Universally composable privacy preserving finite automata execution with low online and offline complexity*

Peeter Laud and Jan Willemson

Cybernetica AS

Abstract. In this paper, we propose efficient protocols to obliviously execute non-deterministic and deterministic finite automata (NFA and DFA) in the arithmetic black box (ABB) model. In contrast to previous approaches, our protocols do not use expensive public-key operations, relying instead only on computation with secret-shared values. Additionally, the complexity of our protocols is largely offline. In particular, if the DFA is available during the precomputation phase, then the online complexity of evaluating it on an input string requires a small constant number of operations per character. This makes our protocols highly suitable for certain outsourcing applications.

Keywords. Finite automata, secure multiparty computation, arithmetic black box

1 Introduction

Finite automata (FA) are among the most often used algorithmic tools for analyzing textual data. They are used in filtering spam, recognizing malware, genetic analysis, log mining, etc. Often, these applications make use of data with various owners, having certain expectations of privacy (e.g. genetic microdata may reveal the subject's medical condition, network log items may show security vulnerabilities, etc.). Hence, the problem of executing finite automata in a privacy-preserving manner is highly relevant.

Usually, the execution of the FA does not comprise the whole algorithm for a particular task. E.g. in spam filtering, the automata are used to recognize whether the e-mail message matches certain signatures. Afterwards, these matchings are suitably weighted and combined to decide whether the message was spam or not. Hence the result of privacy-preserving FA execution should be obtained in a manner that is easily usable by further secure computation algorithms — we need *composable* protocols for FA execution (as well as for other algorithmic tasks that are used in the application).

* The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 284731 (Usable and Efficient Secure Multiparty Computation, UaESMC), and from the European Regional Development Fund through the Estonian Center of Excellence in Computer Science, EXCS.

In the setting of secure multiparty computation, the composability is well-captured by the notion of the *arithmetic black box (ABB)* [9, 22]. It is an ideal functionality that stores the data items that we wish to keep private and performs computations with them. It is realized with the help of suitable cryptographic protocols. Higher-level protocols use the protocols of ABB as their subprotocols. To prove the security of the higher-level protocol, it is sufficient to consider its execution together with the ideal ABB-functionality.

In this paper, we are considering algorithms for FA execution where the description of the FA, as well as the input string have been stored in the ABB. As the result of the algorithm, the information about the reached state(s) of the automaton is also stored in the ABB. During the computation, no information about the input string (except its length) nor the FA (except the number of its states) is leaked outside the ABB.

Without security considerations, the execution of a DFA is trivial — at each step, read the next state of the automaton from the transition table, using the current state and the next character of the input string. Inside the ABB, this is complicated due to the lack of an efficient privacy-preserving look-up operation. In this paper, we consider DFA execution algorithms making use of only such operations that are typically provided in an efficient manner by ABB implementations.

NFAs have the same expressive power as DFAs, but they can offer considerably smaller size of the automata. However, this efficiency comes with a price of a more complicated execution paradigm. Generally, the subsequent states are not uniquely determined and this non-uniqueness is classically implemented by using backtracking. However, backtracking and other constructions with complex control flow are inefficient to implement on ABB; our proposal is based on different ideas [21].

For certain tasks, it is possible to partition the necessary computations into the offline part — these that can be done without knowing the actual inputs —, and the online part — these that require the inputs. To increase the responsiveness of algorithm implementations, one tries to minimize the online computations by structuring the algorithm in such a way that more computations can be performed off-line. In case of FA execution, it makes sense to consider even three stages of input availability — offline, FA-only, and online. In the FA-only stage, the description of the automaton is already available for the computation, but the input string is still missing. This naturally corresponds to certain practical settings, e.g. spam filtering, where the filters are known before the e-mail messages to which they are applied.

There are applications where the input string is private, but the description of the FA is "public" — known to all parties implementing the ABB. The outsourcing of spam filtering into the cloud can be one of such applications. Here the service provider sets up the multiparty computation for realizing the ABB, deploying the servers implementing the ABB at different cloud providers and providing them all with the descriptions of filters. The customers upload their e-mail messages to the ABB and receive back their classification, with no sin-

gle server learning the contents of the messages. The DFA execution algorithms presented in this paper are extremely well suited for the use in these settings, rapidly returning answers while preserving the privacy of the customers against subsets of servers.

Our contribution For a DFA with m states over an alphabet with n characters, we propose an execution algorithm in the ABB model, processing the input string character-wise and performing $(1 + o(1))mn$ ABB multiplications in the offline stage, $(1 + o(1))mn$ ABB multiplications in the FA-only stage and only a single ABB multiplication in the online stage (all these costs are per character of the input string). If the DFA description is public, then the FA-only stage has zero computational cost. Also, for a particular ABB, the cost of offline stage can be reduced to $O(\sqrt{mn})$. For a different ABB, all computations of the FA-only stage may be moved to the on-line stage without increasing the cost of the latter. As usual, the additions and public linear combinations of private values are considered to have zero execution cost due to being local operations in all implementations. The online performance of this protocol exceeds all other protocols in the state of the art, all of which perform at least $O(m + n)$ online work (computation and/or communication; whichever is the bottleneck for the particular method) per character. Also, we stress that our protocol works in the ABB model, making DFA execution usable as a subprotocol in protocols for more complex tasks.

As a separate contribution, we also propose private execution algorithm for the NFAs. In case the automaton description is also captured in the ABB, its online complexity is $m(mn + m + 1)$ multiplications in 3 rounds per input character. If the description of the automaton is public, these complexities drop to $m(m + 1)$ and 2, respectively. In both cases, the amount of required precomputation is $O(m \log m)$ per input character.

Interestingly, the protocols in this paper are the first *information-theoretically* secure protocols for FA execution, if an information-theoretically secure ABB is used. All previous protocols have used cryptographic constructions (encryption) that rely on computational hardness assumptions for security.

To the best of our knowledge, this paper presents the first protocol for secure NFA execution.

Structure of the paper We review work related to privacy-preserving DFA execution in Sec. 2, describe the necessary preliminaries and notation in Sec. 3, and present our basic DFA protocol in Sec. 4. In Sec. 5 we show how to reduce the complexity of the offline phase. Sec. 7 compares the performance of our protocol with the protocols constructed from generic building blocks with good asymptotic complexity. As the last part of the contribution, Sec. 8 presents our private NFA execution protocol. Finally, we draw the conclusions in Sec. 9.

2 Related work

The possibility of *secure multiparty computation* SMC in general has been known for a long time [25, 14]. Unfortunately, the generic protocols resulting from these possibility results are too inefficient in practice for anything but the simplest functionalities.

The question of privacy-preserving DFA execution seems to have been first considered by Troncoso-Pastoriza et al. [23]. In their setting, there are two parties, one of them (Alice) knowing the DFA description, while the other one (Bob) knows the input string. I.e. they were not considering the ABB model. During the computation, the current state of the DFA is additively shared between these two parties (*modulo* m). Rotations of this table combined with homomorphic encryptions of the shares of the current state and an oblivious transfer allow parties to learn the shares of the next state. The protocol has been improved in several ways by Blanton and Aliasgari [1]. Beside reducing the complexity, they also adapt the protocol for the ABB model. Through a clever reshaping of the transition table, they are actually able to get the communication complexity down to $O(\sqrt{mn})$ per character of the input string. Their techniques resemble those of *private information retrieval* (PIR) protocols using homomorphic encryption [17]; we believe that through a more thorough application of these techniques, the complexity per character might even be lowered to $O(\log^2 mn)$. Nevertheless, we are not pursuing these avenues of research here, as the constant hidden inside the O-notation makes these protocols less efficient than our protocols (which do not use expensive public-key operations) for realistic problem sizes.

Troncoso-Pastoriza et al.'s protocol has been adapted to malicious setting by Gennaro et al. [11], using zero-knowledge protocols to force honest behaviour of parties. Malicious setting is also considered by Wei and Reiter [24]. They propose two-party (called "client" and "server") protocols in the ABB model that are secure against the malicious behaviour of the server party. Similarly to the protcols in this paper, they treat the DFA transition table as a polynomial over some field. All protocols described so far make heavy use of homomorphic encryption, some of them also requiring some extra properties [4].

Garbled circuits originally proposed by Yao [25] can be adapted for DFA execution [10, 19]. In these protocols, the party knowing the transition table δ constructs and garbles "δ-gates", connected sequentially. The party with the input string executes the circuit and learns the last state of the DFA or some property of it.

3 Preliminaries

In this work, a *deterministic finite automaton (DFA)* over the alphabet Σ is a tuple $A = (Q, \delta, q_0)$, where Q is the set of states, $q_0 \in Q$ the starting state of the automaton, and $\delta : Q \times \Sigma \to Q$ the transition table. Given a string $s = a_1 \cdots a_\ell \in \Sigma^*$, the DFA A maps s to the state $\boldsymbol{\delta}_A(s) = \delta(\cdots(\delta(\delta(q_0, a_1), a_2)\cdots), a_\ell)$. Compared to usual definition of DFA, we have left out the set of accepting states

from the structure of a DFA, and consider the function $\boldsymbol{\delta}_A : \Sigma^* \to Q$ as the behaviour of A; this is also computed by our protocol. This omission is justified by our work in the ABB model, as the computation can continue from the last state reached by the DFA in any manner deemed necessary by the designer of the whole system.

Similarly, a *nondeterministic finite automaton (NFA)* over the alphabet Σ is a tuple $A = (Q, \delta, q_0)$, with Q and q_0 meaning the same as before, and $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ being the transition table (here $\mathcal{P}(Q)$ is the set of subsets of Q). We can extend δ to sets of states — $\delta(\mathcal{Q}, a) = \bigcup_{q \in \mathcal{Q}} \delta(q, a)$ for $\mathcal{Q} \subseteq Q$ and $a \in \Sigma$. Given a string $s = a_1 \cdots a_\ell \in \Sigma^*$, the NFA A maps s to the set of states $\boldsymbol{\delta}_A(s) = \delta(\cdots(\delta(\delta(\{q_0\}, a_1), a_2)\cdots), a_\ell)$.

The *arithmetic black box* is an ideal functionality $\mathcal{F}_{\mathsf{ABB}}$. It allows its users (a fixed number p of parties) to securely store and retrieve values, and to perform computations with them. When a party sends the command $\mathsf{store}(v)$ to $\mathcal{F}_{\mathsf{ABB}}$, where v is some value, the functionality assigns a new *handle* h (sequentially taken integers) to it by storing the pair (h, v) and sending h to all parties. If a sufficient number (depending on implementation details) of parties send the command $\mathsf{retrieve}(h)$ to $\mathcal{F}_{\mathsf{ABB}}$, it looks up (h, v) among the stored pairs and responds with v to all parties. When a sufficient number of parties send the command $\mathsf{compute}(op; h_1, \ldots, h_k; params)$ to $\mathcal{F}_{\mathsf{ABB}}$, it looks up the values $v_1, \ldots, v_k$ corresponding to the handles $h_1, \ldots, h_k$, performs the operation op (parametrized with $params$) on them, stores the result v together with a new handle h, and sends h to all parties. In this way, the parties can perform computations without revealing anything about the intermediate values or results, unless a sufficiently large coalition wants a value to be revealed. In this paper, we use the functionality $\mathcal{F}_{\mathsf{ABB}}$ to implement privacy-preserving FA execution.

The existing implementations of ABB are based on either secret sharing [7, 2, 5] or threshold homomorphic encryption [9, 15]. Fully homomorphic encryption [13] may also be used to implement ABB in a conceptually very simple way, but with prohibitively slow performance. Depending on the implementation, the ABB offers protection against a honest-but-curious, or a malicious party, or a number of parties (up to a certain limit). E.g. the implementation of the ABB by Sharemind [2] consists of three parties, providing protection against one honest-but-curious party.

Typically, the ABB performs computations with values v from some ring $\mathbb{R}$. The set of operations definitely includes addition/subtraction, multiplication of a stored value with a public value (this operation motivates the *params* in the compute-command), and multiplication. Even though all algorithms can be expressed using just these operations, most ABB implementations provide more operations (as primitive protocols) for greater efficiency of the implementations of algorithms on top of the ABB. In all ABB implementations, addition, and multiplication with a public value occur negligible costs; hence they're not counted when analyzing the complexity of protocols using the ABB. Other operations may require a variable amount of communication (in one or several rounds) between parties, and/or expensive computation. The ABB can execute several

operations in parallel; the *round complexity* of a protocol is the number of communication rounds all operations of the protocol require, when parallelized as much as possible.

It is common to use $[\![v]\!]$ to denote the value v stored in the ABB. The notation $[\![v_1]\!]$ *op* $[\![v_2]\!]$ denotes the computation of v_1 *op* v_2 by the ABB (translated to a protocol in the implementation of $\mathcal{F}_{\mathsf{ABB}}$).

4 Basic protocol for DFA execution

Our basic protocol combines the idea to consider the transition table δ of the DFA as a polynomial over a field $\mathbb{F}$ [24] with a method to move offline most of the computations for polynomial evaluation in the ABB [18]. Both ideas have been slightly improved and expanded here.

We have a DFA $A = (Q, \delta, q_0)$ with $|Q| = m$, working over the alphabet Σ with $|\Sigma| = n$. Let $\mathbb{F}$ be a finite field with at least $mn + 1$ elements; moreover, let $Q \subseteq \mathbb{F}$, $\Sigma \subseteq \mathbb{F}$ and let $\gamma \in \mathbb{F}$ be such, that $(q, a) \mapsto \gamma q + a$ is an injective mapping from $Q \times \Sigma$ to $\mathbb{F}\backslash\{0\}$.

There exists a polynomial $f : \mathbb{F} \to \mathbb{F}$, such that $f(\gamma q + a) = \delta(q, a)$ for all $q \in Q$ and $a \in \Sigma$; this polynomial has the degree of at most $mn - 1$. There exist Lagrange interpolation coefficients λ_{iqa} with $0 \leq i \leq mn - 1$, $q \in Q$, $a \in \Sigma$, depending only on m and n (i.e. these coefficients are public), such that $f(x) = \sum_{i=0}^{mn-1} c_i x^i$, where $c_i = \sum_{q \in Q} \sum_{a \in \Sigma} \lambda_{iqa} \delta(q, a)$.

Let our ABB work with values from the field $\mathbb{F}$. In this case, there exists a protocol for generating a uniformly random element of $\mathbb{F}$ inside the ABB (denote: $[\![r]\!] \xleftarrow{\$} \mathbb{F}$), and for generating a uniformly random non-zero element of $\mathbb{F}$ together with its inverse (denote: $([\![r]\!], [\![r^{-1}]\!]) \xleftarrow{\$} \mathbb{F}^*$). These protocols require a small constant number of multiplications on average [6]. Using these subprotocols, Algorithm 1 gives the protocol for executing the DFA A on an ℓ-character string. Note that all inputs to the algorithm (except for the sizes m, n and ℓ, which are public) are stored inside the ABB. Its result, the state of the DFA $[\![q_l]\!]$ after processing ℓ characters is similarly stored inside the ABB.

Correctness First we note that the polynomial $f(x) = \sum_{j=0}^{mn-1} c_j x^j$ satisfies the equality $f(\gamma q + a) = \delta(q, a)$ due to the construction of c_j in the DFA-only stage and the definition of the coefficients λ_{jqa}. We also note that the values r_i^j constructed in the offline stage are indeed the j-th powers of the values r_i.

We can now easily show that the value q_i computed by the on-line loop is equal to the state of the DFA A after processing the characters $a_1, \ldots, a_i$. For $i = 0$, this claim trivially holds. If it holds for $i - 1$, then it also holds for i:

$$q_i = \sum_{j=0}^{mn-1} z_i^j y_{ij} = \sum_{j=0}^{mn-1} (\gamma q_{i-1} + a_i)^j r_i^{-j} c_j r_i^j = f(\gamma q_{i-1} + a_i) = \delta(q_{i-1}, a) \ .$$

Algorithm 1: DFA execution protocol

Data: DFA components $[\![\delta(q,a)]\!]$ and $[\![q_0]\!]$, where $q \in Q$, $a \in \Sigma$.
Data: Characters of the input string $[\![a_1]\!], \ldots, [\![a_\ell]\!]$.
Result: Last state $[\![q_\ell]\!]$ in ABB.

offline processing
foreach $i \in \{1, \ldots, \ell\}$ **do**
    $([\![r_i]\!], [\![r_i^{-1}]\!]) \xleftarrow{\$} \mathbb{F}^*$
    for $j = 2$ **to** $mn - 1$ **do** $[\![r_i^j]\!] \leftarrow [\![r_i]\!] \cdot [\![r_i^{j-1}]\!]$
DFA-only processing
foreach $j \in \{0, \ldots, mn-1\}$ **do** $[\![c_j]\!] \leftarrow \sum_{q \in Q} \sum_{a \in \Sigma} \lambda_{jqa} [\![\delta(q,a)]\!]$
foreach $i \in \{1, \ldots, \ell\}$, $j \in \{0, \ldots, mn-1\}$ **do** $[\![y_{ij}]\!] \leftarrow [\![c_j]\!] \cdot [\![r_i^j]\!]$
online processing
for $i = 1$ **to** ℓ **do**
    $[\![z_i]\!] \leftarrow (\gamma [\![q_{i-1}]\!] + [\![a_i]\!]) \cdot [\![r_i^{-1}]\!]$
    $z_i \leftarrow \mathsf{retrieve}([\![z_i]\!])$
    $[\![q_i]\!] \leftarrow \sum_{j=0}^{mn-1} z_i^j [\![y_{ij}]\!]$

Privacy Except for computing z_i, al operations in Alg. 1 are performed either inside the ABB, or with public values. Hence all guarantees provided by the ABB against certain kinds of attacks involving certain coalitions of parties carry directly over to Alg. 1, if there weren't the computations involving the values z_i. Regarding the values z_i — they do not leak anything about the inputs to the algorithm, because each of them is a product of a non-zero secret value with a uniformly randomly distributed non-zero value. Hence z_i is also a uniformly randomly distributed element of $\mathbb{F}^*$. As independent values r_i^{-1} are used for computing different z_i, the different z_i-s are mutually independent as well. Regarding the correctness of the use of z_i-s in further computation — as these values become known to all parties, we can be sure that in the computation of q_i, correct z_i is used.

Complexity It is straightforward to count the number of operations Alg. 1 performs. In the offline stage, we perform $mn - 2$ multiplications per character of the input string. We also generate one random invertible element together with its inverse, this generation costs the same as a couple of multiplications [6] (in the ABB of Sharemind [3], the random number generation is free, while verifying that it is invertible and computing the inverse takes one multiplication, with the probability of the element being rejected being equal to $2/|\mathbb{F}|$). The round complexity of this computation is also $O(mn)$, which would be bad for online computations, but, in our opinion, does not matter for computations where latency is unimportant. We note that the offline phase could be performed in $O(1)$ rounds [6] at the cost of increasing the number of multiplications a couple of times. In the DFA-only stage, we perform a number of multiplications with constants λ_{jqa}; we count these operations as free. We also perform $mn - 1$ mul-

tiplications per character in order to compute $[\![y_{ij}]\!]$ (no multiplication is needed to obtain $[\![y_{i0}]\!]$). But if the DFA description had been public, then the values c_j would have been public, too, and the values $[\![y_{ij}]\!]$ would have been linear combinations of $[\![r_i^j]\!]$ with public coefficients. In this case, the DFA-only stage would have contained no costly operations at all. In the online stage, the only costly operation is the computation of $[\![z_i]\!]$, which takes a single multiplication of private values. Also, the retrieveoperation has the complexity similar to a multiplication in most implementations of the ABB.

5 Improving offline performance

We will now consider the ABB implementation of Sharemind [3] and show how it can be leveraged to speed up the offline stage of Alg. 1, the goal of which was to compute $[\![v^2]\!], \ldots, [\![v^k]\!]$ from $[\![v]\!]$ and $k \in \mathbb{N}$. Let us give a short overview of the relevant protocols in Sharemind.

The Sharemind ABB is realized by three parties, offering protection against passive attacks by one of the parties. The ABB stores elements of some ring $\mathbb{R}$; a value $v \in \mathbb{R}$ is stored in the ABB as $[\![v]\!] = ([\![v]\!]_1, [\![v]\!]_2, [\![v]\!]_3) \in \mathbb{R}^3$ satisfying $[\![v]\!]_1 + [\![v]\!]_2 + [\![v]\!]_3 = v$, where the *share* $[\![v]\!]_i$ is kept by the i-th party P_i. Messages depending on these shares are sent among the parties, hence it is important to rerandomize $[\![v]\!]$ before each use. The *resharing* protocol [3, Algorithm 1] (repeated here as Alg. 5 in Appendix A) is used for this rerandomization. We note that in this algorithm, the generation and distribution of random elements can take place offline. Even better, only random seeds can be distributed ahead of the computation and new elements of $\mathbb{R}$ generated from them as needed. Hence we consider the resharing protocol to involve only local operations and have the cost 0 in our complexity analysis.

Sharemind's multiplication protocol [3, Algorithm 2] (repeated as Alg. 6 in Appendix A) is based on the equality $([\![u]\!]_1 + [\![u]\!]_2 + [\![u]\!]_3)([\![v]\!]_1 + [\![v]\!]_2 + [\![v]\!]_3) = \sum_{i,j=1}^{3} [\![u]\!]_i [\![v]\!]_j$. After the party P_i has sent $[\![u]\!]_i$ and $[\![v]\!]_i$ to party P_{i+1} (here and subsequently, all party indices are *modulo* 3), each of these nine components of the sum can be computed by one of the parties. We see that in order to perform one multiplication in Sharemind, six elements of $\mathbb{R}$ have to be sent from one party to another. All these can be done in parallel. The multiplication protocol is secure against one honest-but-curious party [3, Theorem 2].

We see that sometimes the multiplication or a series of multiplications can be performed more efficiently. To compute $[\![u^2]\!]$ from $[\![u]\!]$, only $[\![u]\!]_i$ has to be sent from P_i to P_{i+1}. To compute $([\![uv_1]\!], \ldots, [\![uv_n]\!])$ from $[\![u]\!]$ and $([\![v_1]\!], \ldots, [\![v_n]\!])$, we start n copies of the multiplication protocol, but the shares $[\![u]\!]_i$ have to be sent only once. The security of the protocol is not affected by such optimizations.

Alg. 1 requires the ring $\mathbb{R}$ to be a field $\mathbb{F}$. In computing $([\![v^2]\!], \ldots, [\![v^k]\!])$ from $[\![v]\!]$, more substantive optimizations are possible if we take $\mathbb{F}$ to be of characteristic 2. In this case, the cardinality of $\mathbb{F}$ is a power of 2 and the equality $1 + 1 = 0$ holds. We note that squaring a value in the ABB now requires only local operations: $(x_1 + x_2 + x_3)^2 = x_1^2 + x_2^2 + x_3^2$ if the characteristic of $\mathbb{F}$ is 2.

Similarly, if parties P_i have sent the share $[\![v]\!]_i$ to parties P_{i+1} (as they do in lines 2&4 of Alg.6), then they have also sent the shares $[\![v^{2^n}]\!]_i$ for all $n \in \mathbb{N}$. The algorithm for computing the powers of $[\![v]\!]$ up to $[\![v^k]\!]$ is given as Alg. 2.

Algorithm 2: Computing $([\![v^2]\!], \ldots, [\![v^k]\!])$ from $[\![v]\!]$

Data: $k \in \mathbb{N}$ and the value $[\![v]\!]$, where $v \in \mathbb{F}$, char $\mathbb{F} = 2$
Result: Values $[\![u_0]\!], \ldots, [\![u_k]\!]$, where $u_j = v^j$
$q \leftarrow \lceil \log \sqrt{k+1} \rceil$
$[\![u_0]\!] \leftarrow (1, 0, 0)$
$[\![u_1]\!] \leftarrow \mathsf{Reshare}([\![v]\!])$
Party P_i sends $[\![u_1]\!]_i$ to party P_{i+1}
for $j = 2$ **to** $2^q - 1$ **do**
  if j *is even* **then**
    Party P_i computes $[\![u_j]\!]_i \leftarrow [\![u_{j/2}]\!]_i^2$ and $[\![u_j]\!]_{i-1} \leftarrow [\![u_{j/2}]\!]_{i-1}^2$
  else
    Party P_i computes
      $[\![t]\!]_i \leftarrow [\![u_1]\!]_i \cdot [\![u_{j-1}]\!]_i + [\![u_1]\!]_i \cdot [\![u_{j-1}]\!]_{i-1} + [\![u_1]\!]_{i-1} \cdot [\![u_{j-1}]\!]_i$
    $[\![u_j]\!] \leftarrow \mathsf{Reshare}([\![t]\!])$
    Party P_i sends $[\![u_j]\!]_i$ to party P_{i+1}
  `// At this point,` P_i `knows` $[\![u_0]\!]_i, \ldots, [\![u_j]\!]_i, [\![u_0]\!]_{i-1}, \ldots, [\![u_j]\!]_{i-1}$
foreach $j \in \{2^q, \ldots, k\}$ **do**
  Let $(r, s) \in \{0, \ldots, 2^q - 1\}$, such that $2^q r + s = j$
  Party P_i computes $[\![t]\!]_i \leftarrow [\![u_r]\!]_i^{2^q} \cdot [\![u_s]\!]_i + [\![u_r]\!]_i^{2^q} \cdot [\![u_s]\!]_{i-1} + [\![u_r]\!]_{i-1}^{2^q} \cdot [\![u_s]\!]_i$
  $[\![u_j]\!] \leftarrow \mathsf{Reshare}([\![t]\!])$

Correctness For $j < 2^q$, the values v^j in the ABB are computed as $v^j = (v^{j/2})^2$ (if j is even) or $v^j = v \cdot v^{j-1}$ (if j is odd). We note that all necessary shares for computing these values are available to the parties. If $j \geq 2^q$ then v^j is computed as $v^j = (v^r)^{2^q} \cdot v^s$, where $2^q r + s = j$. Because char $\mathbb{F} = 2$, the shares of $(v^r)^{2^q}$ are just the shares of v^r, squared q times. This squaring can be performed locally. Again, all shares are available to the parties that need them.

Privacy Similarly to the multiplication protocol, each party knows at most two out of the three shares of $[\![v^j]\!]$, for each j. The last share is a uniformly randomly distributed element of $\mathbb{F}$.

In the second loop of Alg. 2, all $[\![u_j]\!]$ are rerandomized. In the first loop, the values $[\![u_j]\!]$ are not rerandomized for even j. This rerandomization is unnecessary, because of the locality of the computation. We note that all values sent to other parties result from the Reshare protocol.

Complexity The second loop of Alg. 2 has only local computation (recall that Reshare is counted as requiring local computation only). In the first loop, the iterations with odd index j incur the communication of three elements of $\mathbb{F}$, while

the iterations with even j incur no communication. The first loop has at most $\lceil 2\sqrt{k+1} \rceil$ iterations, hence the communication is at most $3\lceil \sqrt{k+1} \rceil$ elements of $\mathbb{F}$.

If we use SHAREMIND's representation of values in ABB, Alg. 2 in place of the offline stage of Alg. 1, and if the DFA description is public, i.e. known to all parties implementing the ABB, then the total offline communication (per character) of executing a m-state DFA on a string over an alphabet of size n is at most $3\lceil \sqrt{mn} \rceil (\lceil \log(m+1) \rceil + \lceil \log(n+1) \rceil) = 3\sqrt{mn} \log(mn) + o(1)$ bits. Here we have assumed that the states of the DFA are encoded as bit-strings $1, \ldots, m$ of length $\lceil \log(m+1) \rceil$, while the characters of the alphabet are encoded as bit-strings $1, \ldots, n$ of length $\lceil \log(n+1) \rceil$. In this way, a suitable γ exists and the elements of $\mathbb{F}$ are encoded as bit-strings of length $\lceil \log(m+1) \rceil + \lceil \log(n+1) \rceil$.

With SHAREMIND's protocols, the online communication (per character of the input string) is 12 elements of $\mathbb{F}$, distributed equally between the multiplication and the retrieve-operation.

6 Improving FA-only / online performance

A different kind of optimization is possible if the ABB implementation is based on Shamir's secret sharing [20] and using the multiplication protocol of Gennaro et al. [12], which is the case for e.g. VIFF [7] or SEPIA [5]. Using this implementation, the computation of a scalar product $[\![\sum_{i=1}^{k} a_i b_i]\!]$ from the values $[\![a_1]\!], \ldots, [\![a_k]\!]$ and $[\![b_1]\!], \ldots [\![b_k]\!]$ stored inside the ABB has the same cost as performing a single multiplication of stored values.

Hence the following modification of the DFA execution algorithm, presented as Algorithm 3 will have the same offline and online complexity as the original algorithm, but perform no costly operations during the FA-only stage.

7 Performance comparison

Private Information Retrieval (PIR) protocols can be adapted to compute $\delta(q, a)$ with asymptotically better communication complexity, if the description of δ is public. A PIR protocol allows the *client* to query for a specific element in *server*'s database, without the server learning the index of that element. In our setting, the ABB would be the client, while the server's computations can be executed in the public. Each character of the input string would need one instance of the PIR protocol to be executed on the transition table δ.

Lipmaa's communication-efficient PIR protocol [17] internally uses the homomorphic cryptosystem by Damgård and Jurik [8]. Its encryption and decryption are usually not considered to be part of the set of operations offered by ABB, but they are often readily available (also in SHAREMIND) using the threshold version of this cryptosystem. The PIR protocol requires the communication of $O(k \log^2(mn))$ bits per query, where mn is the number of elements in the database and k is the size of the RSA-modulus N of the cryptosystem.

Algorithm 3: DFA execution protocol

Data: DFA components $[\![\delta(q,a)]\!]$ and $[\![q_0]\!]$, where $q \in Q$, $a \in \Sigma$.
Data: Characters of the input string $[\![a_1]\!], \ldots, [\![a_\ell]\!]$.
Result: Last state $[\![q_\ell]\!]$ in ABB.

offline processing
foreach $i \in \{1, \ldots, \ell\}$ **do**
  $([\![r_i]\!], [\![r_i^{-1}]\!]) \xleftarrow{\$} \mathbb{F}^*$
  for $j = 2$ **to** $mn - 1$ **do** $[\![r_i^j]\!] \leftarrow [\![r_i]\!] \cdot [\![r_i^{j-1}]\!]$
DFA-only processing
foreach $j \in \{0, \ldots, mn-1\}$ **do** $[\![c_j]\!] \leftarrow \sum_{q \in Q} \sum_{a \in \Sigma} \lambda_{jqa} [\![\delta(q,a)]\!]$
online processing
for $i = 1$ **to** ℓ **do**
  $[\![z_i]\!] \leftarrow (\gamma [\![q_{i-1}]\!] + [\![a_i]\!]) \cdot [\![r_i^{-1}]\!]$
  $z_i \leftarrow \mathsf{retrieve}([\![z_i]\!])$
  foreach $j \in \{0, \ldots, mn-1\}$ **do** $[\![(r_i z_i)^j]\!] = z_i^j [\![r_i^j]\!]$
  $[\![q_i]\!] \leftarrow \sum_{j=0}^{mn-1} [\![c_j]\!] \cdot [\![(r_i z_i)^j]\!]$

To have a valid comparison of the PIR-protocol based DFA execution and our protocols, we have to estimate the constant hidden in the O-notation for the PIR protocol's query complexity, particularly when implemented on top of Sharemind. Per character, the query belongs to the domain of δ, its size is $\alpha = \lceil \log(mn) \rceil$ bits. The single bits of the query must be available, hence we assume that the current pair (q, a) is stored as α separate bits in the ABB. In the PIR protocol, the client encrypts all bits, resulting in α ciphertexts of size $2k$, $3k$, $\ldots$, $(\alpha+1)k$ bits, respectively. In Sharemind's ABB, each of the three parties implementing it may encrypt its share of each bit; these ciphertexts can be combined using the homomorphic properties of the encryption scheme. To minimize the communication, we let two parties send their ciphertexts to the third party that will then perform the operations of the server in the PIR protocol. In this case, the total number of communicated bits for the client's message in the PIR protocol is $2(2k + 3k + \cdots + (\alpha+1)k) = \alpha(\alpha+3)k$.

The third party, performing the operations of the server in the PIR protocol, combines these ciphertexts to a multiply encrypted ciphertext of the query result. This ciphertext must be decrypted using the decryption protocol of the threshold cryptosystem; this causes significant extra communication. To simplify our analysis, let us not estimate the communication costs of these operations, but only compare the total cost of our proposed protocol (per character of the input string) — $\approx 3\sqrt{mn}\log(mn)$ — with the communication costs for producing just the client's message in the PIR protocol — $(\log^2(mn) + 3\log(mn))k$. For acceptable level of security, we have to take $k \geq 1024$. In this case, our protocol has less communication if $mn \leq 10^8$. We are unlikely to have DFA larger than that in real applications.

8 NFA execution

Due to their non-deterministic nature, NFAs are more complicated to handle in a secure manner. We see that even though the NFA execution starts from a single state, after the intermediate steps it can generally be in a subset of states. In order to account for this, we will use characteristic vectors of the intermediate sets $\mathcal{Q}_i = \boldsymbol{\delta}_A(a_1 \cdots a_i)$ to represent them (using the notation of Sec. 3). Let $\mathbf{q}^i = (q_0^i, q_1^i, \ldots, q_{m-1}^i)$ be a binary vector, where $q_j^i = 1$ iff the state $q_j \in \mathcal{Q}_i$.

As $\mathcal{Q}_0 = \{q_0\}$, we have $\mathbf{q}^0 = (1, 0, \ldots, 0)$. Subsequent $\mathbf{q}^i$-s will depend both on the given automaton A and the string s. Namely, in order to determine $\mathbf{q}^i$ from $\mathbf{q}^{i-1}$, δ and a_i, we can compute

$$q_j^i = \bigvee_{q \in \mathcal{Q}_{i-1}} [q_j \in \delta(q, a_i)] = \bigvee_{k=0}^{m-1} q_k^{i-1} \& [q_j \in \delta(q_k, a_i)] \tag{1}$$

for all the components q_j^i of the characteristic vector $\mathbf{q}^i$.

The exact complexity of computing (1) in the ABB depends on which components of the NFA execution problem need to be private. Even if the automaton itself is public and only the string s is private, the characteristic vectors $\mathbf{q}^i$ (for $i > 0$) still need to be protected. However, in order for the term $q_k^{i-1} \& [q_j \in \delta(q_k, a_i)]$ to evaluate to 1, there has to exist a transition from q_k to q_j (even if we do not know whether its label matches a_i or not). Hence, from equation (1) we can leave out all the terms for which there is no such transition. If the NFA has to be private, no such omission is possible.

In order to determine efficiently whether $q_j \in \delta(q_k, a_i)$, we need an efficient representation of δ as well. We will represent it as a look-up table $\overline{\delta} : Q \times Q \to \mathcal{P}(\Sigma)$, where $a_i \in \overline{\delta}(q_k, q_j)$ iff $q_j \in \delta(q_k, a_i)$. To encode subsets of Σ, we will once again use characteristic vectors; let $\mathcal{S} \subseteq \Sigma$ be encoded by vector $\mathbf{s} = (s_1, \ldots, s_n)$ where $s_i = 1$ iff the corresponding $\sigma_i \in \mathcal{S}$. The characteristic vectors in the look-up table $\overline{\delta}$ may be private or public depending on whether A needs to be protected or not. Note that if we have for some q_k and q_j that $\overline{\delta}(q_k, q_j) = \emptyset$, then in the case of public automaton the respective term may be omitted from equation (1).

In order to execute NFA on the (private) string $a_1 \cdots a_\ell$, we also represent the characters of the string using binary characteristic vectors $\mathbf{a}_1, \ldots, \mathbf{a}_\ell$, where $\mathbf{a}_i = (a_i^1, \ldots, a_i^n)$ and $a_i^j = 1$ iff $a_i = \sigma_j$. As a result, the value of the predicate $q_j \in \delta(q_k, a_i)$ can be computed as a dot product $\overline{\delta}(q_k, q_j) \cdot \mathbf{a}_i$. Assuming that addition is free in the underlying secure computation platform (as it is in the case of Sharemind), dot product requires n multiplications that can be performed in parallel.

Next, computing the whole term $q_k^{i-1} \& [q_j \in \delta(q_k, a_i)]$ on equation (1) requires an additional round of multiplication to add the conjunction with q_k^{i-1}. Finally, we need to compute the disjunction over all the states where the transitions to q_i might have come from. This can be accomplished by adding the respective terms, comparing the result to 0 and inverting the comparison result [16]. Working in a suitable field, comparison to 0 may be implemented

using just one round of online multiplications using the protocol by Lipmaa and Toft [18] (though some precomputation is necessary). These operations require that the underlying ring $\mathbb{R}$ of the ABB is a field $\mathbb{F}$ with char $\mathbb{F} \geq m+1$.

The overall procedure of NFA execution is presented as Algorithm 4. The algorithm is obviously private because only ABB operations are used and nothing is declassified. The correctness of the algorithm follows from the discussions above.

Algorithm 4: NFA execution protocol

Data: NFA components $[\![\overline{\delta}(q_k, q_j)]\!]$ for all $q_k, q_j \in Q$, and $[\![\mathbf{q}_0]\!]$.
Data: Characteristic vectors of the characters of the input string $[\![\mathbf{a}_1]\!], \ldots, [\![\mathbf{a}_\ell]\!]$.
Result: Characteristic vector of the last set of achievable states $[\![\mathbf{q}_\ell]\!]$ in ABB.

1 **for** $i = 1$ **to** ℓ **do**
2 **foreach** $j \in \{0, \ldots, m-1\}$ **do**
3 **foreach** $k \in \{0, \ldots, m-1\}$ **do** $[\![t_{ijk}]\!] = [\![q_k^{i-1}]\!] \cdot ([\![\overline{\delta}(q_k, q_j)]\!] \cdot [\![\mathbf{a}_i]\!])$
4 $[\![p_j^i]\!] = \sum_{k=0}^{m-1} [\![t_{ijk}]\!]$
5 $[\![q_j^i]\!] = 1 - [\![p_j^i \stackrel{?}{=} 0]\!]$

Complexity Computing the dot product $[\![\overline{\delta}(q_k, q_j)]\!] \cdot [\![\mathbf{a}_i]\!]$ in line 3 requires n parallel multiplications and the product with $[\![q_k^{i-1}]\!]$ adds an additional multiplication and another round. Altogether, this cycle requires $m(n+1)$ private online multiplications in two rounds.

Summation on line 4 can be performed by the parties without any communication, and the comparisons on 5 require one private online multiplication per comparison, which can all be performed in parallel in one more round. Hence, in order to process each of the input symbols, $m(m(n+1)+1)$ multiplications in three rounds are needed. The overall online complexity of Algorithm 4 is $\ell m(m(n+1)+1)$ multiplications in 3ℓ rounds.

In case of the public automaton, the characteristic vectors $\overline{\delta}(q_k, q_j)$ will be public. Hence the dot product on line 3 will become a local operation performed by the computing parties. As a result, the whole Algorithm 4 requires $\ell m(m+1)$ private multiplications in 2ℓ rounds.

In both cases, the offline complexity consists of the precomputation to facilitate the fast online comparisons. According to [18], the amount of precomputation required for one comparison is $O(\log m)$. Since we need to perform ℓm comparisons, the total work needed in the offline phase is $O(\ell m \log m)$.

9 Conclusions

We have given the first ever algorithm for privacy-preserving NFA execution, as well as fast algorithms for privacy-preserving DFA execution. All our algorithms

are composable and can be easily used as components in the design of larger systems. In case of public FA, our DFA execution algorithms are the fastest for reasonably-sized DFAs. In any case, our DFA execution algorithm has by far the best *online* performance.

References

1. Marina Blanton and Mehrdad Aliasgari. Secure Outsourcing of DNA Searching via Finite Automata. In Sara Foresti and Sushil Jajodia, editors, *DBSec*, volume 6166 of *Lecture Notes in Computer Science*, pages 49–64. Springer, 2010.
2. Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In Sushil Jajodia and Javier López, editors, *ESORICS*, volume 5283 of *Lecture Notes in Computer Science*, pages 192–206. Springer, 2008.
3. Dan Bogdanov, Margus Niitsoo, Tomas Toft, and Jan Willemson. High-performance secure multi-party computation for data mining applications. *Int. J. Inf. Sec.*, 11(6):403–418, 2012.
4. Dan Boneh, Eu-Jin Goh, and Kobbi Nissim. Evaluating 2-DNF Formulas on Ciphertexts. In Joe Kilian, editor, *TCC*, volume 3378 of *Lecture Notes in Computer Science*, pages 325–341. Springer, 2005.
5. Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas Dimitropoulos. SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. In *USENIX Security Symposium*, pages 223–239, Washington, DC, USA, 2010.
6. Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In Shai Halevi and Tal Rabin, editors, *TCC*, volume 3876 of *Lecture Notes in Computer Science*, pages 285–304. Springer, 2006.
7. Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. Asynchronous Multiparty Computation: Theory and Implementation. In Stanislaw Jarecki and Gene Tsudik, editors, *Public Key Cryptography*, volume 5443 of *Lecture Notes in Computer Science*, pages 160–179. Springer, 2009.
8. Ivan Damgård and Mads Jurik. A Generalisation, a Simplification and Some Applications of Paillier's Probabilistic Public-Key System. In Kwangjo Kim, editor, *Public Key Cryptography*, volume 1992 of *Lecture Notes in Computer Science*, pages 119–136. Springer, 2001.
9. Ivan Damgård and Jesper Buus Nielsen. Universally composable efficient multiparty computation from threshold homomorphic encryption. In Dan Boneh, editor, *CRYPTO*, volume 2729 of *Lecture Notes in Computer Science*, pages 247–264. Springer, 2003.
10. Keith B. Frikken. Practical Private DNA String Searching and Matching through Efficient Oblivious Automata Evaluation. In Ehud Gudes and Jaideep Vaidya, editors, *DBSec*, volume 5645 of *Lecture Notes in Computer Science*, pages 81–94. Springer, 2009.
11. Rosario Gennaro, Carmit Hazay, and Jeffrey S. Sorensen. Text search protocols with simulation based security. In Phong Q. Nguyen and David Pointcheval, editors, *Public Key Cryptography*, volume 6056 of *Lecture Notes in Computer Science*, pages 332–350. Springer, 2010.
12. Rosario Gennaro, Michael O. Rabin, and Tal Rabin. Simplified vss and fact-track multiparty computations with applications to threshold cryptography. In *PODC*, pages 101–111, 1998.

13. Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *STOC*, pages 169–178. ACM, 2009.
14. Oded Goldreich, Silvio Micali, and Avi Wigderson. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *STOC*, pages 218–229. ACM, 1987.
15. Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. TASTY: tool for automating secure two-party computations. In *CCS '10: Proceedings of the 17th ACM conference on Computer and communications security*, pages 451–462, New York, NY, USA, 2010. ACM.
16. Sven Laur, Jan Willemson, and Bingsheng Zhang. Round-Efficient Oblivious Database Manipulation. In *Proceedings of the 14th International Conference on Information Security. ISC'11*, pages 262–277, 2011.
17. Helger Lipmaa. An oblivious transfer protocol with log-squared communication. In Jianying Zhou, Javier Lopez, Robert H. Deng, and Feng Bao, editors, *ISC*, volume 3650 of *Lecture Notes in Computer Science*, pages 314–328. Springer, 2005.
18. Helger Lipmaa and Tomas Toft. Secure equality and greater-than tests with sublinear online complexity. In Fedor V. Fomin, Rusins Freivalds, Marta Z. Kwiatkowska, and David Peleg, editors, *ICALP (2)*, volume 7966 of *Lecture Notes in Computer Science*, pages 645–656. Springer, 2013.
19. Payman Mohassel, Salman Niksefat, Seyed Saeed Sadeghian, and Babak Sadeghiyan. An Efficient Protocol for Oblivious DFA Evaluation and Applications. In Orr Dunkelman, editor, *CT-RSA*, volume 7178 of *Lecture Notes in Computer Science*, pages 398–415. Springer, 2012.
20. Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.
21. Ken Thompson. Regular Expression Search Algorithm. *Commun. ACM*, 11(6):419–422, 1968.
22. Tomas Toft. *Primitives and Applications for Multi-party Computation*. PhD thesis, University of Aarhus, Denmark, BRICS, Department of Computer Science, 2007.
23. Juan Ramón Troncoso-Pastoriza, Stefan Katzenbeisser, and Mehmet Utku Celik. Privacy preserving error resilient DNA searching through oblivious automata. In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *ACM Conference on Computer and Communications Security*, pages 519–528. ACM, 2007.
24. Lei Wei and Michael K. Reiter. Third-Party Private DFA Evaluation on Encrypted Files in the Cloud. In Sara Foresti, Moti Yung, and Fabio Martinelli, editors, *ESORICS*, volume 7459 of *Lecture Notes in Computer Science*, pages 523–540. Springer, 2012.
25. Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *FOCS*, pages 160–164. IEEE, 1982.

A Basic protocols of Sharemind

The rerandomization protocol of Sharemind is depicted as Alg. 5 and the multiplication protocol as Alg. 6. We have reordered some steps of the protocols in order to have a grouping more relevant to the other algorithms in this paper. All indices of the parties are *modulo* 3.

Algorithm 5: Resharing protocol $[\![w]\!] \leftarrow \mathsf{Reshare}([\![u]\!])$ in SHAREMIND [3]

Data: Value $[\![u]\!]$.
Result: Value $[\![w]\!]$ such that $w = u$ and the components of $[\![w]\!]$ are independent of everything else.

Party P_i generates $r_i \xleftarrow{\$} \mathbb{R}$, sends it to party P_{i+1}
Party P_i computes $[\![w]\!]_i \leftarrow [\![u]\!]_i + r_i - r_{i-1}$

Algorithm 6: Multiplication protocol in the ABB of SHAREMIND [3]

Data: Values $[\![u]\!]$ and $[\![v]\!]$
Result: Value $[\![w]\!]$, such that $w = uv$

1 $[\![u']\!] \leftarrow \mathsf{Reshare}([\![u]\!])$
2 Party P_i sends $[\![u']\!]_i$ to party P_{i+1}
3 $[\![v']\!] \leftarrow \mathsf{Reshare}([\![v]\!])$
4 Party P_i sends $[\![v']\!]_i$ to party P_{i+1}
5 Party P_i computes $[\![w']\!]_i \leftarrow [\![u']\!]_i \cdot [\![v']\!]_i + [\![u']\!]_i \cdot [\![v']\!]_{i-1} + [\![u']\!]_{i-1} \cdot [\![v']\!]_i$
6 $[\![w]\!] \leftarrow \mathsf{Reshare}([\![w']\!])$