Privacy-preserving String-Matching With PRAM Algorithms

Report in MTAT.07.022 Research Seminar in Cryptography, Fall 2014

Author: Sander Siim
Supervisor: Peeter Laud

December 14, 2014

Abstract

In this report, we investigate the applicability of algorithms designed for parallel computers (PRAM) to build efficient computations in a secure multi-party (SMC) computation setting based on secret sharing. As an example, we have chosen to implement string-matching algorithms to assess the efficiency and possible gains of parallel algorithm design paradigms in SMC. The use of parallel algorithm methods is made possible and motivated by recent advances of efficient protocols used to perform oblivious data accesses in a parallel manner. We implemented the algorithms on the Sharemind platform and show that in our example case, more efficient computations can be performed using this approach as opposed to previous canonical methods.

1 Introduction

The general theme of this report is *secure multi-party computation* (SMC). SMC is an area of study in cryptography, which aims to develop methods for multiple mutually distrusting parties to securely perform computations on their joint secret data, without leaking one's inputs to the other parties. Formally, in the most general setting, parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$ with respective inputs $x_1, \ldots, x_n$ would like to securely evaluate a function f, such that $f(x_1, \ldots, x_n) = (y_1, \ldots, y_n)$ and in the process, each party $\mathcal{P}_i$ should learn only what can be derived from its output y_i and nothing else.

Over the last ~30 years, since SMC has become a topic of great interest among researchers and cryptographers, many well-known and provably secure methods to develop SMC protocols have been established. One of the most efficient among these methods in the semihonest model is *secret sharing* [2][11], which allows for fast and highly parallelized SMC protocols. For implementing the protocols discussed in this report, we use the Sharemind [3] privacy-preserving computational platform, which is

mostly based on secret sharing. For efficient integer arithmetic, the *additive* secret sharing scheme is used in Sharemind. However, for non-linear operations, such as bitwise operations, the *bitwise* sharing scheme is used instead as it is more efficient for these types of operations. In the context of this work, we are interested in both of these schemes, the concepts of which are explained in the next section (Sec. 2).

Throughout this work, we consider the *semi-honest* security model, as the most efficient Sharemind protocols provides security only against passive adversaries, who are assumed to follow the protocol, but try to deduce some private information from the data that they receive during the protocol. As such, the concepts discussed here might also expand to the malicious case, if the underlying primitives are advanced to provide security against active adversaries.

The main goal of this project report is to experiment with implementing algorithms that are designed for parallel computers in a SMC setting. Since SMC protocols based on secret sharing require the computing parties to exchange network messages, the efficiency concerns are mostly related to reducing the number of network messages sent back and forth. This is referred to as network *rounds*. A high number of rounds is undesirable, as the network latency introduces a significant overhead to the running-time of the protocols. Thus, better efficiency in large computations can be gained if many messages are packed together in one round of communication. For example, most of the protocols in Sharemind can be *parallelized* in such a manner to provide near constant-time complexity, given that the network bandwidth is sufficiently large.

Given this intuition of using parallelization to decrease the time-complexity of SMC protocols, it is natural to look into the world of parallel algorithms to leverage the methods developed there for efficiently dividing a computation into subproblems that can be solved independently in parallel. In this work, we focus on the PRAM computation model, since it is rather easy to work with and provides a well-developed body of techniques and methods to handle different classes of computational problems.

In the past, this approach has been hindered by the fact, that oblivious data access in SMC is very costly, but it is usually difficult to translate regular PRAM algorithms to SMC without having such a primitive available. The recent work by P. Laud [10] introduces very efficient protocols for obliviously reading and writing data according to private addresses. Moreover, the asymptotic complexity is best when many data accesses are performed in parallel, making it the ideal tool for approaching PRAM algorithms.

In this report, we describe how an efficient string-matching algorithm is implemented in PRAM and describe how this can be implemented in a SMC setting based on secret sharing. Most of our discussion concerning PRAM and parallel string algorithms is based on the textbook "An Introduction to Parallel Algorithms" by Joseph JaJa [8].

In Section 2, we describe preliminary notions about secret sharing, strings and PRAM. We then present the privacy-preserving string-matching protocol which is based on the corresponding PRAM algorithm in Section 3. Finally, Section 4 describes our implementation on Sharemind and the results of performance benchmarks.

2 Preliminaries

2.1 Secret sharing

Secret sharing is a mechanism of distributing data between participants without giving any of them direct access to the data, but enabling computations [2][11]. Suppose that in our setting there are m parties $\mathcal{P}_1, \ldots, \mathcal{P}_m$ who want to perform secure computations. Then, a secret value x is divided into m *shares*, and each party receives a share of the secret. The basis of any secret sharing scheme is that each of the individual shares is uniformly random and leaks nothing about the actual value to the receiver. However, the secret can be later reconstructed by combining a subset of the shares. For a *k-out-of-m secret sharing scheme*, the secrets are divided into m shares and knowing any $k-1$ shares does not reveal the original secret. Secret sharing schemes can be used to build SMC protocols that perform distributed computations on secret-shared data, without revealing any information about the inputs or outputs to the computing parties [1, 6].

Concerning notation, we use $[\![x]\!]$ to denote that a value x is secret-shared. For denoting a secret-shared vector $\vec{x} = (x_1, x_2, \ldots, x_n)$ of n elements, we use $[\![\vec{x}]\!] = [\![x_1, x_2, \ldots, x_n]\!] = ([\![x_1]\!], \ldots, [\![x_n]\!])$. Note that each element of the vector is shared separately. The individual share of a computing party $\mathcal{P}_i$ for secret-shared $[\![x]\!]$ is denoted with $[\![x]\!]_i$.

In the *additive secret sharing scheme*, the shared values and corresponding shares are elements of a ring, say $\mathbb{Z}_{2^k}$. This corresponds to the set of k-bit integers. Sharing is then defined with $x = \sum_{i=1}^{m} [\![x]\!]_i$, where the shares $[\![x]\!]_1, \ldots, [\![x]\!]_{m-1}$ are chosen uniformly from the set $\mathbb{Z}_{2^k}$, and the last share is taken as $[\![x]\!]_m = x - \sum_{i=1}^{m-1} [\![x]\!]_i$. The `additive3pp` protection domain of Sharemind implements all basic integer arithmetic protocols on a 3-out-of-3 additive secret sharing scheme, which are detailed and proven to be secure in [5][3].

In the following, we will mostly ignore the low-level details of these protocols and use them implicitly as existing black-box primitives in a more high-level protocol description. For example, we denote addition and multiplication of additively shared integers by writing $[\![x]\!] + [\![y]\!]$ and $[\![x]\!] \cdot [\![y]\!]$. Also, since most Sharemind protocols can be parallelized and run on a set of inputs, we denote performing an element-wise binary operation $\otimes$ on two secret-shared vectors $[\![\vec{x}]\!]$ and $[\![\vec{y}]\!]$ as $[\![\vec{z}]\!] = [\![\vec{x}]\!] \otimes [\![\vec{y}]\!]$, where $z_i = x_i \otimes y_i$ and all vectors are of equal length.

Notice that the additive sharing scheme is additively homomorphic by definition, which means adding two secret-shared values $[\![x]\!] + [\![y]\!]$ requires no communication between parties, and can be performed simply by local addition of the individual shares of $[\![x]\!]$ and $[\![y]\!]$. However, multiplication and other operations can not be performed so easily and require the parties to exchange network messages to arrive at the result. It is important to understand that the round-complexity for performing a private multiplication on two elements is the same as performing element-wise multiplication of two arbitrarily long vectors, as the network messages in each round are packed together into a single larger message. Since the sizes of the sent network messages is dependent of

the size of the input vectors, the amount of available network bandwidth between the computing parties ultimately bounds the running-time of vectorized operations.

Although integer arithmetic is very efficient with additive secret sharing, comparisons (e.g $<, \neq$) and bitwise operations (e.g AND, OR, XOR) are more straightforward and efficient when integers are instead *bitwise secret-shared*. This means that a secret value x is shared as $x = [\![x]\!]_1 \oplus [\![x]\!]_2 \oplus \ldots \oplus [\![x]\!]_m$, where $\oplus$ denotes the XOR operation. This is essentially equivalent to additively sharing each bit of the integer x separately in $\mathbb{Z}_2$. Also, it is possible to convert an additively secret-shared value to bitwise-shared form and vice versa [5][7].

In our protocols, we use both additively and bitwise secret-shared values depending on the type of data. We explain in detail in which cases each secret sharing scheme is used when we present our string-matching protocols in Section 3

2.2 Strings

We now describe our formalization and notation of strings that is largely based on Chapter 7 of [8], which presents the PRAM string algorithms used in this work. Note that we actually use a somewhat different notation when presenting our SMC string-matching protocols. However, the discussion here is necessary for understanding the optimal parallel string-matching algorithm, and a different notation is more suitable for this purpose.

Let Σ be an *alphabet* consisting of a finite amount of symbols. For implementation purposes, we can always assume $\Sigma \subseteq \mathbb{N} \setminus \{0\}$, since any finite set of symbols can be encoded with natural numbers by defining the appropriate bijection with a subset of the natural numbers. Noteworthy examples of such encodings are ASCII [1] and UTF-8 (if we pad UTF-8 characters with zeros to always fill 4 bytes, as RFC 3629 [12] defines up to 4-byte encodings for characters). Note that we have excluded 0 from our alphabet, since we will need to reserve 0 as a special symbol in our algorithms that does not appear in any string.

A *string* X is a finite ordered sequence of elements from Σ, the length of which is denoted as $|X|$. Throughout this work we use capitalized variables $A, B, \ldots, Z$ to denote strings. It is natural to represent strings as arrays where $X[i]$ is the i-th element of X for $0 \leq i < |X|$[2]. A concatenation of strings X and Y is denoted as XY and consists of the sequence of elements of X followed by the elements of Y.

Let Y be a string with length m. A *substring* of Y is defined by the subarray $Y[i:j]$ for $0 \leq i < j \leq m$, which contains the sequence of elements $Y[i], Y[i+1], \ldots, Y[j-1]$.[3] Note that the length of $Y[i:j]$ is then exactly $j - i$. A substring of form $Y[0:i]$ is called a *prefix* of Y and similarly, a substring $Y[i:m]$ is called a *suffix*.

[1] `http://www.ascii-code.com`

[2] Note that the formalization of [8] uses indexing starting from 1, but we found it confusing from an implementation point of view, and decided to use 0-indexing instead.

[3] Again, we find this kind of subarray indexing which is used in most programming languages as more intuitive to write down algorithms with.

We say that a string X *occurs* in Y at position i, if X is equal to the substring of Y with length $|X|$ starting at position i, namely $X = Y[i : i + |X|]$. We also say that X matches Y at position i.

An important concept of strings, which we need to introduce, is *periodicity*. Again, let Y be a string on length m. We say X is *a period* of Y if $Y = X^k X'$, where X^k is a concatenation of X with itself for k times, and X' is some prefix of X. This means that Y consists of multiple copies of X followed by a prefix of X. Note that X is a period of Y if and only if Y is a prefix of XY. Also, each string is always a period of itself.

In addition, we define *the period* of Y as its shortest period. In the following, we are mostly interested in the smallest period of a string, and usually denote the length of the period with p. Also, depending on the context, we also refer to p as the period of Y (the length, not the substring itself). If p is the period of Y, then p is the smallest integer between $1 \leq p \leq m$, such that $Y[i] = Y[i + p]$ for all i satisfying $0 \leq i < m - p$. Finally, we call Y *periodic* if its period p satisfies $p \leq m/2$.

Example. Let $Y = ababababa$. Each of the following is a period of Y: ab, $abab$ and $ababab$. Especially, $U = abab$ is a period of Y, since $Y = U^2a$. However, *the* period of Y is ab, since it is the smallest period, and as such, $p = 2$. Also, Y is periodic, since $p \leq |Y|/2$.

Another important concept, which is fundamental in constructing an optimal parallel string-matching algorithm we discuss in Section 3, is the so-called *witness function* for a given string [8]. Let Y be a string of length m and period p. Let $\pi(Y) = \min(p, \lceil \frac{m}{2} \rceil)$, that is $\pi(Y)$ is equal to the period of Y if it is periodic, and $\lceil \frac{m}{2} \rceil$ otherwise. The witness function ϕ_{wit} is defined for $0 \leq i < \pi(Y)$ as follows:

$$\phi_{wit}(i) = \begin{cases} 0, \text{ if } i = 0 \\ k, \text{ such that } Y[k] \neq Y[k + i] \text{ otherwise} \end{cases}$$

where $0 \leq k < m - i$. That is, k is some position in Y such that the element i positions forward is different from the element at position k. First notice that ϕ_{wit} is defined correctly, that is, for an i such that $1 \leq i < \pi(Y)$, there always exists a position k so that $Y[k] \neq Y[k + i]$. This can be intuitively understood by considering the string Y shifted i times to the right and placed on top of Y, so that the elements $Y[0]$ and $Y[i]$ are aligned.

Consider the case when $i = p$, the period of Y. Then clearly the overlapping parts of Y and Y shifted by p elements are equal (e.g consider two strings U^3U' and U^2U'). Now, since $i < \pi(Y)$ is smaller than the period of Y, then there must exist some position k, such that $Y[k] \neq Y[k + i]$, otherwise $i < p$ would be the period of Y, which is a contradiction.

We represent the witness function ϕ_{wit} for a string Y as an array $\vec{\mathcal{W}}$ of size $\pi(Y)$, such that $\mathcal{W}_i = \phi_{wit}(i)$. In Section 3, we show how to calculate the witness array for

a given string and also, how the witness array can be used to easily disqualify many positions, where two strings do not match. This idea is used in a fast algorithm for string-matching.

2.3 The PRAM model

The PRAM (*parallel random-access machine*) is a theoretical model of computation for algorithms that can use multiple processors for simultaneous computations[4]. We do not go into the exact details of parallel computational models, but instead try to give an overview of the most important aspects to provide an intuition, why these models and especially PRAM are interesting and can help in designing more efficient protocols in the SMC setting.

The PRAM model is in general rather simplistic and ignores many aspects of modern computer architecture such as the overhead of synchronization and communication between processors (or processor cores) among others. However, it captures the main principles of parallel computing and allows efficient algorithms to be designed and analyzed in a straightforward manner. The methods and paradigms developed for PRAM have also proven to be robust and usable in other parallel computing models. Since efficient PRAM algorithms have been developed for a variety of computational problems, we think it is a natural choice for investigation to find ways of optimizing parallelizable SMC protocols.

The PRAM model is a *synchronous shared-memory* model. This means that all processors work synchronously under the control of a common clock and share a single global memory unit where data is read from and written into. Most algorithms for PRAM are of the type *single instruction multiple data* (SIMD). That is, all processors execute the same code in parallel, but generally on different input data.

Sharing a common memory unit between processors of course raises the issue of concurrent data accesses. There are multiple variants of PRAM based on whether simultaneous reads and writes to the same memory are allowed or not. We are able to implement algorithms in the most powerful CRCW PRAM (*concurrent read concurrent write*) model, since the protocols from [10] allow us to do both private parallel reads and writes. In particular, we are allowed to read a value from a particular index many times and also write to the same array element based on a priority vector so that only the highest priority write succeeds. This is referred to as *priority* CRCW PRAM.

A simple way to analyze asymptotic complexity of PRAM algorithms is the *work-time* (WT) paradigm. We first assume that we are able to do an arbitrary number of operations in parallel, that is, arbitrarily many processors are available. We can then analyze the total number of operations, or amount of *work*, required for running the algorithm on an input of size n. This amount is denoted as $W(n)$. We can also estimate the *time*-complexity $T(n)$.

Consider calculating the element-wise product of two vectors of size n. Since each

[4]The introduction to PRAM in this section is largely based on [8]

multiplication can be performed independently on a different processor, the time-complexity is $T(n) = O(1)$. However, the total number of operations required is $W(n) = O(n)$. Given these complexities assuming infinitely many processors, we can later evaluate the complexity when exactly p processors are available.

A parallel algorithm is considered *optimal* (in the weak sense), if the required work for the algorithm satisfies $W(n) = \Theta(T^*(n))$, where $T^*(n)$ is the time-complexity of a sequential algorithm solving the same problem, such that it can be proven that this time bound is the best that can be achieved. Notice that this notion of optimality does not take into account the running-time of the parallel algorithm.

If the time-complexity of an optimal parallel algorithm is $T(n)$ and it can be shown that this bound cannot be improved by any other *optimal* parallel algorithm, then the algorithm is considered *optimal in the strong sense*.

3 Privacy-Preserving String-Matching

We are now ready to describe our implemented protocols for solving the *string-matching* problem for secret-shared strings. Let T be a string of length n and P a string of length m, such that $m \leq n$. We will refer to T as the *text* and P as the *pattern*. The string-matching problem is defined then as finding all positions in the text that match with the pattern. Formally, we need to calculate the *match* array $\mathcal{M}$ of length $n - m + 1$, such that $\mathcal{M}_i = 1$ if and only if P matches T at position i and otherwise $\mathcal{M}_i = 0$.

We will start by explaining how we represent strings in secret-shared form. We then continue with presenting a naive protocol based on a brute-force algorithm, which would actually be a reasonable way to implement string-matching on Sharemind for instance, since it has very low round-complexity. We later find that by using a PRAM paradigm called *accelerated cascading* [8], we can obtain a faster protocol, which although it has much higher round-complexity, performs better on larger inputs. We verify this empirically in Section 4 to find that our implementation is already faster on reasonable input sizes.

For implementing the faster protocol, we are required to calculate the witness array for the pattern, for which we present a simple protocol. The witness array computation could also be optimized with similar parallel methods as the string-matching algorithm, but for brevity and lack of time, we use a simpler naive approach.

For string-matching, we actually present multiple variants of the protocol. First, we construct a protocol based on an *optimal* PRAM algorithm. We then present a protocol based on a much faster algorithm, which has slightly higher than optimal work-complexity. Finally, we combine these two to get a fast and optimal algorithm. For all these protocols we restrict ourselves only to the case where the pattern is non-periodic, since the periodic case is very similar, but in the SMC setting, some additional operations are required so that the period of the pattern remains private.

3.1 Secret-shared string representation

In our implementation on Sharemind, we represent strings as vectors of bitwise-shared integers. As such, we denote a secret-shared string X as $[\![\vec{X}]\!]$. We use bitwise-sharing since we are required to perform only comparison operations with strings, which are more efficiently performed when the inputs are bitwise-shared rather than additively shared.

However, all values that represent some positions in strings, including the witness array, are additively shared, since we mostly perform addition and multiplication on these values.

In all our protocols, the goal is to maintain the privacy of all input strings. However, as vector lengths are public, we are not hiding the lengths of the strings. The lengths of the strings could also be hidden, if we use some bounded-length string representation, where the length of the vector representing a string has some fixed length. For example, if a string has a length smaller than 2^k, and this is a tight bound, we could always represent the string with a vector of length 2^k. Currently, we assume that the secret-shared vector representing a string is of the same length as the actual string.

3.2 Baseline protocol

We first present a simple brute-force protocol to solve the string-matching problem as Algorithm 1. The brute-force protocol can be easily implemented without using oblivious data accesses and has good performance on smaller inputs. One could say this is the canonical way of implementing such a protocol, as it has very low round-complexity. For example, trying to implement a faster protocol based on well-known fast sequential string-matching algorithms (e.g Knuth-Morris-Pratt algorithm [9]) would be a futile effort, since the round-complexity of the resulting protocol would most likely grow with the same rate as the time-complexity of the sequential algorithm, resulting in an inefficient protocol. However, as we will see later, using oblivious data accesses and clever use of the witness array, we can improve on this baseline protocol.

Algorithm 1: Brute-force string-matching protocol

Input: Secret-shared text $[\![\vec{T}]\!]$ of length n
Input: Secret-shared pattern $[\![\vec{P}]\!]$ of length m, such that $m \leq n$
Output: The match array $[\![\vec{\mathcal{M}}]\!]$ of length $n - m + 1$ showing all positions where $[\![\vec{P}]\!]$ matches $[\![\vec{T}]\!]$

1 **for** $i := 0$ **to** $n - m$ **do in parallel**
2 | **foreach** $j \in \{0, \ldots, m-1\}$ **do** $[\![C_j]\!] \leftarrow [\![T_{i+j}]\!]$
3 | $[\![\vec{d}]\!] \leftarrow [\![\vec{C}]\!] \stackrel{?}{\neq} [\![\vec{P}]\!]$
4 | $[\![\mathcal{M}_i]\!] \leftarrow \mathsf{sum}([\![\vec{d}]\!]) \stackrel{?}{=} [\![0]\!]$
5 **return** $[\![\vec{\mathcal{M}}]\!]$

The idea of the brute-force protocol is simple, we compare all the substrings $T[i : i + m]$ for $i \in \{0, 1, \ldots, n - m\}$ to the pattern P to see in which positions there is a match. Here, we use the equality protocol of bitwise-shared integers from [5] to compare the substrings with the pattern. Note that computing the inverse of a secret-shared bit is reduced to locally flipping the bits of the shares, thus inequality is computed with the same rounds as equality.

The sum protocol denotes summing all elements in a vector, which can be performed using only local operations for an additively shared input. However, since the vector $[\![\vec{d}]\!]$ holds bit values, we first implicitly cast $[\![\vec{d}]\!]$ up to a vector of additively shared integers from $\mathbb{Z}_{2^{64}}$, before summing the results of the comparison. This conversion is implemented on Sharemind with a 3-round protocol and we omit the details here. The pattern matches if and only if all characters are equal, thus the sum has to be equal to 0.

Notice that all the operations in the for-loop can be performed independently for each cycle. This means we can perform the loop cycles in parallel. In our actual implementation, this for-loop does not exist, but rather, we construct two large input vectors and then perform a single inequality protocol on these vectors using only constant rounds. This is done by concatenating all the $n - m + 1$ substrings of T and also concatenating P with itself for $n - m + 1$ times. We can also sum the results of the comparisons and check whether they are zero in a similar vectorized manner. The construction of the large input vectors is in itself a sequential process, and although it does present some slight overhead, the main concern in practice is always the communication overhead. As such, this trade-off is very reasonable.

This strategy captures the essence of how computations on Sharemind are parallelized to reduce the total number of rounds. In our protocol descriptions, we always present parallel operations in a similar manner, with **do in parallel** loops. This hides the unnecessary details of constructing and indexing the large vectors needed for running the protocols in parallel. This representation of protocols is much alike to those of PRAM algorithms. For a PRAM algorithm, the cycles of the **do in parallel** loop would be performed independently on different processors. This shows how PRAM algorithms can be quite easily translated into protocols on Sharemind.

Also, with this analogy, it is easy to see that the time-complexity of the corresponding PRAM algorithm roughly determines the round-complexity of the corresponding Sharemind protocol. The work-complexity is similarly related to the total amount of communication used by the subprotocols. As we can see, the round-complexity for the above brute-force protocol is very low, however, the amount of communication increases quite fast with larger inputs, as the amount of comparisons we do in total is $(n - m + 1) \cdot (m + 1)$. Consequently, the brute-force protocol will not perform very well for larger inputs due to the fixed amount of bandwidth available.

Again, we can bring an analogy with the PRAM model here, as in reality we also cannot assume that we have infinitely many processors available, much like we cannot assume infinite bandwidth. Thus, our goal is to try and reduce amount of network communication used in the protocol, without increasing the round-complexity much.

We will therefore first try to construct a protocol based on an optimal parallel algorithm, with low work-complexity.

3.3 Towards a faster protocol

Before we can present the protocol based on an optimal algorithm, we must first construct a protocol for calculating the secret-shared witness array for a string.

3.3.1 Computing the witness array

Notice that since the witness array reveals some information about the structure of the pattern, we cannot make the witness array public. Also, since we do not want to reveal the period of the pattern, we always construct the witness array with length $\lceil \frac{m}{2} \rceil$, where m is the length of the pattern. For non-periodic patterns, this is a correct way of computing the witness array. In the periodic case, the extra elements of the computed array should be ignored. A simple protocol for computing the witness array is presented as Algorithm 2.

Algorithm 2: Computing the witness array for a string (brute-force)

Input: Secret-shared string $[\![\vec{Y}]\!]$ of length m
Output: Secret-shared witness array $[\![\vec{\mathcal{W}}]\!]$ of length $\lceil \frac{m}{2} \rceil$ corresponding to $[\![\vec{Y}]\!]$

1 $[\![\mathcal{W}_0]\!] \leftarrow [\![0]\!]$
2 **for** $i := 1$ **to** $\lceil \frac{m}{2} \rceil - 1$ **do in parallel**
3 **for** $j := 0$ **to** $m - 1 - i$ **do**
4 $[\![Y^*_j]\!] \leftarrow [\![Y_j]\!]$ // copy of first $m-i$ elements of $\vec{Y}$
5 $[\![C_j]\!] \leftarrow [\![Y_{i+j}]\!]$ // $\vec{Y}$ shifted left by i elements
6 $[\![\mathcal{I}_j]\!] \leftarrow [\![j+1]\!]$
7 $[\![\vec{d}]\!] \leftarrow [\![\vec{C}]\!] \stackrel{?}{\neq} [\![\vec{Y^*}]\!]$
8 $[\![\vec{d}]\!] \leftarrow \mathsf{msnzb}([\![\vec{d}]\!])$ // get first position where $\vec{C}$ and $\vec{Y}^*$ differ
9 $[\![\mathcal{W}_i]\!] \leftarrow \mathsf{sum}([\![\vec{d}]\!] \cdot [\![\vec{\mathcal{I}}]\!])$ // save corresponding index to $\vec{\mathcal{W}}$
10 **return** $[\![\vec{\mathcal{W}}]\!]$

The strategy is simple. For computing $\vec{\mathcal{W}}_i$ for $i > 0$, we compare the pattern to itself shifted i times to the right. We then find the first position where the overlapping parts differ. Here, we additionally use the most-significant non-zero bit protocol (msnzb) from [5]. The msnzb protocol takes as input a secret-shared vector of bits and outputs a vector of same length, where only the left-most non-zero bit is set. If the input contains no set bits, the output consists of only zeros. Thus, we always find the first such position where the pattern does not match with its shifted version, which is exactly the position k, such that $Y[k] \neq Y[k+i]$.

Note that if hiding the length of the pattern is required, then we could always calculate the witness array as a fixed-length array with padded zeroes for positions after the position $\lceil \frac{m}{2} \rceil - 1$, which can be done in an oblivious manner.

3.3.2 An optimal protocol based on balanced binary tree

We now present a protocol, which is based on an optimal parallel algorithm which has time-complexity $O(\log m)$ and work-complexity $O(n)$, where m is the length of the pattern and n the length of the text. It is optimal, since no sequential algorithm can have better than linear complexity [8].

The algorithm uses the *duel* function to quickly eliminate possibilities where the pattern cannot match the text. The *duel* function eliminates one of two possible matching positions $i \neq j$ of a pattern P in a text T, based on the witness array of P. The constraint is that $|i - j| < \pi(P)$, since only then is it guaranteed that one of the positions definitely is not a matching position. This is because if P would match at both positions, then the period of P would be smaller or equal to $|i - j|$, but we assumed $|i - j| < \pi(P)$. The duel function itself is presented as Algorithm 3.

Algorithm 3: Duel function

Input: Text T of length n
Input: Pattern P of length m, such that $m \leq n$
Input: Witness array $\vec{\mathcal{W}}$ for P
Input: indices i, j such that $j > i$ and $j - i < \pi(P)$
Output: One of i or j, eliminating the other as a candidate position for a match

```
1 k ← W_{j−i}
2 if T[k + j] = P[k] then
3 |   return j
4 return i
```

We briefly argue why the algorithm is correct. Notice that $P[k] \neq P[k + j - i]$, since $k = \mathcal{W}_{j-i}$. Thus, both $T[k + j] = P[k]$ and $T[k + j] = P[k + j - i]$ cannot be true at the same time, if $i \neq j$. Clearly, if $T[k + j] \neq P[k]$, then P does not match T at position j and so we return i instead. However, if $T[k + j] = P[k]$, then clearly $T[k + j] \neq P[k + j - i]$ and then a matching is impossible at position i, in which case we return j. Figure 1 visualizes this fact for better intuition. Note that there is no guarantee that a matching actually occurs in the returned position, but we are definitely eliminating the other position as a candidate.

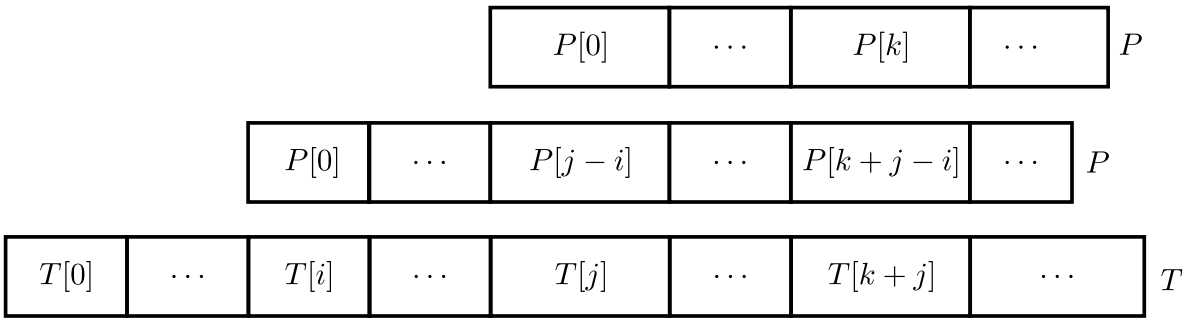

Figure 1: Eliminating one position from i and j as a candidate for matching pattern P in text T. Here $k = \phi_{wit}(j - i)$ and therefore $P[k] \neq P[k + j - i]$.

For the string-matching protocol, we divide the input text into blocks of size $\lceil \frac{m}{2} \rceil$, where m is the length of the pattern. Then, a non-periodic pattern can only match the text at most once in each block. In each block, we use the duel function to eliminate all but one possibility of a matching position. Also, we can process each block separately in parallel. Then finally we use the brute-force protocol based on the remaining candidate indices to find the actual match array.

To find the single possible matching index in a block, we apply the duel function using a balanced binary tree strategy. We write all the candidate positions to the leaves of a balanced binary tree, and each inner node holds the result for the duel function on its child nodes. This strategy is illustrated on Figure 2.

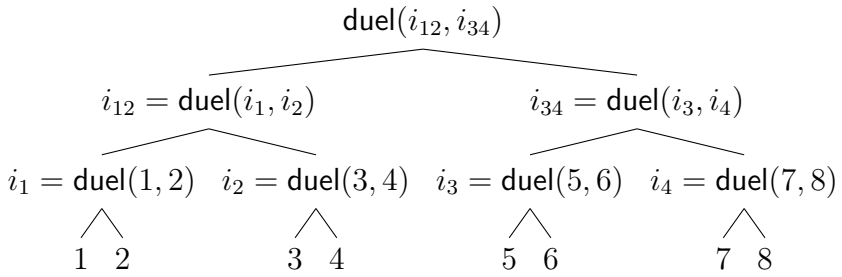

Figure 2: Applying binary tree strategy to eliminate all but one position in $\lceil \frac{m}{2} \rceil = 8$ size block

Note that each level of the tree can be evaluated in parallel with constant time in PRAM, and the time-complexity for evaluating the whole tree is therefore $O(\log \lceil \frac{m}{2} \rceil)$. For our string-matching protocol, we thus need to develop a constant-round protocol for applying the duel function on many pairs of indices in parallel. This protocol is presented as Algorithm 4. For simplicity, we assume that the length of the pattern is $m = 2^k$, which means the the tree will be a complete balanced binary tree.

We assume that the compared candidate indices $[\![\vec{I}]\!]$ and $[\![\vec{J}]\!]$ are also secret-shared, since we will require this in our string-matching algorithm. The parallelDuel protocol requires using the oblivious parallel reading protocol parallelRead from [10], since the

Algorithm 4: Computing the *duel* function in parallel for a set of index pairs (parallelDuel)

Input: Secret-shared text $[\![\vec{T}]\!]$ of length n
Input: Secret-shared non-periodic pattern $[\![\vec{P}]\!]$ with length m, such that $m \leq n$ and $m = 2^k$
Input: Secret-shared witness array $[\![\vec{\mathcal{W}}]\!]$ for $[\![\vec{P}]\!]$ with length $m/2$
Input: Two vectors of secret-shared indices $[\![\vec{I}]\!]$ and $[\![\vec{J}]\!]$, such that all indices are smaller than n and all $[\![I_i]\!] < [\![J_i]\!]$
Output: Vector of secret-shared indices $[\![\vec{\mathcal{I}}]\!]$ such that $[\![\mathcal{I}_i]\!] = \mathsf{duel}([\![I_i]\!], [\![J_i]\!])$

1 $[\![\vec{k}]\!] \leftarrow \mathsf{parallelRead}([\![\vec{\mathcal{W}}]\!], [\![\vec{J}]\!] - [\![\vec{I}]\!])$
2 $[\![\vec{r}]\!] \leftarrow [\![\vec{k}]\!] + [\![\vec{J}]\!]$
3 $[\![\vec{Y}]\!] \leftarrow \mathsf{parallelRead}([\![\vec{P}]\!], [\![\vec{k}]\!])$
4 $[\![\vec{Z}]\!] \leftarrow \mathsf{parallelRead}([\![\vec{T}]\!], [\![\vec{r}]\!])$
5 $[\![\vec{d}]\!] \leftarrow [\![\vec{Y}]\!] \stackrel{?}{=} [\![\vec{Z}]\!]$
6 $[\![\vec{\mathcal{I}}]\!] \leftarrow [\![\vec{d}]\!] \cdot [\![\vec{J}]\!] + ([\![\vec{1}]\!] - [\![\vec{d}]\!]) \cdot [\![\vec{I}]\!]$ `// if` $d_i = 1$ `choose` J_i`, else` I_i
7 **return** $[\![\vec{\mathcal{I}}]\!]$

candidate indices and also the witness array are private. The parallelRead protocol takes as input a secret-shared array and a vector of secret-shared indices and returns a vector which contains exactly the elements from the array corresponding to the given indices.

Remember that after we find the single remaining candidate index from each block, we still need to apply a brute-force protocol to check which candidate positions actually give a matching. The brute-force protocol we presented in Algorithm 1 is not suitable for this task, since we have to check for matchings at specific indices and we do not want to go through all possible matching positions, since it would defeat the purpose of the optimal protocol. We thus present a similar protocol, which checks for matchings only among a given set of secret-shared positions as Algorithm 5.

Here, we additionally need the parallelWrite protocol from [10]. The special case for $i = 0$ is needed because $[\![\vec{\mathcal{L}}]\!]$ will always contain 0 if a matching does not occur in one of the candidate positions. However, this does not mean that the pattern matches at position 0, thus we must explicitly check for matching at position 0 separately. On line 10 we obliviously write $[\![1]\!]$ into the match array $[\![\vec{\mathcal{M}}]\!]$ on all indices specified in $[\![\vec{\mathcal{L}}]\!]$).

We have now constructed all the building blocks that we need for our optimal string-matching algorithm for non-periodic pattern. The resulting protocol is presented as Algorithm 6.

We first divide all the possible $n - m + 1$ matching indices into blocks of size $\lceil \frac{m}{2} \rceil$ to a total of $\lceil \frac{2n}{m} \rceil$ blocks. We then process each block in parallel using the balanced binary tree strategy discussed previously. For each block, we sequentially apply the parallelDuel protocol (Algorithm 4) for $\log \lceil \frac{m}{2} \rceil$ iterations, until only a single candidate index is left for each block. We then apply the matchBruteforce protocol (Algorithm 5)

Algorithm 5: Brute-force string-matching algorithm according to a set of indices (MatchBruteforce)

Input: Secret-shared text $[\![\vec{T}]\!]$ of length n
Input: Secret-shared pattern $[\![\vec{P}]\!]$ of length m, such that $m \leq n$
Input: Secret-shared vector of indices $[\![\vec{\mathcal{J}}]\!]$ of length l, such that $[\![\vec{P}]\!]$ cannot match $[\![\vec{T}]\!]$ at position i if $i \notin \vec{\mathcal{J}}$
Output: The match array $[\![\vec{\mathcal{M}}]\!]$ of length $n - m + 1$ showing all positions where $[\![\vec{P}]\!]$ matches $[\![\vec{T}]\!]$

for $i := 0$ **to** $l - 1$ **do in parallel**
  foreach $j \in \{0, \ldots, m-1\}$ **do** $[\![\mathcal{I}_j]\!] \leftarrow [\![\mathcal{J}_i]\!] + [\![j]\!]$
  $[\![\vec{C}]\!] \leftarrow \mathsf{parallelRead}([\![\vec{T}]\!], [\![\vec{\mathcal{I}}]\!])$
  $[\![\vec{d}]\!] \leftarrow [\![\vec{C}]\!] \stackrel{?}{\neq} [\![\vec{P}]\!]$
  $[\![b]\!] \leftarrow \mathsf{sum}([\![\vec{d}]\!]) \stackrel{?}{=} [\![0]\!]$
  $[\![\mathcal{L}_i]\!] \leftarrow [\![b]\!] \cdot [\![\mathcal{J}_i]\!]$
  if $i = 0$ **then**
    $[\![m_0]\!] \leftarrow ([\![b]\!] \stackrel{?}{=} [\![1]\!]) \wedge ([\![\mathcal{J}_i]\!] \stackrel{?}{=} [\![0]\!])$
foreach $i \in \{0, \ldots, n-m\}$ **do** $[\![\mathcal{M}_i]\!] \leftarrow [\![0]\!]$
$\mathsf{parallelWrite}([\![\vec{\mathcal{M}}]\!], [\![1]\!], [\![\vec{\mathcal{L}}]\!])$
$[\![\mathcal{M}_0]\!] \leftarrow [\![m_0]\!]$
return $[\![\vec{\mathcal{M}}]\!]$

on the remaining candidate indices.

3.3.3 A faster algorithm based on doubly-logarithmic tree

The time-complexity of the optimal string-matching algorithm can actually be reduced to $O(\log \log m)$ by applying a different strategy in eliminating all but candidate matching index in a block. Instead of using a balanced binary tree, we can use a *doubly-logarithmic tree*.

The doubly-logarithmic tree is defined as follows. Assume that our block length $\lceil \frac{m}{2} \rceil$ is 2^{2^k}, which defines the number of leaves of the tree. Then the root of the tree is defined to have at most $2^{2^{k-1}} = \sqrt{\lceil \frac{m}{2} \rceil}$ children. Similarly, each node at the i-th level will have at most $2^{2^{k-1-i}}$ children. The nodes at the k-th level have 2 children as do the nodes at level level $k-1$. An example of a doubly-logarithmic tree for 16 leaves is shown on Figure 3.

Algorithm 6: String-matching algorithm for a non-periodic pattern using balanced binary tree strategy

Input: Secret-shared string $[\![\vec{T}]\!]$ of length n
Input: Secret-shared non-periodic string $[\![\vec{P}]\!]$ with length m, such that $m \leq n$ and $m = 2^k$
Input: Secret-shared witness array $[\![\vec{\mathcal{W}}]\!]$ for $[\![\vec{P}]\!]$ with length $m/2$
Output: The match array $[\![\vec{\mathcal{M}}]\!]$ of length $n - m + 1$ showing all positions where $[\![\vec{P}]\!]$ matches $[\![\vec{T}]\!]$

$t \leftarrow \lceil \frac{2n}{m} \rceil$ `// number of blocks`
for $block := 0$ **to** $t - 1$ **do in parallel**
  foreach $i \in \{0, \ldots, \frac{m}{2} - 1\}$ **do** `// initialize candidate indices`
    $[\![\mathcal{I}_i]\!] \leftarrow block \cdot \frac{m}{2} + i$
  for $iter := 1$ **to** $\log_2(s)$ **do**
    $s \leftarrow m/2^{iter}$ `// block size in current iteration`
    foreach $i \in \{0, \ldots, s/2 - 1\}$ **do** $[\![I_i]\!] \leftarrow [\![\mathcal{I}_{2i}]\!]$
    foreach $j \in \{0, \ldots, s/2 - 1\}$ **do** $[\![J_j]\!] \leftarrow [\![\mathcal{I}_{2i+1}]\!]$
    $[\![\vec{\mathcal{I}}]\!] \leftarrow \mathsf{parallelDuel}([\![\vec{T}]\!], [\![\vec{P}]\!], [\![\vec{\mathcal{W}}]\!], [\![\vec{I}]\!], [\![\vec{J}]\!])$
  $[\![\mathcal{J}_{block}]\!] \leftarrow [\![\mathcal{I}_0]\!]$
$[\![\vec{\mathcal{M}}]\!] \leftarrow \mathsf{MatchBruteforce}([\![\vec{T}]\!], [\![\vec{P}]\!], [\![\vec{\mathcal{J}}]\!])$
return $[\![\vec{\mathcal{M}}]\!]$

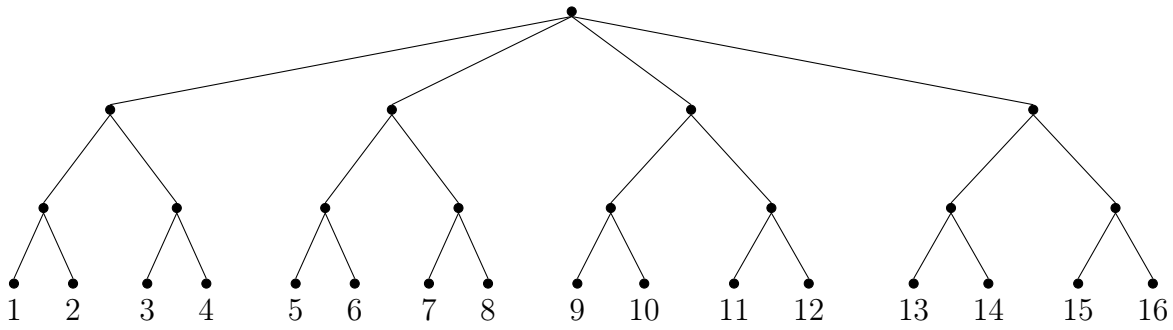

Figure 3: Applying doubly-logarithmic tree strategy to eliminate all but one position in $\lceil \frac{m}{2} \rceil = 16$ size block

For n leaves, the doubly-logarithmic tree has height $\log \log n + 1$. To use the doubly-logarithmic tree strategy for applying the duel function in a single block, we need to have a constant-round protocol for eliminating all but one index from an arbitrary number of candidates to achieve the desired $O(\log \log n)$ round-complexity. That is, we need to be able to process each level of the tree in a constant number of rounds. For this reason we introduce another version of the parallelDuel protocol that takes as input a vector of candidates and outputs only one of them. We will call this the vectorizedParallelDuel protocol since it operates on an arbitrarily large vector of indices.

The idea how to achieve this is we compute the duel function between all possible pairs in the given index set $\vec{\mathcal{I}}$. Then, a position i is the remaining possible candidate if and only if all calls to *duel*(i,j) for $j \in \vec{\mathcal{I}}, j \neq i$ return i as the result. If none of the positions satisfy this property, then we return an arbitrary index. The corresponding protocol is presented as Algorithm 7.

Algorithm 7: Using the *duel* function in parallel for a set of indices to eliminate all but one candidate (vectorizedParallelDuel)

Input: Secret-shared text $[\![\vec{T}]\!]$ of length n
Input: Secret-shared non-periodic pattern $[\![\vec{P}]\!]$ with length m, such that $m \leq n$ and $m = 2^k$
Input: Secret-shared witness array $[\![\vec{\mathcal{W}}]\!]$ for $[\![\vec{P}]\!]$ with length $m/2$
Input: A secret-shared vector of indices $[\![\vec{\mathcal{I}}]\!]$ in growing order with length l, such that all indices are smaller than n
Output: A single secret-shared index such that all other indices from $[\![\vec{\mathcal{I}}]\!]$ are eliminated as possible matching candidates

1 **for** $i := 0$ **to** $l - 1$ **do in parallel**
2 **for** $j := 0$ **to** $l - 1$ **do**
3 **if** $j < i$ **then**
4 $[\![I_j]\!] \leftarrow [\![\mathcal{I}_j]\!]$
5 $[\![J_j]\!] \leftarrow [\![\mathcal{I}_i]\!]$
6 **else if** $i < j$ **then**
7 $[\![I_j]\!] \leftarrow [\![\mathcal{I}_i]\!]$
8 $[\![J_j]\!] \leftarrow [\![\mathcal{I}_j]\!]$
9 $[\![\vec{\mathcal{J}}]\!] \leftarrow \mathsf{parallelDuel}([\![\vec{T}]\!], [\![\vec{P}]\!], [\![\vec{\mathcal{W}}]\!], [\![\vec{I}]\!], [\![\vec{J}]\!])$
10 $[\![\vec{d}]\!] \leftarrow [\![\vec{\mathcal{J}}]\!] \stackrel{?}{\neq} [\![\vec{I}]\!]$
11 $[\![b_i]\!] \leftarrow \mathsf{sum}([\![d]\!]) \stackrel{?}{=} [\![0]\!]$
12 $[\![\vec{\mathcal{I}}]\!] \leftarrow [\![\vec{\mathcal{I}}]\!] \cdot [\![\vec{b}]\!]$
13 $[\![\mathcal{Y}]\!] \leftarrow \mathsf{sum}([\![\vec{\mathcal{I}}]\!])$
14 $[\![\mathcal{Y}]\!] \leftarrow [\![\mathcal{I}_0]\!] \cdot ([\![\mathcal{Y}]\!] \stackrel{?}{=} [\![0]\!]) + [\![\mathcal{Y}]\!] \cdot ([\![\mathcal{Y}]\!] \stackrel{?}{\neq} [\![0]\!])$
15 **return** $[\![\mathcal{Y}]\!]$

Notice that on line 14 we explicitly check if none of the indices was suitable as a candidate, and in that case arbitrarily return the first element of $[\![\vec{\mathcal{I}}]\!]$. Also, in the presented protocol we compute both *duel*($[\![\mathcal{I}_i]\!]$,$[\![\mathcal{I}_j]\!]$) and *duel*($[\![\mathcal{I}_j]\!]$,$[\![\mathcal{I}_i]\!]$) for all pairs i, j. This redundancy can of course be optimized out, as we have done in our implementation, but the resulting protocol would be very complicated to write down so we chose to present a more easily understandable description.

Using the vectorizedParallelDuel protocol, we can construct a string-matching protocol following the doubly-logarithmic tree. This is very similar to the binary tree version

in Algorithm 6, so we omit the detailed description here.

The PRAM algorithm using the doubly-logarithmic tree, although it is much faster than the binary tree, is not optimal in the number of operations performed and has work-complexity $O(n \cdot \log\log n)$. However, combining the two approaches leads to an asymptotically optimal and fast algorithm. The strategy involves using the binary tree version up to $\lceil \log\log\log \frac{m}{2} \rceil$ levels, and then switching to the doubly-logarithmic tree. This results in a string-matching algorithm with time complexity $O(\log\log m)$ and work-complexity $O(n)$. This strategy is called *accelerated cascading* [8] and it can be used for all similar computations, for which a constant-time algorithm exists for evaluating each level of the tree. Out of curiosity, we also implemented this combined version of the string-matching protocol.

4 Experimental results

We performed benchmarks for 4 versions of the string-matching protocol:

1. Brute-force (Algorithm 1)
2. Protocol using balanced binary tree strategy (Algorithm 6)
3. Protocol using doubly-logarithmic tree strategy
4. Protocol using combined strategy

All the protocols were implemented in SecreC [4]. The most difficult aspect of implementing all the presented protocols is actually constructing the large input vectors to perform the required steps in the **do in parallel** loops using constant rounds.

The benchmarks were performed on a cluster of three nodes hosting Sharemind. All nodes had 48 GB of RAM and a 12-core 3GHz Intel CPU supporting HyperThreading. The nodes were connected to a LAN with 1 Gbps full duplex links, however, the bandwidth was throttled to an average of 100 Mbps at the time of the benchmarks. The results for different text sizes are presented below.

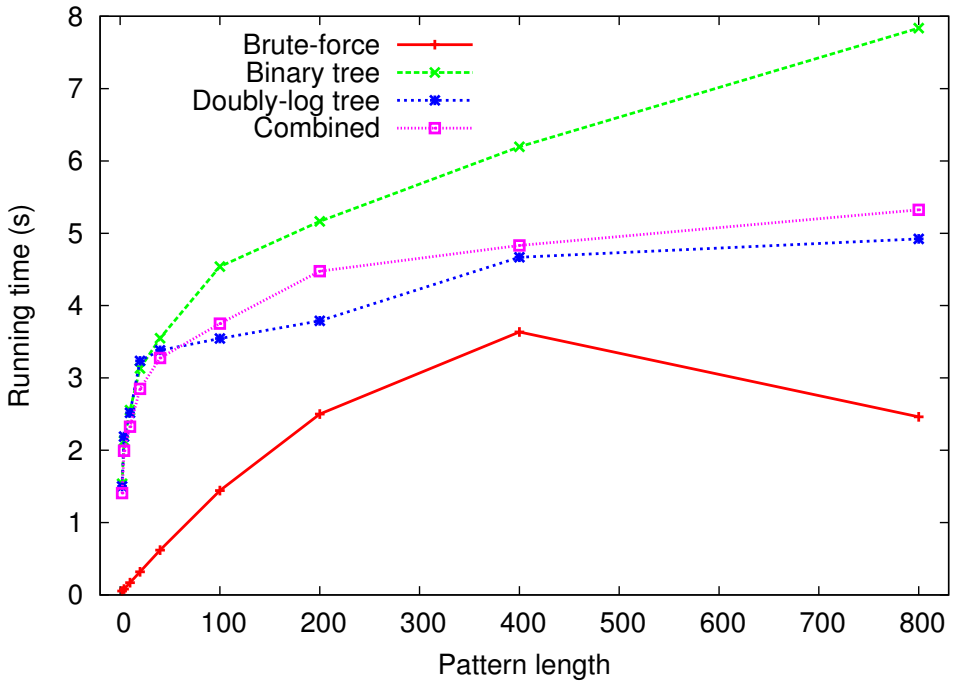

Figure 4: Comparison of string-matching protocols for text size 1000

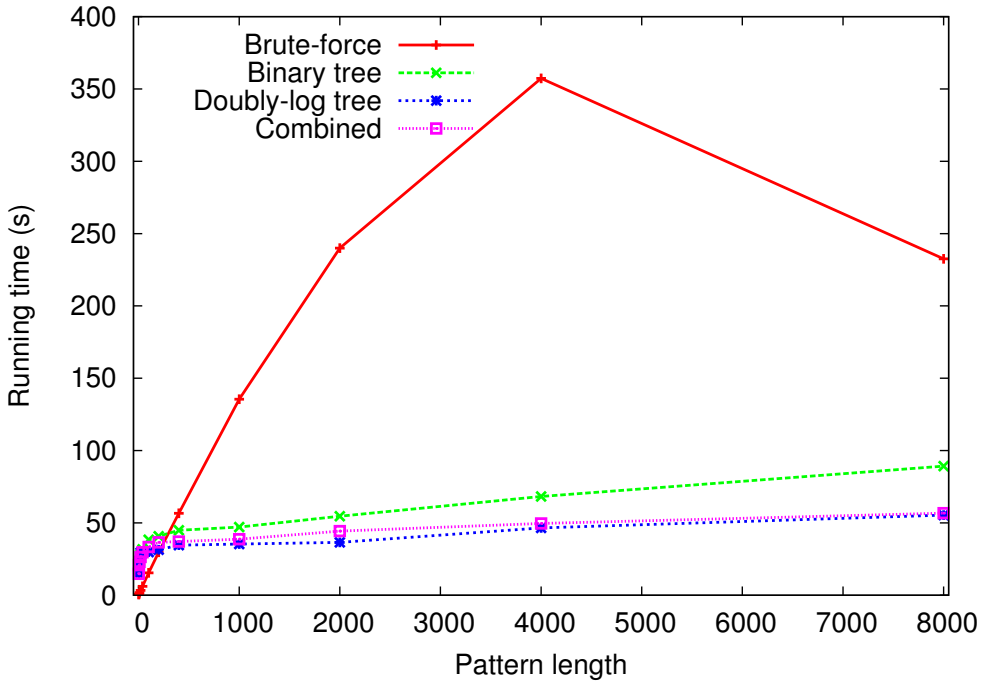

Figure 5: Comparison of string-matching protocols for text size 10000

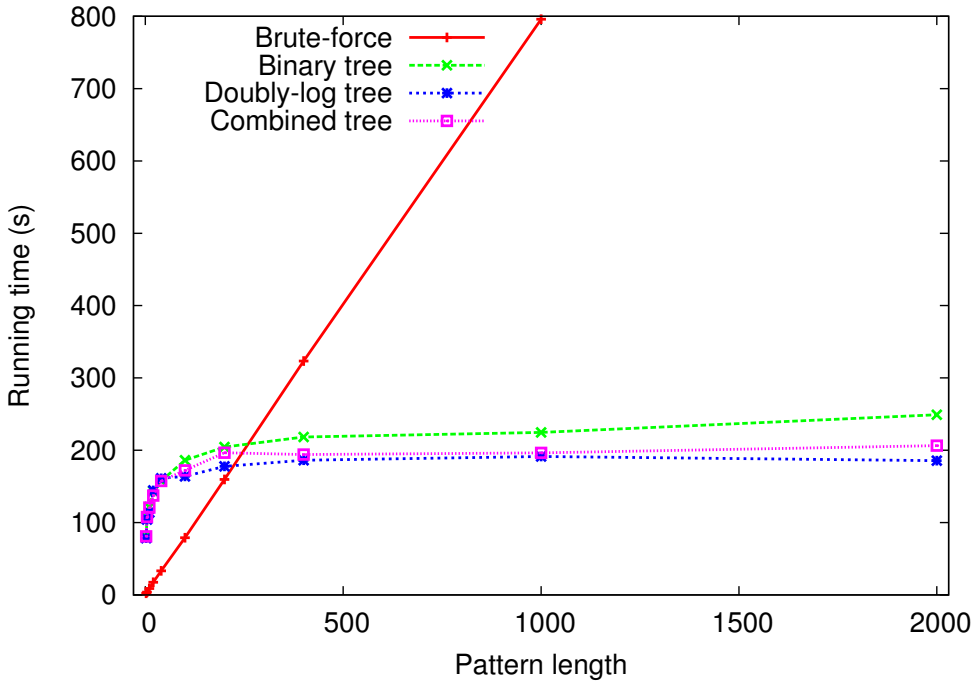

Figure 6: Comparison of string-matching protocols for text size 50000

For the text size 1000, the brute-force protocol is the fastest, since the number of comparison operations required is relatively small and can be done in a single protocol call, without saturating the network links. However, as the results clearly show, the running time of the brute-force protocol grows linearly for larger text sizes, whereas the protocols based on the PRAM algorithms show a logarithmic trend for running time and are much faster starting from pattern length ~200.

The results show clearly, how the trade-off between higher round-complexity but larger communication pays off for larger input sizes. The three protocols based on the PRAM algorithms have very similar performance, but the doubly-logarithmic version seems to be the fastest of the three, most probably since its round-complexity is the smallest. These results actually lead to an important observation. For optimal results, the tree shape that is used for finding the matching index in each block of the text should be chosen such, that a maximum number of operations are done in parallel, without crossing the available bandwidth bound with required communication. Probably a tree with even lower height than the doubly-logarithmic could be used for even better results.

Note that the three protocols based on the PRAM algorithms still require the witness array to be computed for the pattern. This can be done only once, if the same pattern is matched against different texts, however it still presents an additional overhead. We only implemented a brute-force protocol for computing the witness array, which has quadratic complexity with respect to the pattern length, and for larger pattern lengths this almost cancels out the performance difference between the brute-force string-matching and the other three protocols.

However, a logarithmic-time algorithm for computing the witness array exists, but we did not have time to implement it and leave this as future work.

5 Conclusion

In this report, we showed how an efficient privacy-preserving string-matching algorithm could be implemented in an SMC setting using the algorithm design paradigms of parallel algorithms. The experimental results show that this approach can significantly increase the performance of protocols, especially for larger inputs. This approach was made possible by the fact that we were able to use efficient protocols for oblivious data accesses, which is a very recent advancement in the field.

We focused on the accelerated cascading method to perform string-matching efficiently, however, this approach could also be applied for many other computations, e.g finding the maximum/minimum number from an array. Also, the class of solved computational problems in the PRAM model is vast and there is still many algorithms that can be tried for increasing efficiency of SMC computations. For example, an interesting problem to solve is answering on-line string-matching queries efficiently by using suffix trees, for which also PRAM methods are available.

Acknowledgment

The author of this report has received the Skype and IT Academy Master's Scholarship for the academic year 2014/15, funded by Estonian Information Technology Foundation and Skype Technologies OÜ.

References

[1] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In J. Simon, editor, *STOC*, pages 1–10. ACM, 1988.

[2] G. Blakley. Safeguarding cryptographic keys. In *Proceedings of the 1979 AFIPS National Computer Conference*, pages 313–317, Monval, NJ, USA, 1979. AFIPS Press.

[3] D. Bogdanov. *Sharemind: programmable secure computations with practical applications*. PhD thesis, University of Tartu, 2013.

[4] D. Bogdanov, P. Laud, and J. Randmets. Domain-Polymorphic Programming of Privacy-Preserving Applications. In *Proceedings of the First ACM Workshop on Language Support for Privacy-enhancing Technologies, PETShop '13*, ACM Digital Library, pages 23–26. ACM, 2013.

[5] D. Bogdanov, M. Niitsoo, T. Toft, and J. Willemson. High-performance secure multi-party computation for data mining applications. *International Journal of Information Security*, 11(6):403–418, 2012.

[6] D. Chaum, C. Crépeau, and I. Damgård. Multiparty unconditionally secure protocols (extended abstract). In J. Simon, editor, *STOC*, pages 11–19. ACM, 1988.

[7] I. Damgård, M. Fitzi, E. Kiltz, J. B. Nielsen, and T. Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In S. Halevi and T. Rabin, editors, *Theory of Cryptography, Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006, Proceedings*, volume 3876 of *Lecture Notes in Computer Science*, pages 285–304. Springer, 2006.

[8] J. JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.

[9] D. E. Knuth, J. H. M. Jr., and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

[10] P. Laud. Efficient oblivious parallel array reads and writes for secure multiparty computation. Cryptology ePrint Archive, Report 2014/630, 2014. `http://eprint.iacr.org/`.

[11] A. Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[12] RFC 3629 — UTF-8, a transformation format of ISO 10646. `http://tools.ietf.org/html/rfc3629`, November 2003.