Verifiable Pattern Matching on Outsourced Texts ⋆

Dario Catalano, Mario Di Raimondo, and Simone Faro

Dipartimento di Matematica e Informatica, Università di Catania, Italy.
{catalano,diraimondo,faro}@dmi.unict.it

Abstract. In this paper we consider a scenario where a user wants to outsource her documents to the cloud, so that she can later reliably delegate (to the cloud) pattern matching operations on these documents. We propose an efficient solution to this problem that relies on the homomorphic MAC for polynomials proposed by Catalano and Fiore in [14]. Our main contribution are new methods to express pattern matching operations (both in their exact and approximate variants) as low degree polynomials, i.e. polynomials whose degree solely depends on the size of the pattern. To better assess the practicality of our schemes, we propose a concrete implementation that further optimizes the efficiency of the homomorphic MAC from [14]. Our implementation shows that the proposed protocols are extremely efficient for the client, while remaining feasible at server side.

1 Introduction

Imagine that Alice wants to store all her data on the cloud in a way such that she can later delegate, to the latter, basic computations on this data. In particular, Alice wants to be able to do this while retaining some key properties. First, the cloud should not be able to fool Alice by sending back wrong outputs. Specifically, the cloud should be able to provide a "short" (i.e. much shorter than a mere concatenation of the inputs and the output) proof that the output it computed is correct. Second, Alice should be able to check this proof without having to maintain a local copy of her data. In other words, the verification procedure should not need the original data to work correctly. An elegant solution to this problem comes from the notion of homomorphic authenticators. Informally, homomorphic authenticators are like their standard (non-homomorphic) counterparts but come equipped with a (publicly executable) evaluation algorithm that allows to obtain valid signatures on messages resulting from computing on previously signed messages. Slightly more in detail, the owner of a dataset $\{m_1, \ldots, m_\ell\}$ uses her secret key sk to produce corresponding authenticating tags $(\sigma_1, \ldots, \sigma_\ell)$ which are then stored on the cloud together with $\{m_1, \ldots, m_\ell\}$. Later, the server can (publicly) compute $m = f(m_1, \ldots, m_\ell)$ together with a succinct tag σ certifying that m is the correct output of the computation f. A nice feature of

⋆ A full version of this paper is available at http://www.dmi.unict.it/diraimondo/uploads/papers/vpm-full.pdf

homomorphic authenticators is that, as required above, the validity of this tag can be verified *without* having to know the original dataset.

Homomorphic authenticators turned out to be useful in a variety of settings and have been studied in several flavors. Examples include homomorphic signatures for linear and polynomial functions [10, 11], redactable signatures [31], transitive signatures and more [36, 39].

Our Contribution. In this paper we consider the setting where Alice wants to reliably delegate the cloud to perform pattern matching operations (both in their exact and approximate flavors) on outsourced text documents. While in principle this problem can be solved by combining (leveled) fully homomorphic signatures [29] and well known pattern matching algorithms (e.g. [33]), our focus here is on *efficient*, possibly practical, solutions. To achieve this, we develop new pattern matching algorithms specifically tailored to cope well with the very efficient homomorphic MAC solution from [14]. Our methods are very simple and allow to represent several text processing operations via (relatively) low degree polynomials[1]. Specifically, our supported functionalities range from counting the number of exact (or approximate) occurrences of a string in a text to finding the n-th occurrence of a pattern (and its position).

Slightly more in detail, our basic idea is to use the homomorphic MAC for polynomials from [14] to authenticate the texts one wishes to outsource, in a bit by bit fashion. Very informally this can be done as follows. If Alice wants to outsource her file `grades`, denoting with b_i the i-th bit of `grades`, she proceeds by first producing a MAC σ_i, for each b_i and then storing `grades` together with all the σ_i's on the cloud. Later, when Alice delegates a computation f to the cloud, she gets back an output z and a proof of correctness π that, by the properties of homomorphic authenticators, can be verified without having to maintain a copy of the data locally.

The catch with this solution is that, in order to be any practical, f has to be a low degree arithmetic circuit. This is because a drawback of the construction from [14] is that the size of π grows linearly with the degree d of the circuit[2].

To address this issue we observe that, when dealing with bits, relevant pattern matching functionalities can be expressed via polynomials of degree, at most, $2m$ where m is the bit-size of the pattern. To briefly illustrate the ideas underlying our techniques, let us focus on the case of exact pattern matching. There, the key observation is that checking if a pattern x, of size m, occurs in a text y, of size n, can be done via the following easy steps. First, one considers all the $(n - m + 1)$ possible substrings w of y of size m. Next, for each such w, one

[1] In particular the degree of these polynomials solely depends on the size of pattern string and is independent of the size of the texts

[2] Notice that in [14] a solution where the size of π can be made independent of d is also proposed. This solution however is computationally much less efficient as it imposes larger parameters.

checks equality with x via the following simple formula

$$\prod_{i=0}^{m-1} (2x_i w_i + 1 - x_i - w_i) \tag{1}$$

which is 1 if and only if all bits of x and w are equal (and 0 otherwise). Thus, x appears in y if at least one such products is non zero. This can be tested by summing the products corresponding to all possible w and checking if the result is different than zero.

Dynamic Polynomials. Notice however that, for large n, a naive application of the technique above might result in a prohibitively expensive computation for the server[3], as the latter would need to first compute and then add $\mathcal{O}(n)$ polynomials of degree $2m$.

To overcome this limitation we observe that a more careful encoding of the computation at server side, can drastically improve performances. The key point here is that, for a given pattern x, the server can reduce its costs by adapting the computation of the formula in (1) according to the bits of pattern x. Specifically, the formula (1) can be rewritten as

$$\prod_{i=0}^{m-1} (x_i w_i + (1 - x_i)(1 - w_i)) \tag{2}$$

which can be computed in m steps my using the following procedure

$P = 1$
FOR $i = 0$ TO $m - 1$ DO
  IF $(x_i = 0)$ $P \leftarrow P \cdot (1 - w_i)$
  ELSE $P \leftarrow P \cdot w_i$
RETURN P

Thus, for each queried pattern, the computed formula is dynamically adapted to the pattern. This leads to computations which are both simpler and more efficient than those induced by (1). Our tests show that this simple observation allows to reduce the computational costs of the server by a (rough) -71% !

Evaluation over Samples and Experimental Results. As already hinted above, to better assess the efficiency of our solutions we ran extensive experiments. In order to gain better performances we further optimized our techniques as follows. First, as already suggested in [14], we adopt Fast Fourier Transform (FFT) to speed up multiplications of polynomials. Inspired by FFT, we also propose an alternative strategy, named "evaluation over samples", where the whole evaluation is performed representing the polynomials via set of samples

[3] Indeed, our first implementations show that this cost can quickly become unbearable even for texts of few thousands characters.

(rather than via their coefficients). This further simplifies the implementation of polynomial multiplication and, for the case of low degree polynomials, provides an additional speed up. Finally, we note that, using some basic precomputation at client side (see [8]), verification costs can be made essentially negligible.

Our experiments show that our optimized implementations are extremely fast for the client while remaining feasible for the server. In terms of concrete numbers, our tests show that it is possible to count the exact occurrences of a 4 characters pattern in a text of 10 KiB in about 4 seconds with a proof of 528 bytes verifiable in just 300 ms. With a bigger text of 100 KiB the evaluation time raises roughly to 38 seconds. The usage of large patterns sensibly slows down the evaluation process (i.e. the costs for the server). We remark that, for a fixed pattern size, the evaluation costs for the server grow linearly with n (i.e. the size of the text). This means that for very large n our protocols, while feasible, cannot be considered practical anymore.

Related work. The problem of computing reliably on outsourced data can be solved in principle using short non interactive arguments of knowledge on authenticated data (AD-SNARKs) [7]. Such a solution would allow lower verification costs (i.e. independent from the size of the computed circuit). The main disadvantage of AD-SNARKs, with respect to our solution, is that they require much more complex machinery (thus making the costs for the server even more prohibitive). Moreover, even without considering efficiency, our homomorphic-authenticators based solution is preferable for at least two reasons. First, it requires shorter parameters: known AD-SNARKs [7] require evaluation/verification keys that grow significantly with the size of the supported circuits. Also, our solution requires only standard, falsifiable assumptions.

The questions considered in this paper share some similarities with those addressed by Verifiable Computation (VC)[26]. There, a client wants to outsource some computationally intensive task and still be able to quickly verify the correctness of the received result. Typically, VC schemes assume that the input remains available to the verifier. In our context, on the other hand, the difficulty comes from the fact that the (not necessarily complex) task involves data not locally available to the client.

The notion of homomorphic MAC was first considered (in the setting of linear functions) by [1] and later extended to more general functionalities in [28, 14, 8, 15] In the asymmetric setting the idea of homomorphic signature was first proposed by Desmedt [23] and later refined by Johnson *et al.* [31]. Starting from the work of Boneh *et al.* [10], several other papers further studied this notion both in the standard model [4, 19, 5, 20, 25, 6, 18] and in the random oracle model [27, 12, 11, 16, 17]. Beyond linear functions, Boneh and Freeman in [11] proposed an homomorphic signature scheme for constant degree polynomials. This result was later improved by Catalano, Fiore and Warinschi [21] and, more recently, by Gorbunov *et al.* [29]. This latter construction provides the first realization of a (leveled) fully homomorphic signature scheme. See [13] for a survey on homomorphic authenticators.

Polynomial encodings have been extensively studied in past. Among others, we recall the works by Applebaum *et al.* [3, 2] on randomized encodings.

Other related work. The string matching problem is one of the most fundamental problems in computer science. It consists in finding all the occurrences of a given pattern x of length m, in a text y of length n. The worst case time complexity of string matching problem is $\mathcal{O}(n+m)$, and was achieved for the first time by the well known Knuth-Morris-Pratt algorithm [33]. However the most efficient solutions to the problem in the average case have an $\mathcal{O}(nm)$ worst case time complexity [24]. In the approximate string matching problem we allow the presence of errors in the occurrences of the pattern in the text. Specifically we are interested in the string matching problem with δ errors, where at most δ substitutions of characters are allowed in order to make the pattern occur in the text. Solution to both exact and approximate string matching problems are based on comparisons of characters [33], deterministic finite state automata [22], simulation of non-deterministic finite state automata [9] and filtering methods [32]. In this paper we are interested in solving the string matching problem by using polynomial functions. To our knowledge this is the first time the string matching problem is defined in polynomial form.

In [37] Papadopoulos *et al.* propose an efficient solution for an outsourced pattern matching scenario similar to the one considered here. Their idea combines suffix trees with cryptographic accumulators. The resulting proofs have size comparable to ours but, thanks to an heavy pre-processing over the outsourced texts[4], they can be generated very efficiently. We note also that this preprocessing step is not update friendly: after the text is updated it becomes necessary to re-create the whole suffix tree. Our solution is slightly better than this as, for the specific case of append-updates it does not require any recomputations for the original tags.

Road Map. In Section 2 we recall the efficient homomorphic MAC scheme from [14]. Our new pattern matching algorithms are presented in Section 3, while the details of the proposed implementation together with relevant experimental results are given in Section 4.

2 Homomorphic MACs

In this section we briefly recall the construction of homomorphic MACs from [14] that is going to be used in our constructions. For details not discussed here we defer the reader to the original paper. Intuitively, the constructions proposed in [14], are given in the setting of *labeled programs* [28]. To authenticate a computation f one authenticates its inputs $m_1, \ldots m_n$ by also specifying corresponding labels $\tau_1 \ldots \tau_n$. A label can be seen as an index of a database record or, simply,

[4] Moreover this pre-processing has to be done by the text owner (the weak client in our scenario) and cannot be delegated to the untrusted cloud server.

as a name given to identify the (outsourced) input. In the application considered in this paper a label might simply be the name of the document followed by an indexing of its characters (bits). For example, each bit b_i of the documents `exams` could simply be the string $\tau_i = \texttt{exams}||\texttt{i}$ (here $||$ denotes concatenation).

The combination of f and the labels is a labeled program $\mathcal{P}$, that is what is later executed by the cloud. For the case of pattern matching applications, labeled programs are used as follows. When outsourcing a text document T to the cloud, the client proceeds as follows. First she computes a MAC of T, by authenticating each bit b_i of T using its corresponding label τ_i. Denoting with σ_{b_i} the MAC corresponding to the i-th bit of T, the client stores $(T, \sigma_{b_1}, \ldots, \sigma_{b_{|T|}})$ on the cloud.

As in [14], we consider circuits where additive gates do not get inputs labeled by constants. We stress that adding such gates can be done easily, as one can adopt an equivalent circuit where a special variable/label for the value 1 is added. A MAC of 1 is also added to the public parameters. It is worth mentioning the fact that the construction given below does not provide succinct authenticating tags, if the number of multiplications performed is too high. This is because the size of the tag grows with the degree d of the arithmetic circuit one wants to authenticate. In our case this is not going to be a problem as (see Section 3) d is bounded by the size of the pattern.

The following description is taken (almost) verbatim from [14]

$\mathsf{KeyGen}(1^\lambda)$**.** Let p be a prime of roughly λ bits. Choose a seed K of a pseudo-random function $F_K : \{0,1\}^* \to \mathbb{Z}_p$ and a random value $x \xleftarrow{\$} \mathbb{Z}_p$. Output $\mathsf{sk} = (K, x), \mathsf{ek} = p$ and let the message space $\mathcal{M}$ be $\mathbb{Z}_p$.

$\mathsf{Auth}(\mathsf{sk}, \tau, m)$**.** To authenticate a message $m \in \mathbb{Z}_p$ with label $\tau \in \{0,1\}^\lambda$, compute $r_\tau = F_K(\tau)$, set $y_0 = m$, $y_1 = (r_\tau - m)/x \bmod p$ and output $\sigma = (y_0, y_1)$. Thus, y_0, y_1 are the coefficients of a degree-1 polynomial $y(z)$ with the special property that it evaluates to m on the point 0 ($y(0) = m$), and it evaluates to r_τ on a hidden random point x ($y(x) = r_\tau$).

Tags σ are seen as polynomials $y \in \mathbb{Z}_p[z]$ of degree $d \geq 1$ in some (unknown) variable z, i.e., $y(z) = \sum_i y_i z^i$.

$\mathsf{Eval}(\mathsf{ek}, f, \boldsymbol{\sigma})$**.** The homomorphic evaluation algorithm takes as input the evaluation key $\mathsf{ek} = p$, an arithmetic circuit $f : \mathbb{Z}_p^n \to \mathbb{Z}_p$, and a vector $\boldsymbol{\sigma}$ of tags $(\sigma_1, \ldots, \sigma_n)$.

Eval proceeds gate-by-gate as follows. At each gate g, given two tags σ_1, σ_2 (or a tag σ_1 and a constant $c \in \mathbb{Z}_p$), it runs the algorithm $\sigma \leftarrow \mathsf{GateEval}(\mathsf{ek}, g, \sigma_1, \sigma_2)$ described below that returns a new tag σ, which is in turn passed on as input to the next gate in the circuit.

When the computation reaches the last gate of the circuit f, Eval outputs the tag vector σ obtained by running GateEval on such last gate.

To complete the description of Eval we describe the subroutine GateEval.

- $\mathsf{GateEval}(\mathsf{ek}, g, \sigma_1, \sigma_2)$. Let $\sigma_i = \boldsymbol{y}^{(i)} = (y_0^{(i)}, \ldots, y_{d_i}^{(i)})$ for $i = 1, 2$ and $d_i \geq 1$ (see below for the special case when one of the two inputs is a constant $c \in \mathbb{Z}_p$).

  If $g = +$, then:

- let $d = \max(d_1, d_2)$. Here we assume without loss of generality that $d_1 \geq d_2$ (i.e., $d = d_1$).
- Compute the coefficients $(y_0, \ldots, y_d)$ of the polynomial $y(z) = y^{(1)}(z) + y^{(2)}(z)$. This can be efficiently done by adding the two vectors of coefficients, $\boldsymbol{y} = \boldsymbol{y}^{(1)} + \boldsymbol{y}^{(2)}$ ($\boldsymbol{y}^{(2)}$ is eventually padded with zeroes in positions $d_1 ... d_2$).

If $g = \times$, then:

- let $d = d_1 + d_2$.
- Compute the coefficients $(y_0, \ldots, y_d)$ of the polynomial $y(z) = y^{(1)}(z) * y^{(2)}(z)$ using the convolution operator $*$, i.e., $\forall k = 0, \ldots, d$, define $y_k = \sum_{i=0}^{k} y_i^{(1)} \cdot y_{k-i}^{(2)}$.

If $g = \times$ and one of the two inputs, say σ_2, is a constant $c \in \mathbb{Z}_p$, then:

- let $d = d_1$.
- Compute the coefficients $(y_0, \ldots, y_d)$ of the polynomial $y(z) = c \cdot y^{(1)}(z)$.

Return $\sigma = (y_0, \ldots, y_d)$.

Notice that the size of a tag grows only after the evaluation of a multiplication gate (where both inputs are not constants). It is not hard to see that after the homomorphic evaluation of a circuit f, it holds $|\sigma| = d + 1$, where d is the degree of f.

$\mathsf{Ver}(\mathsf{sk}, m, \mathcal{P}, \sigma)$. Let $\mathcal{P} = (f, \tau_1, \ldots, \tau_n)$ be a labeled program, $m \in \mathbb{Z}_p$ and $\sigma = (y_0, \ldots, y_d)$ be a tag for some $d \geq 1$. Verification proceeds as follows:

- If $y_0 \neq m$, then output 0 (reject). Otherwise continue as follows.
- For every input wire of f with label τ compute $r_\tau = F_K(\tau)$.
- Next, evaluate the circuit on $r_{\tau_1}, \ldots, r_{\tau_n}$, i.e., compute $\rho \leftarrow f(r_{\tau_1}, \ldots, r_{\tau_n})$, and use x to check whether the following equation holds:

$$\rho = \sum_{k=0}^{d} y_k x^k \tag{3}$$

If this is true, then output 1. Otherwise output 0.

In [14] it is proved that the scheme above is secure under the sole assumption that pseudorandom functions exist.

3 String Matching Using Polynomial Functions

In this section we describe our new pattern matching solutions. These are specifically tailored to work nicely with the practical homomorphic MACs from [14]. We start by describing a simple methodology to count the number of exact occurrences of a pattern in a text. Next, we describe how to modify this procedure to encompass other cases.

Let X be the input pattern of length M, and let Y be the input text of length N, both over the same alphabet Σ of size σ. We use the symbol X_i to indicate the $(i+1)$-th character of X, with $0 \leq i < m$. Moreover we use the

symbol $X[i..j]$ to indicate the substring of X starting at position i and ending at position j (included), where $0 \leq i \leq j < n$. We say that X has an occurrence in Y at position j if $X = Y[j..j+m-1]$.

Our methods performs computation using the bitwise representation of the input strings. To this purpose, observe that each character in Σ can be represented using $\log(\sigma)$ bits. For instance each character in the set of 256 elements of the ASCII table can be represented using 8 bits. Let x and y be bitwise representation of X and Y, respectively. We use m to indicate the length of x and n for the length of y, so that $m = M\log(\sigma)$ and $n = N\log(\sigma)$. Moreover $x_i, y_j \in \{0,1\}$, for each $0 \leq i < m$ and $0 \leq j < n$.

In the following sections we will describe string matching problems in terms of functions, where the input strings play the role of variables. Additional relevant definitions will be introduced where needed. Proofs to lemmas and theorems stated below are deferred to the full version of this paper.

3.1 Counting the Number of Exact Occurrences of a String.

In this section we address the problem of counting the number of exact occurrences of a string X of size M in a string Y of size N. We recall that a string X has an exact occurrence at position j of Y if and only if $X = Y[j..j+N-1]$. More formally the problem of counting all exact occurrences of a string can be defined as the problem of computing the cardinality of the set

$$\{j \ : \ 0 \leq j < N \text{ and } X = Y[j..j+M-1]\}$$

When both strings are defined over the binary alphabet $\Sigma = \{0,1\}$, comparisons between strings and characters can be represented as polynomials. For instance we can use the polynomial function $(2ab + 1 - a - b)$ for computing comparison between two given binary values $a, b \in \{0,1\}$.

Formally we have

$$2ab + 1 - a - b = \begin{cases} 1 & \text{if } \ a = b \\ 0 & \text{otherwhise} \end{cases} \tag{4}$$

Specifically we come up with the following definition for a polynomial which count the number of occurrences X in Y, using their bitwise representations, x and y. For the sake of clarity and brevity we will use in the following the symbol $y_{(i,j)}$ to indicate the character $y[j\log(\sigma) + i]$.

Definition 1 (Exact matches function).
Let X be a pattern of length M, and let Y be a text of length N, both over the same alphabet Σ of size σ. Let x and y be their bitwise representations, of length m and n, respectively. Then we can compute the number of exact occurrences of X in Y by using the polynomial function $\alpha(X,Y)$ defined as

$$\alpha(X,Y) = \sum_{j=0}^{N-M} \left(\prod_{i=0}^{m-1} \left(2x_i y_{(j,i)} + 1 - x_i - y_{(j,i)}\right) \right) \tag{5}$$

The function $\alpha(X,Y)$ defined above requires $\mathcal{O}(NM\log(\sigma))$ multiplications while the resulting polynomial has degree $2m = 2M\log(\sigma)$. When computing such polynomial function we are able to retrieve the number of occurrences of X in Y, but we are not able to know the positions of such occurrences. Theorem 1 given below proves the correctness of the function given in (5). We first prove the following technical lemma which defines a method for comparing two binary strings with the same length.

Lemma 1. *Let $x = x_0x_1..x_{m-1}$ and $w = w_0w_1..w_{m-1}$ be two strings of length m, both over the binary alphabet $\Sigma = \{0,1\}$. Then we have that*

$$x = w \Leftrightarrow \prod_{i=0}^{m-1} (2x_iw_i + 1 - x_i - w_i) = 1 \tag{6}$$

□

Theorem 1. *Given a pattern X, of length M, and a text Y, of length N, both over the binary alphabet Σ of size σ, let x and y their bitwise representations, of length m and n, respectively. Then the exact matches polynomial function given in Definition 1 computes correctly the number of occurrences of X in Y. Formally*

$$\alpha(X,Y) = \left|\left\{j \ : \ 0 \le j \le N - M \text{ and } X = Y[j..j+M-1]\right\}\right|$$

□

3.2 Finding the Positions of All Occurrences

In many applications it is required to find the positions of the occurrence of the pattern X in Y. Let $\pi(X,Y,j)$ be initial position of the j-th occurrence of X in Y, with $i > 0$. We assume that $\pi(X,Y,j) = \infty$ if the number of occurrences of X in Y is less than i. The position of the first occurrence (i.e. $\pi(X,Y,1)$) can be obtained by asking the server to compute such position, say p_1, and subsequently to verify if such information is correct. Specifically we have that

$$\pi(X,Y,1) = p_1 \Leftrightarrow \alpha(X,Y[0..p_1+M-2]) = 0 \text{ and } \alpha(X,Y[p_1..p_1+M-1]) = 1$$

If $\pi(X,Y,1) = \infty$, indicating that no occurrence of X is contained in Y, we can verify such information by computing $\alpha(X,Y)$. Specifically we have

$$\pi(X,Y,1) = \infty \Leftrightarrow \alpha(X,Y) = 0$$

In general, if we are interested in computing the position of all occurrences of X in Y it is possible to iterate the above procedure along the whole text Y. Let p_j be the position of the j-th occurrence of X in Y, i.e. $\pi(X,Y,j) = p_j$, for $j > 0$, and let $k = \alpha(X,Y)$ the total number of occurrences. Thus we have that $\pi(X,Y,j) = \infty$ for all $j > k$.

It turns out that, for all $0 < j \le k$, $\pi(X,Y,j) = p_j$ if and only if we have $\alpha(X,Y[p_{j-1}+1..p_j+M-2]) = 0$ and $\alpha(X,Y[p_j..p_j+M-1]) = 1$. Moreover, for $j > k$, we have that $\pi(X,Y,j) = \infty$ if and only if $\alpha(X,Y[p_k+1..N]) = 0$.

3.3 Counting the Approximate Occurrences of a String

In our setting of the approximate string matching problem, given a pattern X of length M, a text Y of length N, and a bound $\delta < M$, we want to find all substring of the text of length M which differ from the pattern of, at most, δ characters. In literature such variant of the approximate string matching problem is referred as string matching with δ errors [35].

More formally we want to find all substrings $Y[j..j+M-1]$, for $0 \leq j < N$, such that

$$\left|\left\{i \ : \ 0 \leq i < M \text{ and } X_i \neq Y_{j+i}\right\}\right| \leq \delta$$

We first define the following k-error constant τ_k, for a strung of length M, which will be used later. Specifically we set

$$\tau_k = \prod_{i=1}^{k} i \times \prod_{i=k-M}^{-1} i \tag{7}$$

We next prove the following lemma which introduces a polynomial function for computing the number of mismatches between two strings of equal length. We recall that we use the symbol $y_{(j,i)}$ to indicate $y_{j\log(\sigma)+i}$.

Lemma 2 (Mismatch function). *Let X and W two strings over a common alphabet Σ of size σ. Let x and w be their bitwise representations of length $m = M\log(\sigma)$. The mismatch function $\Psi : \Sigma^M \times \Sigma^M \to \{0, 1, .., M\}$, defined as*

$$\Psi(X, W) = \sum_{j=0}^{M-1} \left[1 - \prod_{i=0}^{\log\sigma-1} \left(2x_{(j,i)}w_{(j,i)} + 1 - x_{(j,i)} - w_{(j,i)}\right)\right] \tag{8}$$

counts the number of mismatches between X and W.

□

Let us take into account the value $\tau_k(x, w)$, defined as the product of the differences between $Psi(x, w)$ and the values int he range $\{1..m\}$. Formally

$$\tau(X, W) = \prod_{i=0}^{M} \left(\Psi(X, W) - i\right). \tag{9}$$

Since $0 \leq \Psi(X, W) \leq M$, it turns out that the value of $\tau(X, W)$ is always equal to 0. In fact one (and only one) of the factors in (9) is equal to zero.

We are now ready to prove the following lemma which introduces a polynomial function for detecting if X and W differs exactly of k characters.

Lemma 3 (k-mismatch function). *Let X and W be two strings of length M over an alphabet Σ of size σ. Moreover let k an error value in $\{0, .., M\}$. Then the k-mismatch function, $\tau_k : \Sigma^m \times \Sigma^m \to \{0, 1\}$, defined as*

$$\tau_k(X, W) = \frac{1}{\tau_k} \prod_{i=0}^{k-1} \left(\Psi(X, W) - i\right) \times \prod_{i=k+1}^{M} \left(\Psi(X, W) - i\right). \tag{10}$$

is equal to 1 if X and W has k mismatches, otherwise it is equal to 0. □

Observe that the resulting polynomial for computing $\tau(X,Y)$ has degree $2m$ while the polynomial for computing τ_k has degree $(m-1)$.

The following corollary gives a method to compute the number of approximate occurrences of a given pattern X in a text Y with exactly k errors. It trivially follows from Lemma 3.

Corollary 1 (Count k errors matches function). *Given a pattern X, of length M, a text Y, of length N, and an error value $k \leq M$, we can compute the number of occurrences of X in Y with (exactly) k errors by using the function $\beta_k(X,Y)$ defined as*

$$\beta_k(X,Y) = \sum_{i=0}^{N-M} \tau_k(X, Y[i..i+M-1])$$

Finally, the following corollary introduces the function for computing the number of approximate occurrences of X in Y assuming an error bound δ. It trivially follows from Corollary 1.

Corollary 2 (Count δ-approximate matches function). *Given a pattern X, of length M, a text Y, of length N, and an error bound $\delta \leq M$, we can compute the number of occurrences of X in Y with at most δ errors by using the function $\gamma(X,Y)$ defined as*

$$\gamma(X,Y) = \sum_{k=0}^{\delta} \beta_k(X,Y) \tag{11}$$

As previously described we can also adapt such technique to find the position of the first occurrence, as the position of all occurrences of X in Y.

3.4 Using Dynamic Polynomials.

In our experimental results, using the polynomials introduced above, we observed a prohibitively expensive computation for the server, especially for large texts. It turns out, in fact, that for both exact and approximate pattern matching we need to first compute and then add $\mathcal{O}(N)$ polynomials of degree $2m$.

In this section we present a method to overcome this limitation and decrease the degree of the resulting polynomials. Specifically we observe that a more careful encoding of the computation at server side can drastically improve the performances. The key point here is that, for a given pattern X, the server can reduce its costs by adapting the computation of the polynomials according to the bits of the pattern X.

Specifically, the formulas (6) and (8) can be rewritten, respectively, as

$$\prod_{i=0}^{m-1} (x_i w_i + (1-x_i)(1-w_i)) \tag{12}$$

$$\sum_{i=0}^{M-1} \left[1 - \prod_{i=0}^{\log \sigma - 1} \left(x_{(j,i)}(1-w_{(j,i)}) + (1-x_{(j,i)}) w_{(j,i)} \right) \right] \tag{13}$$

Thus for instance, if all bits in x are equal to 0, i.e $x = 0^m$, then the polynomial in (12) is equal to $\prod_{i=0}^{m-1}(1 - w_i)$, while it is equal to $\sum_{i=0}^{m-1} w_i$ when $x = 1^m$. According to such observation the number of exact and approximate occurrences

```
EXACT-MATCHING(X, M, Y, N)
1. F = 0
2. FOR j = 0 TO N − M DO
3.    P = 1
4.    FOR i = 0 TO m − 1 DO
5.       IF (x_i = 0) THEN
6.       P ← P · (1 − y_(j,i))
6.       ELSE P ← P · y_(j,i)
7.    F ← F + P
8. RETURN F

PRODUCT-FACTORS(X, M, Y, N)
1. FOR j = 0 TO N − M DO
2.    FOR i = 0 TO M − 1 DO
3.       P[j, i] = 1
4.       FOR h = 0 TO log(σ) − 1 DO
5.          IF (x_(i,h) = 0) THEN
6.             P[j, i] ← P[j, i] · (1 − y_(j,h))
6.          ELSE P[j, i] ← P · y_(j,h)
```

```
APPROXIMATE-MATCHING(X, M, Y, N, δ)
1.  PRODUCT-FACTORS(X, M, Y, N)
2.  FOR k = 0 TO δ DO
3.     τ_k = ∏_{i=1}^{k} i · ∏_{i=k−M}^{−1} i
4.  F = 0
5.  FOR j = 0 TO n − m DO
6.     Ψ = 0
7.     FOR i = 0 TO M − 1 DO
8.        Ψ ← Ψ + (1 − P[j + i, i])
9.     FOR k = 0 TO δ DO
10.       β_k = 1
11.       FOR i = 0 TO m DO
12.          IF (i ≠ k) THEN
13.             β_k ← β_k · (Ψ − i)
14.       β_k = β_k/τ_k
15.       F = F + β_k
16. RETURN F
```

Fig. 1: Procedure EXACT-MATCHING (on the left) for computing the dynamic polynomial given in (5) and procedure APPROXIMATE-MATCHING (on the right) for computing the dynamic polynomial in (11).

of the string X in Y can be computed using the algorithms shown in Figure 1, which construct the polynomial according to the bits contained in X.

Specifically, the algorithm EXACT-STRING-MATCHING shown in Figure 1 (on the left) computes the dynamic polynomial correspondent to the function in (5) in $\mathcal{O}(NM\log(\sigma))$ time. The resulting polynomial has a degree equal to $m = M\log(\sigma)$. Similarly, the algorithm APPROXIMATE-STRING-MATCHING shown in Figure 1 (on the right) computes the dynamic polynomial correspondent to the function in (11). Procedure PRODUCT-FACTORS computes a matrix P of dimension $N \times M$ where $P[j, i]$ is 1 if $Y_j = X_i$, and 0 otherwise. Such computation is performed in time $O(NM\log(\sigma))$. The overall time complexity of procedure APPROXIMATE-STRING-MATCHING is $\mathcal{O}(NM\delta\log(\sigma))$ while the resulting polynomial has a degree equal to m.

4 Implementation Details

In this section we discuss the details of our implementation together with some optimizations. These, in particular, target both server evaluation and client verification.

Optimizations. The first optimization we consider is the usage of the dynamic polynomials technique described in Section 3.4. Beyond reducing computational costs at server side, this technique also reduces bandwidth costs *both* when the client sends a pattern query and when the server provides back the answer. In the first case, the gain comes from the fact that the pattern can be sent unauthenticated (i.e. without authenticating it bit by bit, as the basic, non dynamic, version of our technique would require). In the second case, one gains from computing a lower degree polynomial (m instead of $2m$)[5].

The second optimization (referred as "Evaluation over Samples" in our tables) works at a lower level: the way the server evaluates tags (i.e. polynomials). Recall that in our case a MAC is a polynomial with coefficients in $\mathbb{Z}_p$ and Eval essentially performs additions and multiplications of polynomials (with multiplication being the computationally most intensive operation). A naive implementation of polynomial multiplication has time complexity $\mathcal{O}(n^2)$, when starting from polynomials of degree n. It is well known, that this can be reduced to $\mathcal{O}(n \log n)$ using FFT. Very informally, FFT allows to quickly perform multiplication by temporarily switching to a more convenient representation of the starting polynomials. In particular a set of complex points is (carefully) chosen and the polynomials are computed over such points. Multiplication can now be achieved by multiplying corresponding points and then going back to the original representation via interpolation.

Inspired by this, we notice that, since we work with low degree polynomials, we can stick to a "fast" point representation the whole time, without switching representation at each multiplication, as done in FFT. Specifically, instead of representing each polynomial f via its coefficients, we keep the points $f(i_1), \ldots, f(i_\ell)$, where $i_1, \ldots, i_\ell$ are (non complex) fixed points and ℓ is large enough to perform interpolation at the end. In particular addition (multiplication) of polynomials is obtained by adding (multiplying) the corresponding points[6]. We stress that, differently than FFT we keep this alternative representation along the whole evaluation: polynomial interpolation is applied only once, to compute the final tag that the server sends back to the client. We also remark that our technique is alternative to FFT and they cannot be used together.

Our experiments show that verification at client's side is very fast (few seconds even with the largest considered texts). Still, we could reduce these costs

[5] Recall that in the homomorphic MAC scheme from [14] the size of the tags grows with the degree of the arithmetic circuit.

[6] Notice that the fact that we consider low degree polynomials is crucial here. Our technique is efficient solely because ℓ does not need to be too big to be able to interpolate correctly at the end.

even further via preprocessing. This is because the homomorphic MAC from [14] allows for a two phase verification procedure. The most expensive phase is the one that involves the computation of ρ (see Section 2). This phase, however, can be done "offline", before knowing of the answer provided by the server (in particular it can be done while waiting for the server's response). Once receiving an answer, the client can complete the residual verification procedure with a total cost of $\mathcal{O}(d)$ multiplications, where d is the degree of the tag. Our experimental results show that this on-line phase has a negligible cost of few milliseconds.

Testing environment and experiment parameters. Our code was written in C using (mainly) the GMP [30] library but also exploiting NTL [38] and gcrypt [34] codes, respectively, for a good implementation of the FFT-based polynomial multiplication and for AES (as underlying PRF). Our single-thread code was executed on a laptop equipped with a 64-bit Intel i7 6500U dual-core CPU running at 2.50 GHz speed. Given a specific experiment, the reported timing is obtained as the average value over multiple runs.

In our experiments, we first implemented the pattern matching algorithm reporting the number of exact matches of a pattern in a given text, as explained in Section 3. Then, we progressively applied the proposed optimizations in order to properly quantify the contribution added by each technique. We also implemented the approximate variant of our algorithm to test its performances. The algorithmic solutions of Section 3 producing the position of selected occurrences are clearly a mere application of previous algorithms, so no specific tests were conducted.

All the involved cryptographic tools where tuned to work with a long-term security level of 128 bits. We also implemented a (very) low security, 64 bit variant of our methods[7]. In this latter case it is possible to get an additional $20\% - 30\%$ gain in performances.

Optimization timings. A former set of experiments were carried on in order to estimate the single contribution of each considered optimization. The usage of (standard) FFT on the single tag multiplications was also included. The experiments involved a wide range of parameters (mainly varying text and pattern length). Here we report, the timings of a representative sample: a 1024 characters long text with a pattern of 8 characters. The time complexity of the evaluation step is clearly linear in the size of the text, so the performance on larger or smaller texts can be easily deduced.

For the chosen parameters, the timings for the server evaluation are reported in Table 1. It is interesting to note that evaluation over samples beats FFT only when used in conjunction with the dynamic polynomials optimization.

[7] These timings are not reported in this paper but are available upon request.

algorithm + optimizations	evaluation time (s)
"count exact occurrences" algorithm	35.585
+FFT	8.572
+evaluation over sample	15.937
+dynamic polynomials	10.012
+dynamic polynomials +FFT	3.835
+dynamic polynomials +evaluation over samples	1.424

Table 1: Evaluation of an 8 chars pattern on a 1024 chars text

Additional tests. Next, we considered the behaviour of our methods when considering different text sizes and pattern lengths[8].

We consider three possible pattern lengths: 4, 8 and 16 characters. These patterns are searched in texts of sizes: 1 KiB, 10 KiB and 100 KiB. As stated above, the linear complexity in the length of the text allows to easily deduce the behaviour with longer texts.

The timings and some bandwidth/memory measures using the considered settings are reported in Table 2. For a specific pattern size, the sampled evaluation and verification timings confirm the linearity in the text length. On the other side, it rapidly grows using longer patterns. The reported verification timings do not include the possible on-line/off-line optimization discussed before (in such a case the off line cost of verification becomes essentially the whole cost).

text (chars)	pattern	key gen.	text auth.	evaluation (ms)	verification	text tags (bytes)	proof tag
1K	4	0.107	12	408	29	256K	528
1K	8	0.107	12	1424	59	256K	1040
1K	16	0.100	12	7685	117	256K	2064
10K	4	0.100	117	4106	307	2.5M	528
10K	8	0.100	113	15263	581	2.5M	1040
10K	16	0.100	116	81383	1176	2.5M	2064
100K	4	0.100	1430	37826	3274	25M	528
100K	8	0.133	1169	151369	6431	25M	1040
100K	16	0.133	1155	788093	11717	25M	2064

Table 2: Timings and sizes of exact pattern matching with both optimizations applied

The cloud storage space for the authenticated text indicates a non-negligible fundamental factor of 1 KiB/character: it could be almost halved with a smart

[8] We stress that we focused on our optimized techniques, as they are better than the alternative solutions discussed before in essentially all settings considered here.

implementation considering that the known term of the 1-degree polynomial representing the tag is always a single bit and not a full 128 bits field element. The size of the proof reported by the server is quite small and it grows linearly with the size of the pattern.

Further experimental results on the approximate pattern matching algorithms are available in the full version of this paper.

Acknowledgements. This research was supported in part by a FIR 2014 grant by the University of Catania. Thanks to Nuno Tiago Ferreira de Carvalho for his Homomorphic MACs library[9].

References

1. Shweta Agrawal and Dan Boneh. Homomorphic MACs: MAC-based integrity for network coding. In Michel Abdalla, David Pointcheval, Pierre-Alain Fouque, and Damien Vergnaud, editors, *ACNS 09*, volume 5536 of *LNCS*, pages 292–305. Springer, June 2009.
2. Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. Computationally private randomizing polynomials and their applications. *Computational Complexity*, 15(2):115–162, 2006.
3. Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. From secrecy to soundness: Efficient verification via secure computation. In Samson Abramsky, Cyril Gavoille, Claude Kirchner, Friedhelm Meyer auf der Heide, and Paul G. Spirakis, editors, *ICALP 2010, Part I*, volume 6198 of *LNCS*, pages 152–163. Springer, July 2010.
4. Nuttapong Attrapadung and Benoît Libert. Homomorphic network coding signatures in the standard model. In Dario Catalano, Nelly Fazio, Rosario Gennaro, and Antonio Nicolosi, editors, *PKC 2011*, volume 6571 of *LNCS*, pages 17–34. Springer, March 2011.
5. Nuttapong Attrapadung, Benoît Libert, and Thomas Peters. Computing on authenticated data: New privacy definitions and constructions. In Xiaoyun Wang and Kazue Sako, editors, *ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 367–385. Springer, December 2012.
6. Nuttapong Attrapadung, Benoît Libert, and Thomas Peters. Efficient completely context-hiding quotable and linearly homomorphic signatures. In Kaoru Kurosawa and Goichiro Hanaoka, editors, *PKC 2013*, volume 7778 of *LNCS*, pages 386–404. Springer, February / March 2013.
7. Michael Backes, Manuel Barbosa, Dario Fiore, and Raphael M. Reischuk. ADSNARK: Nearly practical and privacy-preserving proofs on authenticated data. In *2015 IEEE Symposium on Security and Privacy*, pages 271–286. IEEE Computer Society Press, 2015.
8. Michael Backes, Dario Fiore, and Raphael M. Reischuk. Verifiable delegation of computation on outsourced data. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 13*, pages 863–874. ACM Press, November 2013.
9. Ricardo A. Baeza-Yates and Gaston H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, 1992.

[9] Available at `https://bitbucket.org/ntfc/cf-homomorphic-mac/`

10. Dan Boneh, David Freeman, Jonathan Katz, and Brent Waters. Signing a linear subspace: Signature schemes for network coding. In Stanislaw Jarecki and Gene Tsudik, editors, *PKC 2009*, volume 5443 of *LNCS*, pages 68–87. Springer, March 2009.
11. Dan Boneh and David Mandell Freeman. Homomorphic signatures for polynomial functions. In Kenneth G. Paterson, editor, *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 149–168. Springer, May 2011.
12. Dan Boneh and David Mandell Freeman. Linearly homomorphic signatures over binary fields and new tools for lattice-based signatures. In Dario Catalano, Nelly Fazio, Rosario Gennaro, and Antonio Nicolosi, editors, *PKC 2011*, volume 6571 of *LNCS*, pages 1–16. Springer, March 2011.
13. Dario Catalano. Homomorphic signatures and message authentication codes. In *SCN 14*, LNCS, pages 514–519. Springer, 2014.
14. Dario Catalano and Dario Fiore. Practical homomorphic MACs for arithmetic circuits. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 336–352. Springer, May 2013.
15. Dario Catalano, Dario Fiore, Rosario Gennaro, and Luca Nizzardo. Generalizing homomorphic MACs for arithmetic circuits. In Hugo Krawczyk, editor, *PKC 2014*, volume 8383 of *LNCS*, pages 538–555. Springer, March 2014.
16. Dario Catalano, Dario Fiore, Rosario Gennaro, and Konstantinos Vamvourellis. Algebraic (trapdoor) one-way functions and their applications. In Amit Sahai, editor, *TCC 2013*, volume 7785 of *LNCS*, pages 680–699. Springer, March 2013.
17. Dario Catalano, Dario Fiore, Rosario Gennaro, and Konstantinos Vamvourellis. Algebraic (trapdoor) one-way functions: Constructions and applications. *Theoretical Computer Science*, 592:143–165, 2015.
18. Dario Catalano, Dario Fiore, and Luca Nizzardo. Programmable hash functions go private: Constructions and applications to (homomorphic) signatures with shorter public keys. In *CRYPTO 2015, Part II*, LNCS, pages 254–274. Springer, August 2015.
19. Dario Catalano, Dario Fiore, and Bogdan Warinschi. Adaptive pseudo-free groups and applications. In Kenneth G. Paterson, editor, *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 207–223. Springer, May 2011.
20. Dario Catalano, Dario Fiore, and Bogdan Warinschi. Efficient network coding signatures in the standard model. In Marc Fischlin, Johannes Buchmann, and Mark Manulis, editors, *PKC 2012*, volume 7293 of *LNCS*, pages 680–696. Springer, May 2012.
21. Dario Catalano, Dario Fiore, and Bogdan Warinschi. Homomorphic signatures with efficient verification for polynomial functions. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part I*, volume 8616 of *LNCS*, pages 371–389. Springer, August 2014.
22. Maxime Crochemore and Wojciech Rytter. *Text Algorithms*. Oxford University Press, 1994.
23. Yvo Desmedt. Computer security by redefining what a computer is. NSPW, 1993.
24. Simone Faro and Thierry Lecroq. The exact online string matching problem: A review of the most recent results. *ACM Comput. Surv.*, 45(2):13, 2013.
25. David Mandell Freeman. Improved security for linearly homomorphic signatures: A generic framework. In Marc Fischlin, Johannes Buchmann, and Mark Manulis, editors, *PKC 2012*, volume 7293 of *LNCS*, pages 697–714. Springer, May 2012.
26. Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In Tal Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 465–482. Springer, August 2010.

27. Rosario Gennaro, Jonathan Katz, Hugo Krawczyk, and Tal Rabin. Secure network coding over the integers. In Phong Q. Nguyen and David Pointcheval, editors, *PKC 2010*, volume 6056 of *LNCS*, pages 142–160. Springer, May 2010.
28. Rosario Gennaro and Daniel Wichs. Fully homomorphic message authenticators. In Kazue Sako and Palash Sarkar, editors, *ASIACRYPT 2013, Part II*, volume 8270 of *LNCS*, pages 301–320. Springer, December 2013.
29. Sergey Gorbunov, Vinod Vaikuntanathan, and Daniel Wichs. Leveled fully homomorphic signatures from standard lattices. In *47th ACM STOC*, pages 469–477. ACM Press, 2015.
30. Torbjrn Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 6.1.0 edition, 2016.
31. Robert Johnson, David Molnar, Dawn Xiaodong Song, and David Wagner. Homomorphic signature schemes. In Bart Preneel, editor, *CT-RSA 2002*, volume 2271 of *LNCS*, pages 244–262. Springer, February 2002.
32. Juha Kärkkäinen and Joong Chae Na. Faster filters for approximate string matching. In *Proceedings of the Nine Workshop on Algorithm Engineering and Experiments, ALENEX 2007, New Orleans, Louisiana, USA, January 6, 2007*. SIAM, 2007.
33. Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.
34. Werner Koch and the Libgcrypt development team. *Libgcrypt*, 1.7.0 edition, 2016.
35. Gad M. Landau and Uzi Vishkin. Efficient string matching with k mismatches. *Theor. Comput. Sci.*, 43:239–249, 1986.
36. Silvio Micali and Ronald L. Rivest. Transitive signature schemes. In Bart Preneel, editor, *CT-RSA 2002*, volume 2271 of *LNCS*, pages 236–243. Springer, February 2002.
37. Dimitrios Papadopoulos, Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Practical authenticated pattern matching with optimal proof size. *Proceedings of the VLDB Endowment*, 8(7):750–761, 2015.
38. Victor Shoup. *NTL: A Library for doing Number Theory*, 9.7.1 edition, 2016.
39. Xun Yi. Directed transitive signature scheme. In Masayuki Abe, editor, *CT-RSA 2007*, volume 4377 of *LNCS*, pages 129–144. Springer, February 2007.