einer endlichen Menge. *Mathematische Zeitschrift*, 27:544–548. DOI: 10.1007/BF01171114.

Steinfeld, R., Bull, L., and Zheng, Y. (2001). Content Extraction Signatures. In Kim, K., editor, *Information Security and Cryptology (ICISC 2001)*, number 2288 in Lecture Notes in Computer Science, pages 285–304. DOI: 10.1007/3-540-45861-1_22.

Wang, J., Krumdick, M., Tong, B., Halim, H., Sokolov, M., Barda, V., Vendryes, D., and Tanner, C. (2023). A Graphical Approach to Document Layout Analysis. In Fink, G. A., Jain, R., Kise, K., and Zanibbi, R., editors, *Document Analysis and Recognition (ICDAR 2023)*, volume 14191 of *Lecture Notes in Computer Science*, pages 53–69. DOI: 10.1007/978-3-031-41734-4_4.

Wei, R. (2006). On Cover-Free Families. DOI: 10.48550/arXiv.2303.17524.

Xiong, E. (2020). Why PDF Technology Is More Relevant Than Ever. Available at:https://archive.is/CBLC0.

Yan, H., Hu, H., and Ye, Q. (2023). Partial message verification in fog-based industrial Internet of things. *Computers & Security*, 135:103530:1–103530:11. DOI: 10.1016/j.cose.2023.103530.

A Blocks representation

XML blocks representation

Original XML:


```xml
<alert>
  <pluginId>40012</pluginId>
  <alert>Cross Site Scripting</alert>
  <instances>
    <instance>
      <uri>https://example.com/search</uri>
      <method>GET</method>
      <param>q</param>
      <evidence>&lt;script&gt;alert(1)&lt;/script&gt;</evidence>
      <attack>&lt;script&gt;alert(1)&lt;/script&gt;</attack>
      <messageId>42</messageId>
      <source>
        <origin>Passive Scanner</origin>
        <timestamp>2024-11-22T10:42:33Z</timestamp>
      </source>
    </instance>
  </instances>
</alert>
```


Generated blocks:

```
1|alert
2|pluginId|40012
2|alert|Cross Site Scripting
2|instances
3|instance
4|uri|https://example.com/search
4|method|GET
4|param|q
4|evidence|<script>alert(1)</script>
4|attack|<script>alert(1)</script>
4|messageId|42
4|source
5|origin|Passive Scanner
5|timestamp|2024-11-22T10:42:33Z
```

Figure 6. Example of how an XML response is evaluated and structured into hierarchical blocks using DivideBlocks.

JSON blocks representation

Original JSON:


```json
{
  ”alert”: {
    ”pluginId”: ”40012”,
    ”alert”: ”Cross Site Scripting”,
    ”instances”: [
      {
        ”uri”: ”https://example.com/search”,
        ”method”: ”GET”,
        ”param”: ”q”,
        ”evidence”: ”<script>alert(1)</script>”,
        ”attack”: ”<script>alert(1)</script>”,
        ”messageId”: 42,
        ”source”: {
          ”origin”: ”Passive Scanner”,
          ”timestamp”: ”2024-11-22T10:42:33Z”
        }
      }
    ]
  }
}
```


Generated blocks:

```
1|alert
2|pluginId|40012
2|alert|Cross Site Scripting
2|instances
4|uri|https://example.com/search
4|method|GET
4|param|q
4|evidence|<script>alert(1)</script>
4|attack|<script>alert(1)</script>
4|source
5|origin|Passive Scanner
5|timestamp|2024-11-22T10:42:33Z
```

Figure 7. Example of how a JSON response is evaluated and structured into hierarchical blocks using DivideBlocks.