Private Intersection of Regular Languages

Roberto Guanciale
KTH Royal Institute of Technology
Stockholm, Sweden
robertog@csc.kth.se

Dilian Gurov
KTH Royal Institute of Technology
Stockholm, Sweden
dilian@csc.kth.se

Peeter Laud
Cybernetica AS, Tartu, Estonia
Tartu, Estonia
peeter@cyber.ee

***Abstract*—This paper addresses the problem of computing the intersection of regular languages in a privacy-preserving fashion. Private set intersection has been addressed earlier in the literature, but for finite sets only. We discuss the various possibilities for solving the problem efficiently, and argue for an approach based on minimal deterministic finite automata (DFA) as a suitable, non-leaking representation of regular language intersection. We propose two different algorithms for DFA minimization in a secure multiparty computation setting, illustrating different aspects of programming based on universal composability and the constraints this sets on existing algorithms. The implementation of our algorithms is based on the programming language SECREC, executing on the SHAREMIND platform for secure multiparty computation. As one application domain we consider fusion of virtual enterprise business processes.**

I. INTRODUCTION

Formal languages and the related notion of automata have a wide range of applications, including the analysis of structured text, circuit design, pattern matching, analysis of concurrent systems, compression, and DNA computing. While the theories of automata and formal languages are well established, there are few results that take into account *privacy constraints*.

In this work we consider two mutually distrustful parties, each knowing a formal language, wishing to obtain their "combined" language. The two parties are reluctant to reveal any information about their own languages that is not strictly deducible from this combined result. Moreover, we assume that there is no trusted third party. The combined language we consider here is the *intersection* of the two languages. Intersection of formal languages is a primitive used for a wide range of applications. Use cases include: (i) enterprises that are building a cooperation and want to establish the cross-organizational business process, (ii) competitive service providers that collaborate to detect intrusions, and (iii) agencies that intersect their compressed databases.

In some restricted cases, private language intersection can be achieved using existing protocols to compute private set intersection (see e.g. [1]). However, this requires the input languages to respect two constraints: the words of the two languages have to be subsets of a given domain of finite size, and the languages themselves must be finite. In this paper we go beyond finite sets and propose a technique for privacy preserving intersection of regular languages. Our approach is based on: (i) representing the input languages as finite automata, (ii) a well-known result that for every regular language there is a unique (up to isomorphism) DFA accepting it, and (iii) designing privacy preserving versions of some classic algorithms on automata: product, trimming and minimization.

The secure domain in which we implement our solution is the domain of secure multiparty computation (SMC). Frameworks for universal composition of privacy preserving protocols are restrictive as to the set of primitive datatypes and operations they offer. This in turn restricts the algorithms that can be coded efficiently in the framework. The problem here is not to develop a new algorithm for regular language intersection, but instead decompose the problem without compromising the participants' privacy and identify the algorithms that best fit the domain of secure multiparty computation. There are several well-known algorithms for DFA minimization. It turns out that efficient algorithms based on partition refinement like Hopcroft's cannot be readily encoded in standard SMC frameworks. Instead, we had to implement a more abstract version of the protocol, namely Moore's algorithm, which has a worse polynomial asymptotic complexity. As a second minimization algorithm we implemented Brzozowski's algorithm. It has a worst-case exponential complexity, but in many cases behaves better than Moore's algorithm, and is directly encodable using generic constructions for SMC. We use the algorithm also to illustrate how algebraic properties of regular languages can be utilized to reduce the computations that need to be performed in the computationally more expensive secure domain.

To demonstrate the feasibility of private language intersection we implement our proposals in the programming language SECREC executing on the SHAREMIND platform for secure multiparty computation. Finally, we demonstrate how private regular language intersection can be used as a main building block to address a more complex application scenario: the fusion of business processes of virtual enterprises.

The paper is organized as follows. In Section II we recall some well-known notions and results from automata theory. In the next section we develop privacy preserving versions of Moore's and Brzozowski's algorithms, taking into account the restrictions imposed by universal composability, while the following Section IV presents their implementations in SECREC. Section V describes the application of private regular language intersection to the privacy preserving fusion of business processes of virtual enterprises. In the last two sections we describe related work, draw conclusions, and provide directions for future research.

II. AUTOMATA, LANGUAGES AND MINIMIZATION

A. Automata and Languages

We recall several standard notions from the theory of automata and formal languages. For a deeper introduction we refer the reader to standard textbooks such as Kozen [2].

A *deterministic finite automaton* (DFA) is a quintuple $A = (Q, \Sigma, \delta, q_0, F)$, where:

- (i) Q is a finite set of *states*;
- (ii) Σ is a finite set of symbols called *input alphabet*;
- (iii) $\delta : Q \times \Sigma \to Q$ is a *transition function*;
- (iv) $q_0 \in Q$ is the *initial* state;
- (v) $F \subseteq Q$ is a set of *final* (or *accepting*) states.

The transition function is lifted to strings $\sigma \in \Sigma^*$ in the natural fashion; the lifted version is denoted by $\hat{\delta}$. A string σ is *accepted* by the automaton A if $\hat{\delta}(q_0, \sigma) \in F$. The set of strings that A accepts is called the *language* of (or *recognized* by) A, and is denoted by $\mathcal{L}(A)$. The class of languages recognized by DFA is the class of *regular languages*.

Equivalently, regular languages can be represented by *nondeterministic finite automata* (NFA), which only differ from DFA by having a set of start states (rather than exactly one), and having as a co-domain of the transition function the set 2^Q (rather than Q), thus specifying a set of possible transitions from a given state on a given input symbol. Thus, every DFA can also be seen as an NFA. Conversely, every NFA A can be converted to an equivalent DFA (i.e., accepting the same language) by means of the standard *subset* construction; we denote the resulting automaton by $SC(A)$. In general, NFA can be exponentially more succinct in representing regular languages than DFA.

Regular languages are closed under reverse and intersection, among other operations; the standard operations of *reverse* and *product* on finite automata have this effect on their languages. We denote by A^{-1} the reverse automaton of A (obtained simply by reversing the transitions and by swapping initial with final states), and by $A_1 \times A_2$ the product of A_1 and A_2.

B. DFA Minimization

For every regular language there is a unique (up to isomorphism) minimal DFA accepting it (that is, automaton with a minimum number of states). However, there is no canonical minimal NFA for a given regular language. In this subsection we present the two most well-known approaches to DFA minimization. By abuse of notation, we shall use $min(\mathcal{L})$ and $min(A)$ to denote the minimal DFA that recognize $\mathcal{L}$ and $\mathcal{L}(A)$, respectively.

1) Partition Refinement: The standard algorithms for DFA minimization, as e.g. those by Moore [3], Hopcroft [4] and Watson [5], are based on partitioning the states of the DFA into equivalence classes that are not distinguishable by any string, and then constructing a *quotient automaton* w.r.t. the equivalence. Notice that the uniqueness result assumes that the automaton has no unreachable states.

The equivalence $\approx$ is computed iteratively, approximating it with a sequence of equivalences capturing indistinguishability of strings up to a given length. Below we give an account of partition refinement that is suitable for our implementation.

Let $A = (Q, \Sigma, \delta, q_0, F)$ be a DFA. Consider the family of relations $\approx_n \subseteq Q \times Q$ defined as follows:

$$\begin{array}{rcl} \approx_0 & = & F^2 \cup (Q \backslash F)^2 \\ \approx_{i+1} & = & \approx_i \cap \bigcap_{a \in \Sigma} \delta^{-1}(\approx_i, a) \end{array}$$

where the inverse of δ is lifted over pairs of states. The family defines a sequence of equivalences, or *partition refinements*, that in no more than $|Q|$ steps stabilizes in $\approx$.

One standard representation of partitions is as the *kernel* relation of a mapping. Recall that any mapping $f : A \to B$ induces an equivalence relation κ_f on A, called the kernel of f and defined as $(a_1, a_2) \in \kappa_f$ whenever $f(a_1) = f(a_2)$. Applying this representation to partition refinement of the set of states Q of a given DFA, we define the family of index sets:

$$\begin{array}{rcl} I_0 & = & \{0, 1\} \\ I_{i+1} & = & I_i \times [\Sigma \to I_i] \end{array}$$

to be used as co-domains, where $[A \to B]$ denotes the space of (total) mappings from A to B. Next, we define the family of mappings $\rho_n : Q \to I_n$ as follows:

$$\rho_0(q) \;=\; \begin{cases} 0 & \text{if } q \in F \\ 1 & \text{if } q \in Q \backslash F \end{cases}$$

$$\rho_{i+1}(q) \;=\; (\rho_i(q),\ \lambda a \in \Sigma.\ \rho_i(\delta(q, a)))$$

It is easy to show by mathematical induction on n that $\kappa_{\rho_n} = \approx_n$ under the standard notions of equality on pairs and mappings, i.e. that ρ_n represents the n-th approximant in the above partition refinement sequence.

Notice that for representing a partition on Q a codomain of the same cardinality as Q suffices. Therefore one can, at every stage of an actual computation of the approximation sequence, "normalize" the codomain to the set Q itself. For example, let $\chi_{\rho_n} : rng(\rho_n) \to Q$ be a family of injective mappings from the ranges (i.e. active codomains) of ρ_n into Q. We then obtain a family of mappings $\pi_n : Q \to Q$ defined by $\pi_n = \chi_{\rho_n} \circ \rho_n$, which induce the same kernel, i.e. $\kappa_{\pi_n} = \kappa_{\rho_n}$. Now, since $\rho_n = \chi_{\rho_n}^{-1} \circ \pi_n$, one can re-express the above partition refinement sequence via the latter mappings as follows:

$$\begin{array}{rcl} \pi_0(q) & = & \chi_{\rho_0}(\rho_0(q)) \\ \pi_{i+1}(q) & = & \chi_{\rho_{i+1}}(\chi_{\rho_i}^{-1}(\pi_i(q)),\ \lambda a \in \Sigma.\ \chi_{\rho_i}^{-1}(\pi_i(\delta(q, a)))) \end{array}$$

So, for an actual implementation it only remains to define the injections χ_{ρ_n} in a suitable fashion. The approach we adopt in our algorithm (see Section III) is to assume an ordering on the two elements of I_0 and on the symbols of the input alphabet Σ. These orderings induce a (lexicographical) ordering on I_n for all $n > 0$. Now, assuming also an ordering on the states Q, we define χ_{ρ_n} as the least order-preserving injection of $rng(\rho_n)$ into Q, i.e. the injection that maps the least element of $rng(\rho_n)$ to the least element of Q, etc.

2) Brzozowski's algorithm: This algorithm takes a different approach to minimization [6]. Even though its worst-case behavior is exponential, it is conceptually simple and is efficient in many practical cases [7]. Moreover, it does not require the input automaton to be deterministic.

In the following, we shall use $R(A)$ to represent the automaton obtained by trimming A, that is, by removing all states

from which no terminating state is reachable. Brzozowski's algorithm is based on the following property.

Proposition 1 ([6], [7]): Let A be a DFA. Then $SC(R(A^{-1}))$ is a minimal DFA for the language $(\mathcal{L}(A))^{-1}$.

These propositions lead to an intuitive minimization algorithm, which executes determinization and trimming twice:

$$min(A) = SC(R((SC(R(A^{-1})))^{-1}))$$

III. Private regular language intersection

Let two parties P_1 and P_2 hold their respective regular languages $\mathcal{L}_1$ and $\mathcal{L}_2$, both defined over a common and public alphabet Σ. Private regular language intersection allows the two parties to compute the regular language $\mathcal{L} = \mathcal{L}_1 \cap \mathcal{L}_2$ in a privacy-preserving manner, that is, without leaking elements of $\mathcal{L}_1 \setminus \mathcal{L}$ or $\mathcal{L}_2 \setminus \mathcal{L}$.

Here we assume a "semi-honest" adversary model, where every involved agent correctly follows the protocol, but might also record intermediate messages to infer additional information. We use a simplified version of the standard definition of security:

Definition 1 (Privacy w.r.t. semi-honest adversary [8]): A protocol π securely computes the deterministic function f in the presence of semi-honest adversaries if for each party $i \in \{1 \dots n\}$ there exists a polynomial-time simulator S_i such that:

$$\{S_i(x_i, f(x_1 \dots x_n))\}_{x_1 \dots x_n} =^c \{view_i^{\pi}(x_1 \dots x_n)\}_{x_1 \dots x_n}$$

where $=^c$ denotes computationally indistinguishability, x_i is the input of the party i, and $view_i^{\pi}$ is the party's "view" of the protocol, consisting of its inputs, its internal coin tosses, and the ordered sequence of the messages it received.

Regular languages can be infinite; we thus need to choose a finite representation. Here we use finite automata; another obvious choice would have been regular expressions. The two involved parties thus represent their languages $\mathcal{L}_1$ and $\mathcal{L}_2$ by means of two (deterministic or non-deterministic) automata A_1 and A_2 recognizing these; formally, $\mathcal{L}(A_1) = \mathcal{L}_1$ and $\mathcal{L}(A_2) = \mathcal{L}_2$. Definition 1 requires the function f to be deterministic; thus it must yield a canonical representative automaton recognizing the language $\mathcal{L} = \mathcal{L}_1 \cap \mathcal{L}_2$. As such a representative we choose the unique (up to isomorphism) minimal DFA A recognizing $\mathcal{L}$ (see subsection II-B). Finally, in order to enable standard SMC constructions to operate on DFA, we relax the privacy constraint by considering public an upper limit on the number of states of the two input automata. That is, it is publicly known that the languages $\mathcal{L}_1$ and $\mathcal{L}_2$ are in the set of languages accepted by DFA that respect this limit (see [9] for the analysis of the size of this set).

In this section we present two techniques for private regular language intersection that utilize and adapt two principally different algorithms for DFA minimization in a manner that can be implemented in SHAREMIND. We use the second algorithm to also illustrate how certain algebraic properties of regular languages can be exploited to reduce the computations that need to be performed in the computationally more expensive secure domain. In the last subsection we briefly explain why some common strategies for speeding up private computations are not applicable as alternative solutions to the problem addressed here, thus justifying our approach based on minimal DFA.

A. Moore's algorithm

As shown in Section II-B1, automaton $min(\mathcal{L}_1 \cap \mathcal{L}_2)$ can be obtained by using four composable sub-protocols to: (i) compute the product of the DFA A_1 and A_2, (ii) trim the non-reachable states from the resulting automata, (iii) refine the partitions, and (iv) compute the quotient automaton. Here, the most costly step is partition refinement, which requires $|A_1| \cdot |A_2|$ iterations of a sub-protocol that, starting from a partition π_i and the transition function $\delta = \delta(A_1) \times \delta(A_2)$, yields the new partition π_{i+1}.

Since we are not interested in the identity of the states of the involved automata, let us assume that the automata states Q are represented through the natural numbers of the interval $[1 \dots |Q|]$. Each partition π_i, as well as the transition function associated to a symbol (i.e. $\delta_a(q) = \delta(q, a)$), can be represented as mappings of type $Q \to Q$. Thus, each partition refinement step can be implemented as follows:

1) for each state $q \in Q$ and symbol $a \in \Sigma$, compute the mapping composition $x_a(q) = \pi_i(\delta_a(q))$;
2) generate π_{i+1} so that $\pi_{i+1}(q) = \pi_{i+1}(q')$ whenever $(\pi_i(q), \lambda a.x_a(q)) = (\pi_i(q'), \lambda a.x_a(q'))$.

1) Composition of mappings: The mappings representing the symbol transition functions can be represented as *graphs*, or matrices of size $|Q| \times |Q|$ with entries in $\{0, 1\}$. The current partition can be represented as a vector of length $|Q|$. Hence, the mappings x_a can be computed as products of a matrix with a vector, utilizing a protocol for this operation.

Alternatively, certain protocol sets for secure multiparty computation allow the symbol transition functions δ_a to be represented in a manner that preserves their privacy, and allow efficient protocols to compute $\delta_a \circ \pi$ from a mapping π, where both π and $\delta_a \circ \pi$ are represented as vectors of length $|Q|$. This is the case for e.g. the *additive three-party* protocol set used by SHAREMIND [10], [11]. If δ_a is a permutation of Q, then one can use the protocols described in [12]. If δ_a is not a permutation, then it can be represented as $\sigma_a \circ g_{|Q|} \circ \tau_a \circ \iota_{|Q|}$, where ι_n is the identity mapping from $\{1, \dots, n\}$ to $\{1, \dots, \ell_n\}$, g_n is a fixed mapping from $\{1, \dots, \ell_n\}$ to $\{1, \dots, n\}$, and σ_a, τ_a are permutations depending on δ_a. Moreover, $\ell_n = \sum_{k=1}^{n} \lfloor n/k \rfloor = (1 + o(1)) n \ln n$. Hence δ_a can be encoded through the encodings of σ_a and τ_a.

2) Generation of the new partition IDs: The second step of the partition refinement can be achieved by employing any composable protocol that implements the following straightforward algorithm:

1) initialize the new mapping as the identity mapping: $\forall q.\pi_{i+1}(q) = q$;
2) for each state q, update the mapping of every other state q' as: $\pi_{i+1}(q') = \pi_{i+1}(q)$ whenever $\pi_i(q') = \pi_i(q)$ and $\bigwedge_a x_a(q') = x_a(q)$.

The representation of Q through natural numbers proposed above provides us with a natural lexicographical ordering on $(\pi_i(q), \lambda a.x_a(q))$. This allows the second step of the partition

refinement to be achieved by composing the following substeps:

1) generate a matrix M of $|Q| \times (|\Sigma|+3)$ elements, such that $M[q,0] = 0$, $M[q,1] = q$, $M[q,2] = \pi_i(q)$ and $M[q,3+a] = x_a(q)$;
2) sort the matrix lexicographically on the last $|\Sigma|+1$ columns;
3) iterate the matrix and update the 0-th column with a counter; the counter is increased if at least one of the last $|\Sigma|+1$ columns of the current row differs from the corresponding value of the previous row;
4) "invert" the sort, i.e. sort again the matrix on the 1-st column; alternatively, the additive three-party protocol set used by SHAREMIND allows the sorting permutation to be remembered in a privacy-preserving manner and its inverse to be efficiently applied to the rows of the matrix;
5) set $\pi_{i+1}(q) = M[q,0]$.

B. Brzozowski's algorithm

Proposition 1 and the corresponding minimization algorithm provide a naïve strategy to privately intersect regular languages. The strategy requires three main building blocks: (i) a composable algorithm to compute the product of two DFA, (ii) a composable algorithm to trim an NFA, and (iii) a composable algorithm to determinize an NFA. Brzozowski's algorithm involves an intermediate step that deals with a deterministic automaton (not necessarily minimal) recognizing the inverse of the language $\mathcal{L}_1 \cap \mathcal{L}_2$. In the worst case, the number of states of this automaton is $2^{|A_1|\cdot|A_2|}$, thus the algorithm is not practical in settings where only an upper limit on the number of states of the two input automata is public. However, in many cases (see e.g. [7]) this exponential behavior does not occur. In these cases, the adoption of the algorithm is possible if the two participants agree to disclose an upper limit on the number of states of the minimal automata recognizing their reversed languages (i.e. $min((\mathcal{L}_1)^{-1})$ and $min((\mathcal{L}_2)^{-1})$).

The naïve strategy requires the execution of two trims and two determinizations using SMC constructions. We propose two refinements that reduce the number of operations performed privately.

1) Local computation of the reverse languages: Since the reverse operation on finite automata reverses the languages they accept, Proposition 1 guarantees that the minimal automaton $min(\mathcal{L}_1 \cap \mathcal{L}_2)$ can be computed as $SC(R(A^{-1}))$, where A is any DFA whose language is the inverse of $\mathcal{L}_1 \cap \mathcal{L}_2$, that is $\mathcal{L}(A) = (\mathcal{L}_1 \cap \mathcal{L}_2)^{-1}$. Such an automaton can be built as $A = SC(A_1^{-1}) \times SC(A_2^{-1})$. Furthermore, on DFA, reversal distributes over product. These results combine to the following equality:

$$min(\mathcal{L}_1 \cap \mathcal{L}_2) = SC(R(A_1' \times A_2'))$$

where $A_i' = (SC(A_i^{-1}))^{-1}$, allowing for a more efficient algorithm to privately compute language intersection. In fact, each DFA $(SC(A_i^{-1}))^{-1}$ only depends on the input of the corresponding participant, and is thus computable locally. The new algorithm requires only one application of automata product, trim and subset construction to be implemented using standard SMC constructions.

2) Disclosing the reverse language intersection: The computation can be made even more efficient, observing that knowing $\mathcal{L}_1 \cap \mathcal{L}_2$ is equivalent to knowing $(\mathcal{L}_1 \cap \mathcal{L}_2)^{-1}$. In fact, Proposition 1 guarantees that the corresponding minimal automata can be computed from each other:

- $min(\mathcal{L}_1 \cap \mathcal{L}_2) = SC(R(min((\mathcal{L}_1 \cap \mathcal{L}_2)^{-1}))^{-1})$
- $min((\mathcal{L}_1 \cap \mathcal{L}_2)^{-1}) = SC(R(min(\mathcal{L}_1 \cap \mathcal{L}_2))^{-1})$

Moreover, the automaton $min((\mathcal{L}_1 \cap \mathcal{L}_2)^{-1})$ can be computed from the reversed input automata using one application of product, trim and subset construction only:

$$\begin{aligned} min((\mathcal{L}_1 \cap \mathcal{L}_2)^{-1}) &= SC(R((A_1 \times A_2)^{-1})) \\ &= SC(R(A_1^{-1} \times A_2^{-1})) \end{aligned}$$

The resulting automaton can be safely disclosed, allowing the two participants to locally reverse the language.

C. Discussion: Alternative approaches

To justify further our approach based on minimal DFA, we discuss below briefly why some common strategies for speeding up the private computation of a function f are not applicable as alternative solutions to the problem addressed here (i.e., f yielding the intersection of two regular languages).

a) Randomization of the result: If $f = g \circ f'$, a common approach is to first privately compute f', and then privately and randomly construct a public result that is $g-$equivalent to the output of f' so that the distribution of the result is uniform. Any automaton A recognizing the intersection language $\mathcal{L}$ is equivalent to the automaton $A_1 \times A_2$. This observation leads to a straightforward strategy for private regular language intersection: (i) use a composable protocol to compute $f' = A_1 \times A_2$, and then (ii) build an "efficient" composable protocol that randomly constructs an equivalent automaton (to $A_1 \times A_2$). However, there is no efficient way to generate a uniform distribution over all automata equivalent to a given one. Instead, what essentially is achieved here, we construct an equivalent automaton with a degenerate distribution, namely the minimal deterministic one.

b) Problem transformation: Another common approach is to generate a new "random" problem f' by transforming f in a way that (i) there is no relation between the random distribution of f' and the original problem, and (ii) it is possible to compute the solution of f knowing the solution of f' and the applied transformation. This allows (i) to generate and apply the "random" transformation in a privacy preserving way, (ii) to offload the computation of f' to a non-trusted third party, and (iii) to compute the expected output from the solution of f' and the generated transformation in a privacy preserving way. This approach has been successfully applied to scientific computations (see e.g. [13]), where the inputs are represented as matrices and the transformation can be easily generated as random invertible matrices. Again, however, there is no mechanism to generate a "random and invertible" transformation such that, starting from an arbitrary input automaton, the distribution of the resulting transformed automaton is uniform.

c) Generation of a combinatorial problem: This approach has been successfully used for scientific computations (see e.g. [14]). The idea is as follows: (a) one of the two parties decomposes the problem in a set of n sub-tasks, such that knowing the solution of each sub task allows the party to discover only the intended private output, (b) the same party generates $m-1$ other (random) problems and the corresponding n sub-tasks, (c) the other party locally solves $m*n$ sub-tasks, and finally, (d) the two parties execute an n-out-of-$n*m$ oblivious transfer, allowing the first party to compute the intended private output. The security of the approach depends on the fact that if the $m*n$ sub-tasks are "indistinguishable", then the attacker needs to solve a combinatorial problem $\begin{pmatrix} m*n \\ n \end{pmatrix}$ to discover the original inputs. In the context of private language intersection this would amount to performing the following steps: (i) the first party generates $A_1^{1,1} \dots A_1^{1,n}$ such that $\mathcal{L}_1 = \cup_i \mathcal{L}(A_1^{1,i})$, (ii) the first party generates $\mathcal{L}_1^2 \dots \mathcal{L}_1^m$ and the automata $A_1^{k,n}$ so that $\forall k. \mathcal{L}_1^k = \cup_i \mathcal{L}(A_1^{k,i})$, (iii) the second party solves the problems $\mathcal{L}^{k,i} = \mathcal{L}(A_1^{k,i}) \cap \mathcal{L}_2$, (iv) the two parties use an OT-transfer to allow the first party to obtain the solutions $\mathcal{L}^{1,1} \dots \mathcal{L}^{1,n}$, and finally, (v) the first party computes the final result $\mathcal{L} = \cup_i \mathcal{L}^{1,i}$.

Again, the main obstacle to applying this approach here is that there is no mechanism to obtain a uniform distribution of random regular languages (or automata having an arbitrary number of states). The existing random generators [15] that pick one automaton from the set of automata having up to a fixed number of states cannot be used here. In fact, this prevents to generate $m*n$ indistinguishable sub-tasks (since the constrains (i) and (ii) must be satisfied), thus allowing the attacker to avoid to solve the combinatorial problem.

d) Incremental construction: Yet another common approach is to compute the result of a function f by composing simpler functionalities that incrementally approximate the result, so that each intermediate result is "part" of the final output. For example, starting from an initial empty approximation $A_0 = \emptyset$, the union of two finite sets P_1 and P_2 can be incrementally built by iterating a protocol that yields the minimum of two elements: $A_i = A_{i-1} \cup min(min(P_1 \setminus A_i), min(P_2 \setminus A_i))$. Such an approach has been used to privately compute all pair shortest distances [16] and minimum spanning trees. However, we did not find in the literature such a construction to compute the minimal automaton A starting from A_1 and A_2 (with the exception of the dynamic minimization algorithms that can be used only if one of the two input automata recognizes a finite language).

IV. Implementation

We have implemented privacy-preserving language intersection for both approaches presented in Section III.

A. Moore's algorithm

The implementation of Moore's algorithm (see Section III-A) consists of the following three steps: (i) a product construction, (ii) determining the reachable states, and (iii) determining the equivalent states. The steps (ii) and (iii) are independent of each other, as we do not want to leak the size of the automaton resulting from one of these steps. We have used the SHAREMIND platform and its three-party additively secret shared protection domain for secure multiparty computations, offering efficient vectorized arithmetic operations, as well as array permutations and rearrangements. In this set-up, there are three *computing nodes*, into which the i-th *client party* holding $A_i = (Q_i, \Sigma, \delta_i, q_{0i}, F_i)$ uploads its automaton in shared form [17]. The computing nodes find a shared representation of the minimization of $A = A_1 \times A_2$ and reveal it to the clients. The platform is secure against a semi-honest adversary that can adaptively corrupt one of the computing nodes.

In our implementation, the size of the task — the values $|Q_1|$, $|Q_2|$ and $|\Sigma|$ — is public. Also, q_{01} and q_{02} are public, this is w.l.o.g., as each party can permute its Q_i. The set F_i is represented as a bit-vector $\boldsymbol{\chi}_i$ of length $|Q_i|$; this vector is uploaded to computing nodes in the shared form $[\![\boldsymbol{\chi}_i]\!] = ([\![\boldsymbol{\chi}_i]\!]_1, [\![\boldsymbol{\chi}_i]\!]_2, [\![\boldsymbol{\chi}_i]\!]_3)$ with $[\![\boldsymbol{\chi}_i]\!]_1 \oplus [\![\boldsymbol{\chi}_i]\!]_2 \oplus [\![\boldsymbol{\chi}_i]\!]_3 = \boldsymbol{\chi}_i$. The symbol transition functions $\delta_{i,a} = \delta_i(\cdot, a) : Q_i \to Q_i$ are represented as discussed in Sec. III-A1: shared mappings $[\![\delta_{i,a}]\!]$, allowing $\mathsf{rearr}([\![\delta_{i,a}]\!], [\![\vec{x}]\!]) = ([\![x_{\delta_{i,a}(1)}]\!], \dots, [\![x_{\delta_{i,a}(|Q_i|)}]\!])$ to be computed efficiently from $[\![\vec{x}]\!] = ([\![x_1]\!], \dots, [\![x_{|Q_i|}]\!])$.

Step (i) of our intersection algorithm is essentially a no-op, as computing $F = F_1 \times F_2$ is trivial and there is no efficient way to compute the products $\delta_a = \delta_{1,a} \times \delta_{2,a}$. Instead, to compute $\mathsf{rearr}([\![\delta_a]\!], [\![\vec{x}]\!])$ in steps (ii) and (iii) for $\vec{x}$ indexed with elements of $Q = Q_1 \times Q_2$, we organize $[\![\vec{x}]\!]$ as an array with $|Q_1|$ rows and $|Q_2|$ columns. We will then apply $\mathsf{rearr}([\![\delta_{1,a}]\!], \cdot)$ to the rows of $\vec{x}$, and $\mathsf{rearr}([\![\delta_{2,a}]\!], \cdot)$ to the columns of the result.

The implementation of Moore's algorithm in step (iii) of our intersection algorithm is iterative, following Sec. III-A2. One iteration, constructing the shared partition $[\![\pi_{i+1}]\!]$ from $[\![\pi_i]\!]$ is given in Algorithm 1. Here $[\![\vec{\iota}]\!]$ is a vector of length $|Q|$, with $\iota_i = i$. All vectors are considered to be column vectors. All **foreach**-loops are parallelized.

Algorithm 1: One iteration of Moore's algorithm

```
foreach a ∈ Σ do [[π_{i,a}]] := rearr([[δ_a]], [[π_i]])
[[Π̂_i]] := ([[π]] | [[π_{i,a_1}]] | ··· | [[π_{i,a_{|Σ|}}]] | [[ι]])
([[Π_i]], [[σ]]) := sort_rows([[Π̂_i]])
[[c_1]] := 1
foreach j ∈ {2, ..., |Q|} do
    foreach a ∈ Σ do
        [[c_{j,a}]] := ([[Π_i[j, a]]] ≠ [[Π_i[j − 1, a]]])
    [[c_j]] := ([[Π_i[j, 0]]] ≠ [[Π_i[j − 1, 0]]]) ∨ ⋁_{a∈Σ} [[c_{j,a}]]
foreach j ∈ |Q| do [[π̂_{i+1}[j]]] := Σ_{k=1}^{j} [[c_k]]
[[π_{i+1}]] := unshuffle([[σ]], [[π̂_{i+1}]])
```

This algorithm uses the following functionality from the SHAREMIND platform. The function sort_rows returns both its arguments (rows sorted lexicographically), and a shared permutation $[\![\sigma]\!]$ that brought the rows of the original matrix to the sorted order. The function is not completely privacy-preserving — it leaks, which rows were equal to which other ones. To prevent this leakage, we have used the extra column ι. Boolean values $\{\mathsf{true}, \mathsf{false}\}$ are identified with integers $\{1, 0\}$. The large fan-in disjunction is implemented by first adding up

the inputs and then comparing the result to 0 [12]. The function unshuffle applies the inverse of σ to the vector $\hat{\pi}_{i+1}$, thereby undoing the permutation from sorting.

The iterations have to be run until the number of parts in π remains the same. Making the results of sameness checks public leaks the number of iterations. This number can be derived from the minimized automaton, hence the leak is acceptable, if the final automaton is public. Otherwise an application-specific bound should be provided by the parties holding A_1, A_2, as the worst-case bound is almost $|Q| = |Q_1| \cdot |Q_2|$.

In step (ii), reachable states can be computed in various ways. One can find the reflexive-transitive closure of the adjacency matrix M of A. This requires $\log D$ multiplications of matrices of size $|Q|\times|Q|$, where $D \leq |Q|$ is an upper bound on the distance of the states of $A_1 \times A_2$ from the starting state. Using SHAREMIND, one multiplication of $n \times n$ matrices requires $O(n^3)$ local work, but only $O(n^2)$ communication. Still, for larger values of n, this is too much. Instead, we find the reachable states iteratively: let $R_0 = \{q_0\}$ and $R_{i+1} = R_i \cup R'_i$, where $R'_i = \{q \in Q \mid \exists q' \in R_i, a \in \Sigma : \delta(q', a) = q\}$ (represented as 0/1-vectors $\vec{r}_i$). If the diameter of A is small, and we have a good (application-specific) upper bound D' for it, then this approach may require much less computational effort.

The vector $\vec{r}'_i$ can be found from $\vec{r}_i$ by multiplying it with the the matrix M. Using SHAREMIND, this requires $O(n^2)$ communication and $O(n^2 D')$ local computation due to the size of M for the computation of $\vec{r}_{D'}$ from $\vec{r}_0$. With rearrangements, we can bring both costs down to $O(nD')$.

When computing $\vec{r}'_i$ from $\vec{r}_i$, we have to apply $[\![\delta_a]\!]$ to $\vec{r}_i$ "in the opposite direction", compared to step (iii): $r'_{ij} = 1$ iff $r_{ik} = 1$ and $\delta(q_k, a) = q_j$ for some k and a. SHAREMIND provides the function $[\![\vec{y}]\!] = \mathsf{rearr}^{-1}([\![f]\!], [\![\vec{x}]\!])$, satisfying $y_i = \sum_{j \in f^{-1}(i)} x_j$. This function, with performance equal to rearr, suffices for finding reachable states; Algorithm 2 gives the computation of $[\![\vec{r}_{i+1}]\!]$ from $[\![\vec{r}_i]\!]$.

Algorithm 2: One iteration in finding reachable states

foreach $a \in \Sigma$ **do** $[\![\vec{s}_a]\!] := \mathsf{rearr}^{-1}([\![\delta_a]\!], [\![\vec{r}_i]\!])$
foreach $j \in \{1, \ldots, |Q|\}$ **do**
  $[\![s[j]]\!] := \sum_{a \in \Sigma} [\![s_a[j]]\!]$
  $[\![r_{i+1}[j]]\!] := ([\![s[j]]\!] \neq 0) \vee [\![r_i[j]]\!]$

Our SHAREMIND cluster consists of three computers with 48 GB of RAM and a 12-core 3 GHz CPU with Hyper Threading running Linux (kernel v.3.2.0-3-amd64), connected by an Ethernet local area network with link speed of 1 Gbps. On this cluster, we have benchmarked the execution time of Alg. 1 and 2. If $|\Sigma| = 10$, $|Q_1| = |Q_2| = 100$, then one iteration in determining reachable states (Alg. 2) requires ca. 0.9 s, while one iteration in Moore's algorithm (Alg. 1) requires ca. 4.5 s. For $|\Sigma| = 10$, $|Q_1| = |Q_2| = 300$, these times are 6.2 s and 40 s, respectively. In the worst case, algorithms to converge in $|Q_1| \cdot |Q_2|$ iterations. While these are the execution times of single iterations, our experiments show that privacy-preserving minimization of DFA is feasible even for automata with 100,000 states, if the application producing these automata allows us to give reasonable bounds on the number of iterations necessary for these algorithms to converge.

B. Brzozowski's algorithm

The implementation of Brzozowski's algorithm (see Section III-B) consists of the following three steps: (i) a product construction, (ii) determining the reachable states (as above), and (iii) determinization via the subset construction. As usual, there are three *computing nodes* that find a shared representation of the minimization of $A_1 \times A_2$ (having $|Q| = |Q_1| \cdot |Q_2|$ states) and reveal it to the clients.

The main part of the implementation of the subset construction in step (iii) is iterative. Given an NFA $(Q, \Sigma, \delta, Q_0, F)$, where each δ_a is represented as a $|Q| \times |Q|$ boolean matrix satisfying $\delta_a[i, j] \Leftrightarrow q_j \in \delta(q_i, a)$, one iteration of our algorithm constructs the set $\delta'_a(q'_i)$ for $a \in \Sigma$ and the state $q'_i \in Q'$ of the output DFA $(Q', \Sigma, \delta', q'_0, F')$.

Internally, Algorithm 3 uses a boolean matrix ss of size $max \times |Q|$, where max is an application-dependent upper bound on the size of Q' that tracks the correspondence between the elements of Q' and the subsets of Q (i.e. $q'_j \in Q'$ corresponds to $\bar{Q} \subseteq Q$ if $\forall k. ss[j, k] = (q_k \in \bar{Q})$). Also, the boolean vector σ (of size max) is used to record which elements of Q' are already "active". Initially, ss and σ equal false at all positions. Let $b \,?\, b_1 : b_2$ denote $(b \wedge b_1) \vee (\neg b \wedge b_2)$.

Algorithm 3: The subset construction

$[\![ss[1]]\!] := [\![Q_0]\!]$; $[\![\sigma[1]]\!] :=$ true
for $i := 1$ **to** max **do**
  foreach $a \in \Sigma$ **do** `// sequentially`
    foreach $j \in \{1, \ldots, |Q|\}$ **do**
      $[\![s'[j]]\!] := \bigvee_{k \in 1 \ldots |Q|} ([\![ss[i, k]]\!] \wedge [\![\delta_a[k, j]]\!])$
    $[\![f]\!] :=$ false
    for $j = 1$ **to** max **do**
      $[\![ss[j]]\!] := [\![f]\!] \vee [\![\sigma[j]]\!] \;?\; [\![ss[j]]\!] : [\![s']\!]$
      $[\![c]\!] := \neg [\![f]\!] \wedge ([\![ss[j]]\!] = [\![s']\!])$
      $[\![\delta'_a[i, j]]\!] := [\![c]\!]$
      $[\![f]\!] := [\![f]\!] \vee [\![c]\!]$
      $[\![\sigma[j]]\!] := [\![\sigma[j]]\!] \vee [\![c]\!]$

In one iteration (given by fixed i and a), Algorithm 3 performs two steps. In the first loop it computes the set s' (represented as a boolean vector) of states of the NFA that can be reached by consuming the symbol a from all states corresponding to the i-th state of the DFA. In the second loop it searches for a row of ss that equals s'. In case there is no such row, it is added to ss. During the execution of the second loop, the variable f indicates whether such row has already been found (or added). The variable c shows whether this row is found in the current iteration of the second loop. Assuming that the number of states in the DFA will not supercede max, there is exactly one iteration where c will be true.

Using SHAREMIND, the first step of one iteration of Algorithm 3 requires $O(|Q|^3)$ local work and $O(|Q|^2)$ com-

munications. The second step requires $O(max \cdot |Q|)$ local work and communications.

The algorithm and its data structures depend on the constant max, which limits the number of states of the resulting DFA and bounds the number of required iterations of the subset construction. Even if in the worst case $max = 2^{|Q|}$, in many cases (see e.g. [7]) this exponential behavior does not occur and the two participants can agree to disclose this upper limit.

V. Example Application: Fusion of Virtual Enterprise Business Processes

A virtual enterprise (VE) is a temporary alliance of businesses whose cooperation is supported by computer networks. VEs can be part of long-term strategic alliances or short-term collaborations built to catch business opportunities. To effectively manage the VE it is necessary to support the establishment of the cross-organizational business process, that is to capture the possible executions of the involved parties that are compliant with the business processes of VE constituents. We refer to this problem as *VE process fusion*. Observe that the problem here is not so much the one of computing a global representation of the overall business process, but rather the one of computing *local views* of the fusion. In other words, the problem is, for each partner, to compute what can or is to be performed locally, i.e., the contributing subset of the existing local business process.

One of the main barriers to VE process fusion is the participants' autonomy. In particular, the participants can be reluctant to expose their internal processes, since this knowledge can be analyzed to reveal sensitive information such as efficiency secrets, or weaknesses in responding to a market demand. In this section we demonstrate how private regular language intersection can be applied to support *private VE process fusion*.

A. Formalization of VE Process Fusion

Two widely adopted industry standards to specify enterprise business processes are BPMN [18] and EPC [19]. There is a general agreement (see [20]) that well-formed business processes correspond to sound Workflow Nets (a subclass of Petri Nets), and several tools have been developed to translate diagrams (either BPMN or EPC) to the corresponding formal Workflow Nets (e.g. [21]), thus enabling formal analysis techniques. Since the class of languages of Workflow Nets is a subset of the class of regular languages [22], we can assume for the purposes of the present study that business processes of the enterprises can be expressed as regular languages.

Assume two enterprises, with their own business processes, that cooperate to build a VE. For each of the two enterprises we are given a local alphabet, Σ_1 respectively Σ_2. The letters of an alphabet can represent various types of actions or events: (i) an internal task of the enterprise (e.g. packaging of goods), (ii) an interaction between the two enterprises (e.g. exchange of electronic documents), (iii) an event observed by one of the enterprises only (e.g. the receipt of a payment), or (iv) an event observed by both enterprises (e.g. that a carrier has left the harbor). Each enterprise also owns a local business process, representing all licit possible executions, that is given as a regular language, $\mathcal{L}_1 \subseteq \Sigma_1^*$ respectively $\mathcal{L}_2 \subseteq \Sigma_2^*$, by means of a suitable finite representation.

Our formalization is based on the notions of language projection and product. Let $\mathcal{L}$ be a language over alphabet Σ. Then $\mathbf{proj}_{\Sigma'}(\mathcal{L})$ for $\Sigma' \subseteq \Sigma$ denotes the *projection* of $\mathcal{L}$ onto the alphabet Σ', defined as expected through deleting letters not in Σ', and $\mathbf{proj}^{-1}_{\Sigma''}(\mathcal{L})$ for $\Sigma \subseteq \Sigma''$ denotes the *inverse projection* of $\mathcal{L}$ onto the alphabet Σ'', defined as the greatest language on Σ' such that its projection onto Σ is $\mathcal{L}$.

The *product* of two languages $\mathcal{L}_1$ and $\mathcal{L}_2$ over alphabets Σ_1 and Σ_2, denoted $\mathcal{L}_1 \times^L \mathcal{L}_2$, is the largest language $\mathcal{L}$ over $\Sigma_1 \cup \Sigma_2$ such that $\mathbf{proj}_{\Sigma_1}(\mathcal{L}) = \mathcal{L}_1$ and $\mathbf{proj}_{\Sigma_2}(\mathcal{L}) = \mathcal{L}_2$:

$$\mathcal{L}_1 \times^L \mathcal{L}_2 \;=\; \mathbf{proj}^{-1}_{\Sigma_1 \cup \Sigma_2}(\mathcal{L}_1) \cap \mathbf{proj}^{-1}_{\Sigma_1 \cup \Sigma_2}(\mathcal{L}_2)$$

In terms of finite automata, this corresponds to a weakly synchronous product. In our setting, the global VE business process is represented by the language product: it yields all global processes that are consistent with the participants' constraints.

The problem of *VE process fusion* can thus be defined as computing the mapping:

$$\mathcal{L}_i \mapsto \mathbf{proj}_{\Sigma_i}(\mathcal{L}_1 \times^L \mathcal{L}_2) \qquad (i \in \{1, 2\})$$

that is, each participant computing, from its local business process $\mathcal{L}_i$, the subset of local processes that are consistent with the global VE business process.

Privacy preservation in this context means the two participants to obtain $\mathbf{proj}_{\Sigma_1}(\mathcal{L}_1 \times^L \mathcal{L}_2)$ and $\mathbf{proj}_{\Sigma_2}(\mathcal{L}_1 \times^L \mathcal{L}_2)$, respectively, without being able to learn about the other enterprise's language more than what can be deduced from the own language (i.e., the *private input*) and the obtained result (i.e., the *private output*). Apart from the languages, we also consider private the alphabet differences, that is, we consider public just the common alphabet $\Sigma_1 \cap \Sigma_2$ (i.e. the events of type ii and iv).

B. A Protocol for Private VE Process Fusion

As usual, we first present an ideal protocol for private VE process fusion, and then present a real protocol, the privacy of which is shown relative to the ideal one. The *ideal protocol* for private VE process fusion obtains the private inputs of the enterprises (through perfect channels), computes the private outputs, and sends the latter to the enterprises (again via perfect channels).

To obtain a real protocol for the task, consider the language:

$$\mathcal{L} \;=\; \mathbf{proj}_{\Sigma_1 \cap \Sigma_2}(\mathcal{L}_1) \cap \mathbf{proj}_{\Sigma_1 \cap \Sigma_2}(\mathcal{L}_2)$$

The two public outputs can be computed locally by the respective participant from this language and the respective private input, as the following result shows.

Proposition 2: Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be two languages over alphabets Σ_1 and Σ_2, respectively, and let $\mathcal{L}$ be as defined above. Then the following holds:

$$\mathbf{proj}_{\Sigma_i}(\mathcal{L}_1 \times^L \mathcal{L}_2) = \mathcal{L}_i \cap \mathbf{proj}^{-1}_{\Sigma_i}(\mathcal{L}) \qquad (i \in \{1, 2\})$$

Using as a building block on of the protocols for *private regular language intersection* presented in Section III, we

propose the following *real protocol*, in which every participant $i \in \{1, 2\}$ performs the following steps:

1) Compute locally, from the private input $\mathcal{L}_i$, the language $\mathbf{proj}_{\Sigma_1 \cap \Sigma_2}(\mathcal{L}_i)$.
2) Send this language to the protocol for private language intersection.
3) Receive from the protocol the language $\mathcal{L}$ defined above.
4) Compute $\mathcal{L}_i \cap \mathbf{proj}_{L_i}^{-1}(\mathcal{L})$, which by Proposition 2 is equal to the private output.

Privacy is established relative to the ideal protocol, by exhibiting a *simulator* that computes the received messages of every participant from their private inputs and outputs. We accomplish this by proving that the language $\mathcal{L}$ can be deduced by each participant locally from the respective private input and private output.

Proposition 3: Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be two languages over alphabets Σ_1 and Σ_2, respectively, and let $\mathcal{L}$ be as defined above. Then the following holds:

$$\mathcal{L} = \mathbf{proj}_{\Sigma_1 \cap \Sigma_2}(\mathbf{proj}_{\Sigma_i}(\mathcal{L}_1 \times^L \mathcal{L}_2)) \quad (i \in \{1, 2\})$$

VI. Related work

The theories of finite automata and regular languages are well established. In particular, regular language intersection (i.e. fully synchronous composition) has been used to model the behavior of concurrent agents, validate circuit designs, intersect regular expressions and intersect compressed representations of large databases of strings.

In the literature, few results take into account privacy constraints. For regular languages that are finite and whose words are selected from a finite domain, protocols to compute private set intersection (see e.g. [1]) can be adopted to solve private language intersection. However, practical application of these techniques requires that the word domain is not only finite, but also reasonably small. Clearly, these constraints cannot be satisfied if the input languages represent business processes, since presence of loops directly induces the word domain to be infinite.

If at least one of the two input languages is finite, then privacy preserving pattern matching (i.e. *oblivious DFA execution*, see e.g. [23], [24], [25]) can be used to intersect the languages. In this scenario, the party owning the infinite (or large) language encodes its input as a DFA. Then, assuming as public an upper limit on the size of the finite language, the two parties iteratively repeat the privacy preserving pattern matching, allowing the second party to discover the subset of its own language that matches the language intersection.

The usefulness of automatic systems to support virtual enterprises and dynamic B2B has been widely recognized. Several works investigate algorithms that can automate process integration in the context of Web Services. For example, in [26] the authors use a non-emptiness test on the intersection of DFA as the main machinery to allow an enterprise to dynamically search partners that match the required business process. These results assume that the participants agree to publish their own business process in a publicly accessible directory.

Our formalization of the VE process fusion problem follows the approach described in [27] for *modular distributed monitoring* using formal languages. The main difference between the two applications is that VE process fusion requires to compute what "will" be allowed, while monitoring requires to identify what "happened". For this reason the data structures that support the abstract formalization differ: for process fusion we use DFA, while for monitoring the authors exploit trellis.

VII. Conclusion

In this paper we presented an approach to private intersection of regular languages. Since such languages can be infinite, the present work goes beyond the existing techniques for private set intersection, which can only handle finite sets. And as we argue here, none of the existing general techniques for computing privately a function is readily applicable to the present task.

Our approach uses finite representations of regular languages as DFA, and is based on computing, in a secure domain, the product automaton of the two input automata, and then its minimized form. Since minimal DFA are unique (up to isomorphism) for a given regular language, the result of the computation can be revealed to the parties without leaking any additional information than the intersection language itself. The same approach can be used for any binary operation on regular languages that has its counterpart in the class of finite automata: one needs merely to replace the product construction with the corresponding one, and then proceed with minimization as above.

The main application of private regular language intersection we discuss here is taken from the domain of privacy preserving fusion of virtual enterprise business processes. While other application areas are easy to identify, the presented one is particularly meaningful because it is less sensitive to time efficiency than application areas such as private modular distributed monitoring.

In future work we will investigate the incremental construction of the intersection language suggested in Section III-C, other application domains, as well as wider language classes such as the context-free languages. We also plan to integrate our prototype with tools for business process analysis [21].

Acknowledgment

We are grateful to Jaak Ristioja for help with implementation. This work has been supported by the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 284731 "Usable and Efficient Secure Multiparty Computation (UaESMC)". It was also supported by Estonian Research Council through grant no. IUT27-1, and by ERDF through Centre of Excellence in Computer Science (EXCS).

References

[1] M. J. Freedman, K. Nissim, and B. Pinkas, "Efficient private matching and set intersection," in *Advances in Cryptology-EUROCRYPT 2004*. Springer, 2004, pp. 1–19.

[2] D. C. Kozen, *Automata and computability*. Springer-Verlag New York, Inc., 1997.

[3] E. F. Moore, "Gedanken-experiments on sequential machines," *Automata studies*, vol. 34, pp. 129–153, 1956.

[4] J. Hopcroft, "An n log n algorithm for minimizing states in a finite automaton," DTIC Document, Tech. Rep., 1971.

[5] B. W. Watson, "A taxonomy of finite automata minimization algorithms," 1993.

[6] J. A. Brzozowski, "Canonical regular expressions and minimal state graphs for definite events," *Mathematical theory of Automata*, vol. 12, no. 6, pp. 529–561, 1962.

[7] J. Berstel, L. Boasson, O. Carton, and I. Fagnot, "Minimization of automata," *arXiv preprint arXiv:1010.5318*, 2010.

[8] O. Goldreich, "Secure multi-party computation," *Manuscript. Preliminary version*, 1998.

[9] M. Domaratzki, D. Kisman, and J. Shallit, "On the number of distinct languages accepted by finite automata with n states," *Journal of Automata, Languages and Combinatorics*, vol. 7, no. 4, pp. 469–486, 2002.

[10] D. Bogdanov, S. Laur, and J. Willemson, "Sharemind: A framework for fast privacy-preserving computations," in *ESORICS*, ser. Lecture Notes in Computer Science, S. Jajodia and J. López, Eds., vol. 5283. Springer, 2008, pp. 192–206.

[11] D. Bogdanov, "Sharemind: programmable secure computations with practical applications," Ph.D. dissertation, University of Tartu, February 2013.

[12] S. Laur, J. Willemson, and B. Zhang, "Round-Efficient Oblivious Database Manipulation," in *Proceedings of the 14th International Conference on Information Security. ISC'11*, 2011, pp. 262–277.

[13] W. Du and M. J. Atallah, "Privacy-preserving cooperative scientific computations," in *Computer Security Foundations Workshop, IEEE*. IEEE Computer Society, 2001, pp. 0273–0273.

[14] M. J. Atallah and K. B. Frikken, "Securely outsourcing linear algebra computations," in *ASIACCS*, D. Feng, D. A. Basin, and P. Liu, Eds. ACM, 2010, pp. 48–59.

[15] F. Bassino and C. Nicaud, "Enumeration and random generation of accessible automata," *Theoretical Computer Science*, vol. 381, no. 1, pp. 86–104, 2007.

[16] J. Brickell and V. Shmatikov, "Privacy-preserving graph algorithms in the semi-honest model," in *Advances in Cryptology-ASIACRYPT 2005*. Springer, 2005, pp. 236–252.

[17] R. Talviste, "Deploying secure multiparty computation for joint data analysis - a case study," Master's thesis, University of Tartu, 2011.

[18] S. A. White, "Introduction to bpmn," *IBM Cooperation*, vol. 2, no. 0, p. 0, 2004.

[19] A.-W. Scheer and M. Nüttgens, *ARIS architecture and reference models for business process management*. Springer, 2000.

[20] W. M. van der Aalst, "The application of petri nets to workflow management," *Journal of circuits, systems, and computers*, vol. 8, no. 01, pp. 21–66, 1998.

[21] B. F. van Dongen, A. K. A. de Medeiros, H. Verbeek, A. Weijters, and W. M. Van Der Aalst, "The prom framework: A new era in process mining tool support," in *Applications and Theory of Petri Nets 2005*. Springer, 2005, pp. 444–454.

[22] W. M. Van der Aalst, "Verification of workflow nets," in *Application and Theory of Petri Nets 1997*. Springer, 1997, pp. 407–426.

[23] J. R. Troncoso-Pastoriza, S. Katzenbeisser, and M. Celik, "Privacy preserving error resilient dna searching through oblivious automata," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 519–528.

[24] M. Blanton and M. Aliasgari, "Secure outsourcing of dna searching via finite automata," in *Data and Applications Security and Privacy XXIV*. Springer, 2010, pp. 49–64.

[25] P. Laud and J. Willemson, "Universally composable privacy preserving finite automata execution with low online and offline complexity," Cryptology ePrint Archive, Report 2013/678, 2013.

[26] A. Wombacher, P. Fankhauser, B. Mahleko, and E. Neuhold, "Matchmaking for business processes based on choreographies," in *e-Technology, e-Commerce and e-Service, 2004. EEE'04. 2004 IEEE International Conference on*. IEEE, 2004, pp. 359–368.

[27] E. Fabre and A. Benveniste, "Partial order techniques for distributed discrete event systems: Why you cannot avoid using them," *Discrete Event Dynamic Systems*, vol. 17, no. 3, pp. 355–403, 2007.

APPENDIX

A. Proofs of Proposition 2 and Proposition 3

In order to prove the correctness of the protocol for private VE process fusion, we need several properties of language projection and product.

Definition 2 (Generalized projection π): Let Σ_1 be an alphabet and $\mathcal{L}_1$ be a language over that alphabet. Projection is generalized to an arbitrary alphabet Σ_2 as follows:

$$\pi_{\Sigma_2}(\mathcal{L}_1) = \mathbf{proj}_{\Sigma_2}^{-1}\ (\mathbf{proj}_{\Sigma_1\cap\Sigma_2}(\mathcal{L}_1))$$

Notice that if $\Sigma' \subseteq \Sigma$ then $\pi_{\Sigma'}(\mathcal{L}) = \mathbf{proj}_{\Sigma'}(\mathcal{L})$. Further, we have the following properties of projection and reverse projection.

Proposition 4: Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be two languages over alphabets Σ_1 and Σ_2, respectively, and let $\Sigma_1 \subseteq \Sigma_2$. Then:

$$\mathbf{proj}_{\Sigma_1}(\mathbf{proj}_{\Sigma_2}^{-1}(\mathcal{L}_1)) = \mathcal{L}_1$$

In special cases, projection distributes over language intersection.

Proposition 5: Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be two languages over alphabets Σ_1 and Σ_2, respectively, and let $\Sigma_1 \subseteq \Sigma_2$. Then:

$$\mathbf{proj}_{\Sigma_1}(\mathbf{proj}_{\Sigma_2}^{-1}(\mathcal{L}_1) \cap \mathcal{L}_2) = \mathcal{L}_1 \cap \mathbf{proj}_{\Sigma_1}(\mathcal{L}_2)$$

On the other hand, reverse projection *does* distribute over language intersection.

Proposition 6: Let $\Sigma' \subseteq \Sigma$, and $\mathcal{L}_1, \mathcal{L}_2$ be two languages over Σ'. We have:

$$\mathbf{proj}_{\Sigma}^{-1}(\mathcal{L}_1 \cap \mathcal{L}_2) = \mathbf{proj}_{\Sigma}^{-1}(\mathcal{L}_1) \cap \mathbf{proj}_{\Sigma}^{-1}(\mathcal{L}_2)$$

Proposition 7: Let $\mathcal{L}_1$ and $\mathcal{L}_2$ two languages over alphabets Σ_1 and Σ_2, respectively. We have:

$$\mathbf{proj}_{\Sigma_1}(\mathcal{L}_1 \times^L \mathcal{L}_2) = \mathcal{L}_1 \cap \pi_{\Sigma_1}(\mathcal{L}_2)$$

We are now ready to prove the two propositions from the previous subsection.

Proof of Proposition 2. We have:

$$\begin{array}{lll} & \mathbf{proj}_{\Sigma_1}(\mathcal{L}_1 \times^L \mathcal{L}_2) & \\ = & \mathcal{L}_1 \cap \pi_{\Sigma_1}(\mathcal{L}_2) & \{\text{Prop. 7}\} \\ = & \mathcal{L}_1 \cap \mathbf{proj}_{\Sigma_1}^{-1}(\mathbf{proj}_{\Sigma_1\cap\Sigma_2}(\mathcal{L}_2)) & \{\text{Def. 2}\} \\ = & \mathcal{L}_1 \cap \mathbf{proj}_{\Sigma_1}^{-1}(\mathbf{proj}_{\Sigma_1\cap\Sigma_2}(\mathcal{L}_1)) \cap & \\ & \mathbf{proj}_{\Sigma_1}^{-1}(\mathbf{proj}_{\Sigma_1\cap\Sigma_2}(\mathcal{L}_2)) & \{\text{Prop. 4, Set theory}\} \\ = & \mathcal{L}_1 \cap \mathbf{proj}_{\Sigma_1}^{-1}(\mathcal{L}) & \{\text{Prop. 6, Def. } \mathcal{L}\} \end{array}$$

This concludes the proof. □

Proof of Proposition 3. We have:

$$\begin{array}{lll} & \mathcal{L} & \\ = & \mathbf{proj}_{\Sigma_1\cap\Sigma_2}(\mathcal{L}_1) \cap \mathbf{proj}_{\Sigma_1\cap\Sigma_2}(\mathcal{L}_2) & \{\text{Def. } \mathcal{L}\} \\ = & \mathbf{proj}_{\Sigma_1\cap\Sigma_2}(\mathcal{L}_1 \cap \mathbf{proj}_{\Sigma_1}^{-1}(\mathbf{proj}_{\Sigma_1\cap\Sigma_2}(\mathcal{L}_2))) & \{\text{Prop. 5}\} \\ = & \mathbf{proj}_{\Sigma_1\cap\Sigma_2}(\mathcal{L}_1 \cap \pi_{\Sigma_1}(\mathcal{L}_2)) & \{\text{Def. 2}\} \\ = & \mathbf{proj}_{\Sigma_1\cap\Sigma_2}(\mathbf{proj}_{\Sigma_1}(\mathcal{L}_1 \times^L \mathcal{L}_2)) & \{\text{Prop. 7}\} \end{array}$$

This concludes the proof. □