Journal of Discrete Algorithms 51 (2018) 39–50

Contents lists available at ScienceDirect

Journal of Discrete Algorithms

www.elsevier.com/locate/jda

Verifiable pattern matching on outsourced texts ☆

Dario Catalano, Mario Di Raimondo, Simone Faro *

Dipartimento di Matematica e Informatica, Università di Catania, Italy

ARTICLE INFO

Article history:
Available online 17 October 2018

Keywords:
String matching
Text processing
Outsourced texts
Verified computation
Design of algorithms
Experimental

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 (EuroCrypt 2013). 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 of Catalano and Fiore. 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 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, for instance, homomorphic signatures for linear and polynomial functions [5,4] and redactable signatures [19].

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

☆ This is an extended and revised version of the paper that appears in the Proceedings of the 10th Conference on Security and Cryptography for Networks (SCN 2016) [8].

* Corresponding author.
E-mail addresses: catalano@dmi.unict.it (D. Catalano), diraimondo@dmi.unict.it (M. Di Raimondo), faro@dmi.unict.it (S. Faro).

https://doi.org/10.1016/j.jda.2018.08.002
1570-8667/

can be solved by combining (leveled) fully homomorphic signatures[1] [17] and well known pattern matching algorithms (e.g. [21]), 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 [9]. Our methods are very simple and allow to represent several text processing operations via (relatively) low degree polynomials.[2] 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 [9] 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, 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.

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 [9] is that the size of π grows linearly with the degree d of the circuit.[3]

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.

Dynamic Polynomials. In order to reduce the computational load we propose an optimization that allows to simplify the used polynomial, lowering its degree, adapting it to the specific pattern. Our tests show that it permits to reduce the computational costs on the server by a (rough) 71%!

Evaluation over Samples. 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 [9], 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 a set of samples (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 [2]), verification costs can be made negligible.

Handling Long Patterns. The experimental results show that our technique becomes unpractical using long patterns. In order to overcome this problem we propose an alternative way to apply the verifiable pattern matching algorithm.

Related work. The questions considered in this paper share some similarities with those addressed by Verifiable Computation (VC) [15]. 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 task involves data not locally available to the client.

The notion of homomorphic MAC was first considered (in the setting of linear functions) in [1] and later extended to more general functionalities in [16,9,2,10]. In the asymmetric setting the idea of homomorphic signature was first formalized by Johnson et al. [19]. See [7], and references therein, for more details.

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 [21]. However the most efficient solutions to the problem in the average case have an $\mathcal{O}(nm)$ worst case time complexity [13]. 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. Solutions to both exact and approximate string matching problems are based on comparisons of characters [21], deterministic finite state automata [12], simulation of non-deterministic finite state automata [3] and filtering methods [20]. In this paper we are interested in solving the string matching problem by using polynomial functions. To our knowledge, with the exception of wildcard matching [11], this is the first time the string matching problem is defined in polynomial form.

In literature outsourced pattern matching has been considered mostly focusing on secureness [14,26] rather than on verifiability. In addition, in [24] 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 a 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 case of append-updates it does not require any recomputations for the original tags.

[1] Such powerful constructions allow to compute any functionality (so any generic pattern matching algorithm); as drawback their actual efficiency is far to be considered practical.

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

[3] Notice that in [9] 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.

[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.

2. Homomorphic MACs

In this section we briefly recall the definition of homomorphic MAC and the construction from [9] that is going to be used in our proposals. For details not discussed here we refer the reader to the original paper. A MAC (*Message Authentication Code*) allows to compute and, later, verify a tag using a secret key; a successful verification garantees on the integrity and authenticity of the message; an homomorphic MAC, given the tags associated to some messages, allows to derive the tag on some (linear) combinations of the original messages. Intuitively, the constructions proposed in [9], are given in the setting of *labeled programs* [16]. 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, 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, the label of each bit b_i of the document `exams` could be the string $\tau_i = \texttt{exams||i}$ (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, 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. The client is only required to store the secret key associated to the MAC computation: the labels will be necessary too but they can be later recomputed knowing filename and size of the uploaded texts.

2.1. Homomorphic MAC definition

More formally, a labeled program $\mathcal{P}$ is a tuple $(f, \tau_1, \ldots, \tau_n)$ where $f : \mathcal{M}^n \to \mathcal{M}$ is a function (a polynomial in our case), and the values $\tau_1, \ldots, \tau_n \in \{0,1\}^*$ are the *labels* corresponding to the inputs of f. Given the labeled programs $\mathcal{P}_1, \ldots, \mathcal{P}_t$ and some function $g : \mathcal{M}^t \to \mathcal{M}$ the *composed program* is defined as $\mathcal{P}^* = g(\mathcal{P}_1, \ldots, \mathcal{P}_t)$. This consists in evaluating the circuit g on the outputs of $\mathcal{P}_1, \ldots, \mathcal{P}_t$ respectively. The (labeled) inputs of $\mathcal{P}^*$ are the (labeled) inputs of $\mathcal{P}_1, \ldots, \mathcal{P}_t$.[5] Let $\mathcal{I}_\tau = (g_{id}, \tau)$ be the *identity program* with associated label τ, where g_{id} is the canonical identity function and $\tau \in \{0,1\}^*$ is some input label. By the above positions we notice that any program $\mathcal{P} = (f, \tau_1, \ldots, \tau_n)$ can be expressed as the composition of n identity programs $\mathcal{P} = f(\mathcal{I}_{\tau_1}, \ldots, \mathcal{I}_{\tau_n})$.

As in [9], here we consider the case of arithmetic circuits $f : \mathcal{M}^n \to \mathcal{M}$ where $\mathcal{M}$ is some finite field, e.g., $\mathbb{Z}_p$ for a prime p.

Definition 2.1 *(Homomorphic MAC).* An homomorphic MAC scheme is composed by the following algorithms:

- $\mathsf{KeyGen}(1^\lambda)$: This algorithm takes as input a security parameter λ and produces as output a secret key sk and a (public) evaluation key ek.
- $\mathsf{Auth}(\mathsf{sk}, \tau, m)$: It takes as input the secret key sk, a label τ and a message m. It outputs the tag σ.
- $\mathsf{Ver}(\mathsf{sk}, m, \mathcal{P}, \sigma)$: It takes as input the secret key sk, a message m, a labeled program $\mathcal{P} = (f, \tau_1, \ldots, \tau_n)$ and a tag σ. It outputs 0 (reject) or 1 (accept).
- $\mathsf{Eval}(\mathsf{ek}, f, \vec{\sigma})$: It takes as input the public (evaluation) key ek, a function $f : \mathcal{M}^n \to \mathcal{M}$ and a vector of tags $\vec{\sigma} = (\sigma_1, \ldots, \sigma_n)$. It outputs a new tag σ.

For the properties of homomorphic MACs we refer the reader to [9].

2.2. Homomorphic MAC construction [9]

The details of the construction for homomorphic MAC from [9] follow.
(1) $\mathsf{KeyGen}(1^\lambda)$: Let p be a prime number of λ bits. Let K be the secret key for a pseudorandom function $F_K : \{0,1\}^* \to \mathbb{Z}_p$. Finally, let $x \xleftarrow{\$} \mathbb{Z}_p$. Output $\mathsf{sk} = (K, x)$, $\mathsf{ek} = p$. The message space $\mathcal{M}$ is $\mathbb{Z}_p$.
(2) $\mathsf{Auth}(\mathsf{sk}, \tau, m)$: To authenticate $m \in \mathbb{Z}_p$ with respect to the label $\tau \in \{0,1\}^\lambda$, one computes $r_\tau = F_K(\tau)$ and sets $y_0 = m$, $y_1 = (r_\tau - m)/x \bmod p$. The output is $\sigma = (y_0, y_1)$. Notice that y_0, y_1 can be seen as the coefficients of a degree-1 polynomial $y(z)$ that evaluates to m on the point 0 ($y(0) = m$), and to r_τ on the (hidden) point x ($y(x) = r_\tau$).

In this sense, tags σ can be seen as polynomials $y \in \mathbb{Z}_p[z]$ of degree $d \geq 1$ in some, unknown, variable z. Equivalently, this means that $y(z) = \sum_i y_i z^i$.
(3) $\mathsf{Eval}(\mathsf{ek}, f, \vec{\sigma})$: The evaluation algorithm takes as input the (public) evaluation key $\mathsf{ek} = p$, an arithmetic circuit $f : \mathbb{Z}_p^n \to \mathbb{Z}_p$, and a vector $\vec{\sigma}$ of tags $(\sigma_1, \ldots, \sigma_n)$.

[5] In general, we assume that the labels are all distinct: this is because inputs sharing the same label can be "regrouped" in a single input for the new program.

GateEval(ek, g, σ_1, σ_2) **procedure**.

Let $\sigma_i = \vec{y}^{(i)} = (y_0^{(i)}, \ldots, y_{d_i}^{(i)})$ for $i = 1, 2$ and $d_i \geq 1$
If $g = +$, then:
- let $d = \max(d_1, d_2)$. For simplicity we assume that $d_1 \geq d_2$ (i.e., $d = d_1$).
- Let $(y_0, \ldots, y_d)$ be the coefficients of the polynomial $y(z) = y^{(1)}(z) + y^{(2)}(z)$. This can be efficiently computed by adding the two vectors of coefficients received as inputs, $\vec{y} = \vec{y}^{(1)} + \vec{y}^{(2)}$. Notice that $\vec{y}^{(2)}$ is padded with zeroes in positions $d_1 \ldots d_2$.

If $g = \times$, then:
- let $d = d_1 + d_2$.
- As above, let $(y_0, \ldots, y_d)$ be the coefficients of the polynomial $y(z) = y^{(1)}(z) * y^{(2)}(z)$. These are computed via the standard convolution operator $*$, i.e., $\forall k = 0, \ldots, d$, set $y_k = \sum_{i=0}^{k} y_i^{(1)} \cdot y_{k-i}^{(2)}$.

If $g = \times$ and one of the inputs, say σ_2, is a constant $c \in \mathbb{Z}_p$, then:
- let $d = d_1$.
- Let $(y_0, \ldots, y_d)$ be the coefficients of the polynomial $y(z) = c \cdot y^{(1)}(z)$.

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

Fig. 1. Detailed description of the GateEval procedure.

The algorithm proceeds in a gate-by-gate fashion, as follows. For each gate g, given the tags σ_1, σ_2 (or a tag σ_1 and some constant $c \in \mathbb{Z}_p$), it runs the GateEval algorithm (given below) which returns a new tag σ. This, in turn, will be given as input to the next gate of the circuit.

The output of Eval is thus, the tag vector obtained from the last execution of GateEval. The description of GateEval is shown in Fig. 1.

We stress that tags grow only when performing multiplications involving two (both non-constant) inputs. Notice also that the size of the tag grows linearly with the degree of the arithmetic circuit.

(4) Ver(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 $d \geq 1$. The verification procedure does the following: (i) If $y_0 \neq m$, output 0. If this is not the case, proceed as follows; (ii) For each input wire of f with corresponding label τ set $r_\tau = F_K(\tau)$; (iii) Next, compute $\rho \leftarrow f(r_{\tau_1}, \ldots, r_{\tau_n})$, and perform the following check: $\rho = \sum_{k=0}^{d} y_k x^k$. If the equation holds, output 1 else output 0.

As in [9], we consider circuits where additive gates do not get inputs labeled by constants. Indeed adding these gates can be easily done: one considers the equivalent circuits equipped with a special label/variable for the value 1. In [9] 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, specifically tailored to work nicely with the practical homomorphic MACs from [9]. We start by describing a 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, and 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 perform 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 $x_i, y_j \in \{0, 1\}$.

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.

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

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 comparing two given binary values $a, b \in \{0, 1\}$. Formally

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

Specifically we come up with the following definition for a polynomial which counts the number of occurrences of 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 $y_{(j,i)}$ to indicate the character $y[j\log(\sigma)+i]$.

Definition 3.1 *(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:

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

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 their positions. Theorem 3.1 proves the correctness of the function given in (2). We first prove the following technical lemma which defines a method for comparing two binary strings with the same length.

Lemma 3.1. *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*

$$x=w \Leftrightarrow \prod_{i=0}^{m-1}(2x_iw_i+1-x_i-w_i)=1 \tag{3}$$

Proof. Let x_i and w_i be the $(i+1)$-th character of x and w, respectively, with $0\leq i<m$. Let moreover $\phi(x_i,w_i)=(2x_iw_i+1-x_i-w_i)$ the function given in (1). Then the following expressions are equivalent

i. $x=w$
ii. $x_i=w_i,$ for each $0\leq i<m$
iii. $\phi(x_i,w_i)=1,$ for each $0\leq i<m$
iv. $\prod_{i=0}^{m-1}\phi(x_i,w_i)=1$

where (ii) is by definition, (iii) is according to (ii) and equation (1) while (iv) is according to (iii). The last relation proves our thesis. □

Theorem 3.1. *Given a pattern X, of length M, and a text Y, of length N, both over the 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 3.1 correctly computes the occurrences of X in Y. Formally $\alpha(X,Y)$ is equal to $\left|\left\{j\ :\ 0\leq j\leq N-M \text{ and } X=Y[j..j+M-1]\right\}\right|$.*

Proof. We observe that the number of occurrences of X in Y is given by the number of substrings of length m in y which are equal to x, starting at positions $j\log(\sigma)$, with $0\leq j<N-M$. We use the polynomial function introduced in Lemma 3.1 in order to accumulate the number of such substrings in the text. Formally we have the following equality relations

$$\begin{aligned}
\alpha(X,Y)= \quad & \text{(i)} \quad \sum_{j=0}^{N-M}\left(\prod_{i=0}^{m-1}\left(2x_iy_{(j,i)}+1-x_i-y_{(j,i)}\right)\right)= \\
& \text{(ii)} \quad \left|\left\{j\ :\ \begin{array}{l}0\leq j<N-M \text{ and}\\ \prod_{i=0}^{m-1}\left(2x_iy_{(j,i)}+1-x_i-y_{(j,i)}\right)=1\end{array}\right\}\right|= \\
& \text{(iii)} \quad \left|\left\{j\ :\ \begin{array}{l}0\leq j\leq N-M \text{ and}\\ x=y[j\log(\sigma)..j\log(\sigma)+m-1]\end{array}\right\}\right|= \\
& \text{(iv)} \quad \left|\left\{j\ :\ 0\leq j\leq N-M \text{ and } X=Y[j..j+M-1]\right\}\right|
\end{aligned}$$

where (i) is by Definition 3.1, (ii) is by equation (1), (iii) is by Lemma 3.1 and (iv) is by Definition. The last relation proves our thesis. □

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 the 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 (using procedure in Definition 3.1) such position, say p_1, and subsequently to verify if such information is correct. Specifically

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

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 $0 < j \leq k$, $\pi(X, Y, j) = p_j$ 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 [23].

More formally we want to find all substrings $Y[j..j+M-1]$, for $0 \leq j < N$, such that $\left|\{i : 0 \leq i < M \text{ and } X_i \neq Y_{j+i}\}\right| \leq \delta$.

We first 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 3.2 *(Mismatch function). Let X and W be 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 $\Psi : \Sigma^M \times \Sigma^M \rightarrow \{0, 1, .., M\}$, where $\Psi(X, W)$ is defined as*

$$\sum_{j=0}^{M-1}\left[1 - \prod_{i=0}^{\log\sigma-1}\left(2x_{(j,i)}w_{(j,i)} + 1 - x_{(j,i)} - w_{(j,i)}\right)\right] \tag{4}$$

counts the number of mismatches between X and W.

Proof. First of all observe that $X_j = x[j\log(\sigma)..(j+1)\log(\sigma) - 1]$. Thus the product of the factors $\left(2x_{(j,i)}w_{(j,i)} + 1 - x_{(j,i)} - w_{(j,i)}\right)$, for $i \in \{0..\log\sigma - 1\}$, can be used to detect a mismatch between the characters of X_j and W_j. Specifically it is trivial to observe that

$$X_j = W_j \Leftrightarrow \prod_{i=0}^{\log\sigma-1}\left(2x_{(j,i)}w_{(j,i)} + 1 - x_{(j,i)} - w_{(j,i)}\right) = 1 \qquad \square \tag{5}$$

Observe that the polynomial function given in (5) has degree $2\log(\sigma)$.

Let us take into account the value $\tau(X, W)$, defined as the product of the differences between $\Psi(X, W)$ and the values in the range $\{0..M\}$. Formally

$$\tau(X, W) = \prod_{i=0}^{M}(\Psi(X, W) - i)\,. \tag{6}$$

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 (6) is equal to zero.

We next define the following k-error constant τ_k, for a string 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 are now ready to prove the following lemma which introduces a polynomial function for detecting if X and W differs in exactly k characters.

Lemma 3.3 *(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*

$$\tau_k(X, W) = \frac{1}{\tau_k} \prod_{i=0}^{k-1} (\Psi(X, W) - i) \times \prod_{i=k+1}^{M} (\Psi(X, W) - i) \tag{8}$$

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

Proof. Suppose first that $\Psi(X, W) \neq k$. In such case one of the factors in (8) will be equal to 0, thus we have $\tau_k(X, W) = 0$.

Suppose now $\Psi(X, W) = k$. In this case none of the factors in (8) will be null, obtaining the following relations, which proves the lemma

$$\begin{aligned}\tau_k(X, W) &= \frac{1}{\tau_k} \prod_{i=0}^{k-1} (\Psi(X, W) - i) \times \prod_{i=k+1}^{M} (\Psi(X, W) - i) \\ &= \frac{1}{\tau_k} \prod_{i=0}^{k-1} (k - i) \times \prod_{i=k+1}^{M} (k - i) = \frac{1}{\tau_k} \prod_{i=1}^{k} i \times \prod_{i=k-M}^{-1} i = \frac{1}{\tau_k} \tau_k = 1 \qquad \square\end{aligned}$$

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 shows how to compute the number of approximate occurrences of a pattern X in a text Y with k errors. It trivially follows from Lemma 3.3.

Corollary 3.1 *(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 $\beta_k(X, Y) = \sum_{i=0}^{N-M} \tau_k(X, Y[i..i + M - 1])$.*

The following corollary shows how to compute the approximate occurrences of X in Y assuming an error bound δ. It follows from Corollary 3.1.

Corollary 3.2 *(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*

$$\gamma(X, Y) = \sum_{k=0}^{\delta} \beta_k(X, Y) \tag{9}$$

As previously described we can also adapt such technique to find the position of the first occurrence, as well 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. We need, in fact, for both exact and approximate pattern matching, 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 (3) and (4) can be rewritten, respectively, as

$$\prod_{i=0}^{m-1} (x_i w_i + (1 - x_i)(1 - w_i)) \tag{10}$$

$$\sum_{i=0}^{M-1} \left[1 - \prod_{i=0}^{\log \sigma - 1} \left(x_{(j,i)}(1 - w_{(j,i)}) + (1 - x_{(j,i)}) w_{(j,i)} \right) \right] \tag{11}$$

Thus for instance, if all bits in x are equal to 0, i.e. $x = 0^m$, then the polynomial in (10) is equal to $\prod_{i=0}^{m-1}(1 - w_i)$, while it is equal to $\prod_{i=0}^{m-1} w_i$ when $x = 1^m$. According to such observation the number of exact and approximate occurrences of the string X in Y can be computed using the algorithms shown in Fig. 2.

```
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. 2. Procedures EXACT-MATCHING and APPROXIMATE-MATCHING for computing the dynamic polynomials given in (2) and in (9), respectively.

Specifically, the algorithm EXACT-STRING-MATCHING shown in Fig. 2 computes the dynamic polynomial correspondent to the function in (2) 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 Fig. 2 computes the dynamic polynomial correspondent to the function in (9). 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

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 $2m$).[6]

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.[7] We stress that, differently than FFT we keep this alternative representation along the whole evaluation: interpolation is applied only once, to compute the final tag that the server sends back to the client.

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 even further via preprocessing. This is because the homomorphic MAC from [9] 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.

[6] Recall that in the homomorphic MAC scheme from [9] the size of the tags grows with the degree of the arithmetic circuit.

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

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

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

4.1. Testing environment and experiment parameters

Our code was written in C using (mainly) the GMP [18] library but also exploiting NTL [25] and gcrypt [22] 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. For each experiment, the timing is obtained as the average 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 were 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.[8] In this latter case it is possible to get an additional 20%–30% gain in performances.

4.2. 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.

4.3. Additional tests

Next, we considered the behavior of our methods when considering different text sizes and pattern lengths.[9] 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 behavior 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). 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 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. The client storage space is negligible as it is required to store only the secret key associated to the MAC computation and the pairs (*filename, size*) for each uploaded text; the latter will be used to recompute the labels associated to authenticated text bits.

4.4. Approximate pattern matching experiments

In Table 2 we report the timings of server's evaluation in the case of approximate pattern matching for a representative test setting (10 KiB text and 8 chars pattern). The evaluation is clearly slower than in the exact counterpart, as the polynomial algorithm is more involved here. Also, complexity increases for larger δ. We remark that our evaluation over samples technique still beats FFT, as long as one considers medium length patterns.

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

[9] 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.

Table 2
(On the left) Timings and sizes of exact pattern matching with both optimizations applied; (On the right) Approximate pattern matching of an 8 chars pattern on a 10K chars text.

Text (chars)	Patt. (chars)	Key gen. (ms)	Text auth. (ms)	Eval. (ms)	Verif. (ms)	Text tags (bytes)	Proof tag (bytes)
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

Bound δ (chars)	Evaluation (s)		
	Dyn. polyn.	+FFT	+over sam.
2	35.053	24.262	24.005
3	53.548	26.103	23.518
4	71.675	32.311	30.452
5	93.562	44.728	39.435
6	108.413	57.977	35.882

Server Side Procedure:

(a) locally compute $W = X[u..u+3]$ which minimizes the value $k = \alpha(W, Y)$, with $0 \leq u < M-3$;
(b) locally compute the sequence of all k positions where W occurs in Y, i.e. the values $p_j = \pi(W, Y, j)$, for $j = 1..k$;
(c) compute the proof-tag of $\alpha(W, Y)$ and send it to the client;
(d) compute the $(2 \times k)$ proof-tags of $\alpha(W, Y[p_j..p_j+3])$ and $\alpha(X, Y[p_j - u..p_j - u + M - 1])$, for $j = 1..k$, and send them to the client;

Client Side Procedure:

(a) receive position u together with the proof-tags of $\alpha(W, Y)$ and $\alpha(W, Y[p_j..p_j+3])$, $\alpha(X, Y[p_j - u..p_j - u + M - 1])$, for $j = 1..k$;
(b) verify the proof-tag $\alpha(W, Y)$;
(c) verify the $(2 \times k)$ proof-tags $\alpha(W, Y[p_j..p_j+3])$ and $\alpha(X, Y[p_j - u..p_j - u + M - 1])$, for $j = 1..k$;

Fig. 3. The workflow of the new practical approach, for server and client side.

5. A practical approach in the case of long patterns

In this section we present an approach to drastically speed-up in practice the searching process in the case of long patterns. The need for a more effective approach arises from experimental results reported in Table 2 where it turns out that, when the size of the pattern exceeds 4 characters, the timings for searching a pattern in an outsourced text could be considered excessive.

We start observing that the average number of occurrences reported in a text drastically reduces when the size of the pattern increases. Upon experiments, it turns out that the number of occurrences of a pattern of length 4 reported in a text of 100K characters is 251.62 on average.[10] This value decreases to 13.02 for a pattern of length 8, and to 1.92 for a pattern of length 16. Nevertheless searching for short patterns turns out to be faster.

Based on these observations, the idea behind our approach consists in searching in the text, at first, the positions of all the occurrences of a given (short) substring of the pattern, and then in locally testing such positions in order to retrieve the occurrences of the whole pattern. If the substring involved in the first step is short enough and the number of its occurrences is not excessive, the whole searching can be performed in a reasonable time.

As above, let X be a pattern of length M and Y be a text of length N, both over the alphabet Σ of size σ. The workflow of our proposed approach is described in Fig. 3 for both server and client side. In details, at first the server computes locally W, the substring of X of length 4 which minimizes the number of occurrences in the text, i.e. the value $k = \alpha(W, Y)$. This can be done by naively searching every substring of the pattern of length 4 in the text and comparing the corresponding number of occurrences. Since this step can be run locally at server side it is possible to use any known string matching algorithm. In our experimental settings we use the WFR algorithm [6], one of the faster string matching solutions in practical cases. In the next let W begin at position u of X.

In a second step the server locally computes the sequence of all k positions where W occurs in Y, i.e. the values $p_j = \pi(W, Y, j)$, for $j = 1..k$. Thus we have that $W = Y[p_j..p_j+3]$ for all $j = 1..k$. For each $j = 1..k$ the server locally verifies if a whole occurrence of the pattern can be found at position $p_j - u$ (a positive case: $X = Y[p_j - u..p_j - u + M - 1]$) or if just the sub-string is present (a false-positive case: $X \neq Y[p_j - u..p_j - u + M - 1]$), for $j = 1..k$. In order to allow the client to verify such information the server computes the proof-tag of $\alpha(W, Y)$ and the following $(2 \times k)$ proof-tags: one (positive) proof for the sub-string $(\alpha(W, Y[p_j..p_j+3]))$ and one to eventually prove the presence of the full pattern $(\alpha(X, Y[p_j - u..p_j - u + M - 1]))$, for $j = 1..k$.[11] They are sent to the client, together with the offset u and the number k.

[10] The average number of occurrences reported in a text has been computed by searching 1000 patterns, randomly extracted from the text, and by taking the mean of the reported occurrences. The used document is the English King James version of the Bible.
[11] Please note that the usage of the first tag is not strictly necessary on the positions where the full pattern matches.

Table 3
Timings and sizes of exact pattern matching on 10K and 100K English texts for different lengths of the pattern, ranging from 4 to 16.

Pattern (chars)	10K text			100K text		
	Eval. (ms)	Verif. (ms)	Proof tag (bytes)	Eval. (ms)	Verif. (ms)	Proof tag (bytes)
6	4143.0	308.7	30320	38022.1	3285.8	198096
8	4162.7	309.5	40048	38150.5	3289.3	262608
10	4198.3	310.2	49776	38389.0	3293.0	327120
12	4246.5	310.8	59504	38708.7	3296.9	391632
14	4309.6	311.2	69232	39126.0	3300.6	456144
16	4406.8	311.7	78960	39775.2	3304.2	520656

Observe that the time required for steps (a) and (b) of the server side procedure is negligible, if compared with the time required for steps (c) and (d). This is because (a) and (b) are performed locally and do not need to create a verification tag.

Table 3 reports experimental results obtained by running exact pattern matching on a 10K and on a 100K English text, respectively, for different lengths of the pattern, ranging from 6 to 16. Experimental results have been conducted using the English King James version of the Bible (with and alphabet of $\sigma = 63$ characters). For each experimental evaluation we have generated sets of 1000 patterns of fixed length M randomly extracted from the text, for M ranging from 5 to 16. Running times have been computed as the mean over the 1000 runs. Observe that the new approach allows an evaluation time which is up to 95% faster than the standard approach (speed-up of 74% in the case of $M = 8$ and 95% in the case of $M = 16$). Similarly, regarding the verification time, the new approach leads to a gain in performances which is up to 74% (speed-up of 48% in the case of $M = 8$ and 71% in the case of $M = 16$). It is fair to observe that such method creates bigger proofs. Although such performances cannot be guaranteed in all cases, this approach turns out to be much more suitable for long patterns in most cases.

Acknowledgements

This research was supported in part by a PTR 2014 grant by the University of Catania. Thanks to Nuno Tiago Ferreira de Carvalho for his Homomorphic MACs library.[12]

References

[1] S. Agrawal, D. Boneh, Homomorphic MACs: MAC-based integrity for network coding, in: M. Abdalla, D. Pointcheval, P.-A. Fouque, D. Vergnaud (Eds.), ACNS 09, in: Lect. Notes Comput. Sci., vol. 5536, Springer, Jun. 2009, pp. 292–305.
[2] M. Backes, D. Fiore, R.M. Reischuk, Verifiable delegation of computation on outsourced data, in: A.-R. Sadeghi, V.D. Gligor, M. Yung (Eds.), ACM CCS 13, ACM Press, Nov. 2013, pp. 863–874.
[3] R.A. Baeza-Yates, G.H. Gonnet, A new approach to text searching, Commun. ACM 35 (10) (1992) 74–82.
[4] D. Boneh, D.M. Freeman, Homomorphic signatures for polynomial functions, in: K.G. Paterson (Ed.), EUROCRYPT 2011, in: Lect. Notes Comput. Sci., vol. 6632, Springer, May 2011, pp. 149–168.
[5] D. Boneh, D. Freeman, J. Katz, B. Waters, Signing a linear subspace: signature schemes for network coding, in: S. Jarecki, G. Tsudik (Eds.), PKC 2009, in: Lect. Notes Comput. Sci., vol. 5443, Springer, Mar. 2009, pp. 68–87.
[6] D. Cantone, S. Faro, A. Pavone, Speeding up string matching by weak factor recognition, in: J. Holub, J. Žďárek (Eds.), Proceedings of the Prague Stringology Conference 2017, Czech Technical University in Prague, Czech Republic, 2017, pp. 42–50.
[7] D. Catalano, Homomorphic signatures and message authentication codes, in: SCN 14, in: Lect. Notes Comput. Sci., Springer, 2014, pp. 514–519.
[8] D. Catalano, M. Di Raimondo, S. Faro, Verifiable pattern matching on outsourced texts, in: V. Zikas, R. De Prisco (Eds.), Security and Cryptography for Networks: Proceedings of the 10th International Conference, SCN 2016, Amalfi, Italy, August 31–September 2, 2016, Springer International Publishing, Cham, 2016, pp. 333–350.
[9] D. Catalano, D. Fiore, Practical homomorphic MACs for arithmetic circuits, in: T. Johansson, P.Q. Nguyen (Eds.), EUROCRYPT 2013, in: Lect. Notes Comput. Sci., vol. 7881, Springer, May 2013, pp. 336–352.
[10] D. Catalano, D. Fiore, R. Gennaro, L. Nizzardo, Generalizing homomorphic MACs for arithmetic circuits, in: H. Krawczyk (Ed.), PKC 2014, in: Lect. Notes Comput. Sci., vol. 8383, Springer, Mar. 2014, pp. 538–555.
[11] P. Clifford, R. Clifford, Simple deterministic wildcard matching, Inf. Process. Lett. 101 (2) (2007) 53–54.
[12] M. Crochemore, W. Rytter, Text Algorithms, Oxford University Press, 1994.
[13] S. Faro, T. Lecroq, The exact online string matching problem: a review of the most recent results, ACM Comput. Surv. 45 (2) (2013) 13.
[14] S. Faust, C. Hazay, D. Venturi, Outsourced pattern matching, in: Proceedings of the 40th International Conference on Automata, Languages, and Programming, Part II, ICALP '13, Springer-Verlag, Berlin, Heidelberg, 2013, pp. 545–556.
[15] R. Gennaro, C. Gentry, B. Parno, Non-interactive verifiable computing: outsourcing computation to untrusted workers, in: T. Rabin (Ed.), CRYPTO 2010, in: Lect. Notes Comput. Sci., vol. 6223, Springer, Aug. 2010, pp. 465–482.
[16] R. Gennaro, D. Wichs, Fully homomorphic message authenticators, in: K. Sako, P. Sarkar (Eds.), ASIACRYPT 2013, Part II, in: Lect. Notes Comput. Sci., vol. 8270, Springer, Dec. 2013, pp. 301–320.
[17] S. Gorbunov, V. Vaikuntanathan, D. Wichs, Leveled fully homomorphic signatures from standard lattices, in: 47th ACM STOC, ACM Press, 2015, pp. 469–477.
[18] T. Granlund, The GMP Development Team, GNU MP: the GNU multiple precision arithmetic library, 6th edition, https://gmplib.org, 2016.

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

[19] R. Johnson, D. Molnar, D.X. Song, D. Wagner, Homomorphic signature schemes, in: B. Preneel (Ed.), CT-RSA 2002, in: Lect. Notes Comput. Sci., vol. 2271, Springer, Feb. 2002, pp. 244–262.
[20] J. Kärkkäinen, J.C. 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.
[21] D.E. Knuth, J.H. Morris Jr., V.R. Pratt, Fast pattern matching in strings, SIAM J. Comput. 6 (2) (1977) 323–350.
[22] W. Koch, The Libgcrypt Development Team, Libgcrypt, 1st edition, https://www.gnu.org/software/libgcrypt/, 2016.
[23] G.M. Landau, U. Vishkin, Efficient string matching with k mismatches, Theor. Comput. Sci. 43 (1986) 239–249.
[24] D. Papadopoulos, C. Papamanthou, R. Tamassia, N. Triandopoulos, Practical authenticated pattern matching with optimal proof size, Proc. VLDB Endow. 8 (7) (2015) 750–761.
[25] V. Shoup, NTL: a library for doing number theory, 9th edition, http://www.shoup.net/ntl/, 2016.
[26] M. Yasuda, T. Shimoyama, J. Kogure, K. Yokoyama, T. Koshiba, Secure pattern matching using somewhat homomorphic encryption, in: Proceedings of the 2013 ACM Workshop on Cloud Computing Security Workshop, CCSW '13, ACM, New York, NY, USA, 2013, pp. 65–76.