An Efficient Protocol for Oblivious DFA Evaluation and Applications

Payman Mohassel * Salman Niksefat † Saeed Sadeghian ‡
Babak Sadeghiyan §

Abstract

In this paper, we design an efficient protocol for *oblivious DFA evaluation* between an input holder (client) and a DFA holder (server). The protocol runs in a single round, and only requires a small amount of computation by each party. The most efficient version of our protocol only requires $O(k)$ asymmetric operations by either party, where k is the security parameter. Moreover, the client's total computation is only linear in his own input and independent of the size of the DFA. We prove the protocol fully-secure against a *malicious client* and *private* against a malicious server, using the standard *simulation-based* security definitions for secure two-party computation.

We show how to transform our construction in order to solve multiple variants of the *secure pattern matching* problem without any computational overhead. The more challenging variant is when parties want to compute the number of occurrences of a pattern in a text (but nothing else). We observe that, for this variant, we need a protocol for counting the number of accepting states visited during the evaluation of a DFA on an input. We then introduce a novel modification to our original protocol in order to solve the counting variant, without any loss in efficiency or security.

Finally, we fully implement our protocol and run a series of experiments on a client/server network environment. Our experimental results demonstrate the efficiency of our proposed protocol and, confirm the particularly low computation overhead of the client.

1 Introduction

In the oblivious Deterministic Finite Automata (DFA) evaluation problem, the first party (Server) holds a DFA Γ, while the second party (Client) holds an input string X. Their goal is to collaboratively evaluate the DFA Γ on input X, allowing one or both of the participants to learn the result $\Gamma(X)$ without learning any additional information about each other's input. A number of applications with security and privacy concerns can be efficiently formulated as DFA evaluation and be implemented using secure two-party protocols for oblivious DFA evaluation.

One such example is the problem of *secure pattern matching* (or text processing) and its variants which have been the focus of several recent works in the literature [21, 3, 13, 4, 8]. In the most common variant of the problem, one is interested in finding the locations of a specific pattern p in a text T. Pattern matching has immediate applications in mining and processing DNA data and is often used in practice, e.g. in the Combined DNA Index System (CODIS)[1] run by the FBI for DNA identity testing. There are privacy concerns associated with algorithms that process individual's DNA data and, not surprisingly, privacy issues are the main motivation behind most of the above-mentioned works on designing secure solutions.

*University of Calgary. pmohassel@cspc.ucalgary.ca.
†Amirkabir University of Technology. sniksefa@ucalgary.ca. Work done while visiting University of Calgary.
‡Amirkabir University of Technology. ssadeghian@aut.ac.ir.
§Amirkabir University of Technology. basadegh@aut.ac.ir.
[1] http://www.fbi.gov/hq/lab/codis.

One can formulate the basic variant of the pattern matching problem as the evaluation of a *pattern-specific* automaton Γ_p on a text T [14]. In fact, several of the papers mentioned above solve the secure pattern matching problem by designing protocols for oblivious evaluation of Γ_p on T [4, 21, 3].

Depending on the application being considered, it can be the case that the size of the input string to the DFA is large (e.g. the text T in secure pattern matching), or the size of the DFA Γ itself (e.g. when many patterns are combined into one DFA). Therefore, *for an oblivious DFA evaluation protocol to be a viable solution for practice, it needs to ensure efficiency and scalability when run on large DFAs and/or input strings.* Towards this goal, we focus on the following three efficiency criteria:

- **Small number of asymmetric operations:** Based on existing benchmarks (e.g. `http://bench.cr.yp.to`) asymmetric operations (e.g. exponentiation) require *several thousand* times more cpu cycles compared to their symmetric-key counterparts. Hence, for an ODFA protocol to be scalable for large input strings and large DFA sizes, it is essential to minimize the number of asymmetric operations and to ensure that their number does not grow with the size of the DFA and/or its input. ODFA protocols of [21] and [4] do not satisfy this property since the number of exponentiations they require is linear in the size of the DFA and its input.

- **Small computation for the input holder (client):** In practice, the two involved parties do not always have the same computational resources, and hence it is common to implement the protocols in a client/server model where one party has to perform noticeably less work. Motivated by this concern, we require that the input holder's (client) total work be significantly smaller, and in particular be independent of the size of the server's DFA. All previous solutions for ODFA, including a general solution based on Yao's garbled circuit protocol fail to achieve this goal.

- **Small number of rounds of interaction:** we also require our protocols to have a small number of rounds of interactions (ideally a single round). A *single round* of interaction allows the protocol to be deployed in a non-interactive setting where one party can communicate his message, go offline and connect at a later time to retrieve the final message. Therefore, very little online coordination and computation is necessary.

As mentioned above, the existing solutions for oblivious DFA evaluation do not meet one or more of the above efficiency criteria.

1.1 Our Contributions

A New Protocol For Oblivious DFA Evaluation. Our main contribution is a new and efficient protocol for oblivious DFA evaluation that meets all three of the above-mentioned efficiency criteria. The most efficient variant of our construction runs in one and a half rounds, and only requires $O(k)$ asymmetric operations by either party where k is the security parameter. Moreover, the input holder's total work is only linear in his input and is independent of the DFA size.

We prove the protocol fully-secure against a *malicious client* and *private* against a *malicious server*, using the standard *simulation-based* security definitions for secure two-party computation (see Section 2.4).

Our starting point is a single round protocol between a server who holds the DFA Γ with $|Q|$ states and a client who holds an n-bit input string X. The basic idea is for the server to represent the evaluation of Γ on an arbitrary n-bit string X via a $n \times |Q|$ DFA matrix M_Γ. A DFA matrix is a simple data structure used to efficiently evaluate the DFA on any input string of size n. The server *permutes* and *garbles* this matrix into a garbled DFA matrix GM_Γ and sends it to the client.

After the garbling stage, the server and the client engage in a series of oblivious transfer protocols where the client learns a vector of random keys corresponding to his input X. These random keys allow the client to ungarble a unique path that starts from the first row of the matrix and ends in the last row. This path (referred to as the *transit path*) corresponds to the evaluation of input X using the DFA matrix M_Γ. The client can extract the final output of evaluation from this ungarbled transit path but is not able to ungarble any of the remaining elements in the matrix, or learn any additional information about the DFA.

The number of OTs can be made independent of the client's input size (i.e., the number of OTs remains the same, as the input size increases) via use of the OT extension protocol of [11]. More precisely, this extension reduces the number of exponentiations necessary from $O(n)$ to $O(k)$, but increases the number of rounds from a single round to one and a half round.

Comparison with Yao's protocol. We note that the above approach for DFA evaluation is reminiscent of the Yao's garbled circuit protocol [22, 16], where the circuit being evaluated is garbled and a set of random keys are used to ungarble and evaluate the circuit on a specific input. In fact, it is possible to use Yao's garbled circuit protocol to implement oblivious DFA evaluation. One party's input to the circuit is his input string while the other party's input is the DFA itself. However, as discussed in [4], the resulting protocol would be significantly less efficient compared to ours. Moreover, unlike our construction, an implementation of ODFAs via a direct application of Yao's garbled circuit would yield a protocol wherein the amount of work the client has to perform is linear in the size of the circuit and hence at least linear in the size of the DFA. Such a protocol would not satisfy our second efficiency criteria.

However, as pointed out by one of the reviewers of our paper at CT-RSA 2012, an alternative way of presenting of our construction is to describe it as a generalization of Yao's garbled circuit protocol where the gates are allowed to take non-boolean inputs and return non-boolean outputs. We discuss this variant in more detail, in Section 4.5.

We give a more detailed comparison of efficiency between our protocol and the existing solutions in Section 4.6.

Applications to Secure Pattern Matching. We show how to use our Oblivious DFA evaluation protocol to efficiently solve multiple variants of the *Secure Pattern Matching* problem. In the three main variants we consider, one party holds a text T while the other party holds a pattern p and the aim is for the first party to learn one of the following but nothing else: (i) whether or not p appears in T, (ii) all the locations (if any) where p occurs as a pattern in T, or (iii) the number of occurrences of pattern p in T, while the text holder learns nothing about the pattern.

The first two variants can be implemented in a relatively straightforward manner, using appropriate pattern-specific DFAs. In the third variant, we need to count the number of occurrences of a pattern p in a text T. As discussed in [13], the number of occurrences of a pattern p is in fact what some applications of pattern matching are interested in. It is not clear how to directly cast this problem as a DFA evaluation problem and unlike the existing solutions for the second variant, we need to hide the locations where the patterns occur from both parties. It is not obvious how to modify any of the existing secure pattern matching constructions to solve this variant of the problem without a noticeable increase in complexity.

To design an efficient protocol for this task, we show how to modify our oblivious DFA evaluation protocol so that it returns the total number of times that accepting state(s) are visited during the evaluation of an input, instead of a single bit indicating an accept/reject final state. In particular, we embed a series of "random looking but correlated" values in the DFA matrix before garbling it and show how to modify the original protocol to let the evaluator of the garbled DFA matrix recover all the *embedded strings* on the transit path. The evaluator then uses these values to compute the number of accepting states visited without learning any additional information. The resulting protocol's complexity is similar to our original ODFA construction. When applied to the pattern-specific DFA of [14], our construction automatically yields a secure protocol for counting the number of occurrences of a pattern p in a text T. This new variant of ODFA maybe be of independent interest in other applications as well.

Implementation and Experimental Results. We fully implement our main ODFA protocol in a client/server network environment and use the OT extension of [11] to implement the oblivious transfer component. We measure the performance of our implementation for a wide range of input and DFA sizes. Experiments are ran on two machines as the client and the server, each with an Intel Core i7 processor with 4GB of RAM and connected via a Gigabit Ethernet. Our experiments confirm our theoretical arguments on the scalability of our protocol. For instance, on 20-bit inputs and for DFA sizes of as large as 15000 states, or for DFAs with 20 states and inputs as large as 10000-bits, our protocol runs in less than 1 second. These numbers remain fairly low (under 12 seconds) even when we increase the number of states or the input bits to 150000. Our experiments show that the client's computation is very low, such that for the case of a DFA

with 20 states and inputs of size 150000 bits, his computation hardly reaches 1 second. For the case of 20-bit inputs and 150000 state DFAs, client's computation is even smaller (less than 32 milliseconds). This confirms the suitability of our protocol for client/server settings, where the input holder has limited computational resources.

We also note that since we use the OT extensions of [11], OTs are no longer the computational bottleneck for the server. The main bottleneck for large inputs and DFAs is the computation the server performs to garble the DFA matrix. A more detailed discussion of the implementation and the results of experiments are given in Section 6.

1.2 Related Work

To the best of our knowledge, the first scheme for *oblivious DFA evaluation* was proposed in [21] (motivated by the problem of privacy preserving DNA pattern matching). Their construction is not constant round and only provides security against semi-honest adversaries. This work was later improved by Frikken [3] who designed a protocol that runs in a constant number of rounds (more than one) and has fewer asymmetric computation (exponentiation). This work also only considers semi-honest adversaries.

[4] is the only work on oblivious DFA evaluation that considers malicious adversaries but requires $\min(O(|Q|), O(n))$ rounds of interaction and $O(n|Q|)$ asymmetric computations, where n is the input size and $|Q|$ is the number of states in the DFA. The security of our protocol against the input holder is similar to that of [4], but we achieve a weaker notion of security against a malicious DFA holder (see Section 4 for more detail). It is also possible to use Yao's garbled circuit protocol to implement oblivious DFA evaluation, but as discussed above, the resulting protocol would not satisfy our efficiency criteria.

The problem of oblivious DFA evaluation can also be formulated as computation on encrypted data and be implemented using the construction of [12] for branching programs or the recent fully homomorphic encryption schemes [5]. The problem with these schemes is their high computation cost as the number of times the corresponding public-key encryption schemes are invoked is at least linear in the DFA size and its input. See Table 1 for a more detailed comparison of our protocol with the existing solution for oblivious DFA evaluation.

We also briefly review the status of protocols for *secure pattern matching* here. Let n be the text size and m be the pattern size. The protocol of [4] runs in $O(m)$ rounds and requires $O(mn)$ exponentiations. The constructions of [7] and [8] run in a constant number of rounds (more than one) and require $O(n+m)$ exponentiations. For long texts, where n is large, this renders the exponentiations a major computational overhead. In contrast, an extended version of our protocol (in random oracle model) only requires $O(k)$ exponentiations where k is the security parameter. This improves the efficiency significantly when $n \gg k$.

2 Preliminaries

In this section, we introduce the notations, definitions and primitives used in the rest of the paper.

2.1 Notations

Throughout the paper, we use k to denote the security parameter. We denote an element at row i and column j of a matrix by $M[i,j]$. If the element itself is a pair we use $M[i,j,0]$ to denote the first value of the pair and $M[i,j,1]$ to denote the second value. Vectors are denoted by over-arrowed lower-case letters such as $\vec{v}$. We use $a||b$ to denote the concatenation of the strings a and b. λ is used to denote an empty string and a^b denotes b consecutive concatenation of the string a by itself.

We denote a random permutation function by $Perm$. $\vec{v} \leftarrow Perm(Q)$ takes as input a set of integers $Q = \{1, \ldots, |Q|\}$, permutes the set uniformly at random and returns the permuted elements in a row vector v of dimension $|Q|$. We call a matrix a *permutation matrix* if all of its rows are generated in this way. The following simple algorithm (algorithm 1) can be used to generate a permutation matrix PER with n rows from the set Q.

Algorithm 1 GenPerm(n, Q)

for $1 \leq i \leq n$ **do**
  $PER[i] \leftarrow Perm(Q)$
end for
return PER

2.2 Oblivious Transfer

Our protocols use Oblivious Transfer (OT) as a building block. Since we mostly focus on protocols that run in a single round (with the exception of the enhancement in Section 4.4), we describe an abstraction for one-round OT protocols here. A One-round OT involves a server holding a list of t secrets $(s_1, s_2, \ldots, s_t)$, and a client holding a selection index i. The client sends a query q to the server who responds with an answer a. Using a and its local secret, the client is able to recover s_i.

More formally, a one-round 1-out-of-t oblivious transfer (OT_1^t) protocol is defined by a tuple of PPT algorithms $OT_1^t = (\mathrm{G_{OT}}, \mathrm{Q_{OT}}, \mathrm{A_{OT}}, \mathrm{D_{OT}})$. The protocol involves two parties, a client and a server where the server's input is a t-tuple of strings $(s_1, \ldots, s_t)$ of length τ each, and the client's input is an index $i \in [t]$. The parameters t and τ are given as inputs to both parties. The protocol proceeds as follows:

1. The client generates $(pk, sk) \leftarrow \mathrm{G_{OT}}(1^{\mathrm{k}})$, computes a query $q \leftarrow \mathrm{Q_{OT}}(\mathrm{pk}, 1^{\mathrm{t}}, 1^{\tau}, \mathrm{i})$, and sends (pk, q) to the server.

2. The server computes $a \leftarrow \mathrm{A_{OT}}(\mathrm{pk}, \mathrm{q}, \mathrm{s}_1, \ldots, \mathrm{s}_{\mathrm{t}})$ and sends a to the client.

3. The client computes and outputs $\mathrm{D_{OT}}(\mathrm{sk}, \mathrm{a})$.

In case of semi-honest adversaries many of the OT protocols in the literature are one-round protocols (e.g. see [18, 17]). In case of malicious adversaries, in the CRS model, one can use the one-round OT protocols of [19].

2.3 Pseudorandom Generator

A computationally secure pseudorandom generator (PRG) is a (deterministic) map $G : \{0,1\}^{\ell} \rightarrow \{0,1\}^n$ where ℓ is the "seed length" and $n - \ell \geq 0$ is the "stretch". G should be polynomial-time computable and for any PPT distinguisher D the following should be negligible in k

$$|\Pr[D(U_m) = 1] - \Pr[D(G(U_\ell)) = 1]$$

where U_m denotes a uniformly random string in $\{0,1\}^m$. Here the string U_ℓ is called the "seed".

2.4 Secure Two-party Computation

Let $f = (f_1, f_2)$ of the form $f : \{0,1\}^* \times \{0,1\}^* \rightarrow \{0,1\}^* \times \{0,1\}^*$ be a two party computation and π be a two-party protocol for computing f between the parties p_1 and p_2. The input of p_1 is x and the input of p_2 is y. We briefly review two notions of security for secure two-party computation here, i.e. (i) full security (simulation-based security) and (ii) privacy, both against a malicious adversary. In particular, in our protocols, we can prove full-security against one malicious party and only privacy against the other.

2.4.1 Full-Security Against Malicious Adversaries

Readers can refer to [6] for a detailed discussion. Full-security for a two-party computation is defined by requiring indistinguishability (either perfect, statistical or computational) between a real execution of the protocol and an ideal execution in which there is a TTP (trusted third party) who receives the parties input, evaluates the function and outputs the results to them. Consider a malicious and *admissible* adversary $\mathcal{A}$. An admissible adversary is one that corrupts exactly one of the two party. $\mathcal{A}$ also knows an auxiliary input z. Without loss of generality we assume the $\mathcal{A}$ corrupts the first party.

In the real world, the honest party follows the description of protocol π as instructed and responds to messages sent by $\mathcal{A}$ on behalf of the other party. Let $view_{\pi,\mathcal{A}}(x,y)$ denote $\mathcal{A}$'s view through this interaction, and let $out_{\pi}(x,y)$ denote the output of the honest party. The execution of π in the real model on input pair (x,y) is defined as follows:

$$\mathrm{REAL}_{\pi,\mathcal{A}(\mathrm{z})}(\mathrm{x},\mathrm{y}) \stackrel{\mathrm{def}}{=} (\mathrm{view}_{\pi,\mathcal{A}}(\mathrm{x},\mathrm{y}), \mathrm{out}_{\pi}(\mathrm{x},\mathrm{y}))$$

In the ideal model, in which there is a TTP, the second (honest) party always sends its input y to TTP, while the first (corrupted) party can send an arbitrary input x'. The TTP first replies to the first party with $f_1(x',y)$. Otherwise (i.e., in case it receives only one valid input), the trusted party replies to both parties with a special symbol $\perp$. In case the first party is malicious it may, depending on its input and the trusted party's answer, decide to stop the trusted party by sending it $\perp$ after receiving its output. In this case the trusted party sends $\perp$ to the second party. Otherwise (i.e., if not stopped), the trusted party sends $f_2(x,y)$ to the second party. The honest party outputs whatever is sent by the trusted party, and $\mathcal{A}$ outputs an arbitrary function of its view. Let $out_{f,\mathcal{A}}(x,y)$ and $out_f(x,y)$ denote the output of $\mathcal{A}$ and the honest party respectively in the ideal model. The execution of π in the ideal model on inout pair (x,y) is defined as follow:

$$\mathrm{IDEAL}_{\mathrm{f},\mathcal{A}(\mathrm{z})}(\mathrm{x},\mathrm{y}) \stackrel{\mathrm{def}}{=} (\mathrm{out}_{\mathrm{f},\mathcal{A}}(\mathrm{x},\mathrm{y}), \mathrm{out}_{\mathrm{f}}(\mathrm{x},\mathrm{y}))$$

Definition 1. *We say that π securely computes f in the presence of static malicious adversaries if for every pair of admissible non-uniform probabilistic polynomial-time machines $\bar{A} = (A_1, A_2)$ in the real model, there exists a pair of admissible nonuniform probabilistic expected polynomial-time machines $\bar{B} = (B_1, B_2)$ in the ideal model, such that*

$$\left\{\mathrm{IDEAL}_{f,\bar{B}}(x,y)\right\} \equiv \left\{\mathrm{REAL}_{\pi,\bar{A}}(x,y)\right\}$$

Namely the two distributions are indistinguishable.

2.4.2 Privacy Against Malicious Adversaries

In our protocols, for the party holding the input to the DFA, we achieve a weaker notion of security against malicious adversaries. Intuitively, this level of security guarantees that a corrupted party will not learn any information about the honest parties input. However, this does not always guarantee that the parties joint outputs in the real world is simulatable in an ideal world. We formally describe this notion of security next. Without loss of generality we assume that the first party is the malicious one.

Definition 2. *We say that protocol π is private against a malicious party p_1 if the advantage of any non-uniform polynomial-time adversary $\mathcal{A}$ corrupting p_1 in the real world is negligible in the following game:*

- *$\mathcal{A}$ is given 1^k and generates $y_0, y_1 \in \{0,1\}^n$ for some positive integer n and sends it to p_2.*
- *p_2 generates a random bit $b \stackrel{\$}{\leftarrow} \{0,1\}$. He then uses y_b as his input in protocol π.*
- *At the end of the protocol π, $\mathcal{A}$ should output a bit b'.*

$\mathcal{A}$'s advantage is defined as $\Pr[b' = b] - 1/2$.

2.4.3 The OT hybrid model

We use the OT hybrid model to prove the security of our proposed protocols. In the OT hybrid model (e.g. see [1, 15]), it suffices to analyze the security of a protocol in a hybrid model in which the parties interact with each other and have access to a trusted party (ideal functionality) that computes the oblivious transfer protocol for them. This model is a hybrid of the real and ideal models: on the one hand, the parties send regular messages to each other, similar to the real model; on the other hand, the parties have access to a trusted party, similar to the ideal model.

3 DFA and its Matrix Representation

3.1 DFA

In this paper a deterministic finite automaton (DFA) [20] is denoted by a 5-tuple $\Gamma = (Q, \Sigma, \Delta, s_1, F)$, where Q is a finite set of states, Σ is a finite input alphabet, $s_1 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, and Δ denotes the transition function. Thus $|Q|$ denotes the total number of states. We represent states by integers in $Z_{|Q|}$. $\Delta(j, \alpha)$ returns the next state when the DFA is in state $j \in Q$ and sees an input $\alpha \in \Sigma$. A string $X = x_1 x_2 \dots x_n \in \Sigma^n$ is said to be accepted by Γ if the state $s_n = \Delta(\dots \Delta(\Delta(s_1, x_1), x_2) \dots, x_n)$ is a final state $s_n \in F$. A binary DFA is a DFA with $\Sigma = \{0, 1\}$. From this point forward, we restrict our attention to binary DFAs and the term DFA is used for binarty DFAs.

Our oblivious evaluation protocols take advantage of a *matrix representation* of DFAs. Next we define the notions of a *DFA matrix* and a *permuted DFA matrix* which we use throughout the paper.

3.2 DFA Matrix

Assume that the input string of a DFA $\Gamma = (Q, \{0, 1\}, \Delta, s_1, F)$ is a bitstring $X = x_1 x_2 \dots x_n \in \{0, 1\}^n$. Then we can represent the evaluation of Γ on an arbitrary input X of length n as a matrix M_Γ of size $n \times |Q|$. For $1 \leq i \leq n$, the ith row of M_Γ represents the evaluation of x_i. In particular, the element $M_\Gamma[i][j]$ stores the pair $(\Delta(j, 0), \Delta(j, 1))$ which encodes the indices of the next two states to be visited (at row $i + 1$) for input bits $x_i = 0$ and $x_i = 1$, respectively. At row n where the last bit x_n is processed, instead of storing the indices of the next states, we place a 1 if the next state is an accepting one and a 0 otherwise. An example of a general DFA and its DFA matrix are depicted in Figure 1.

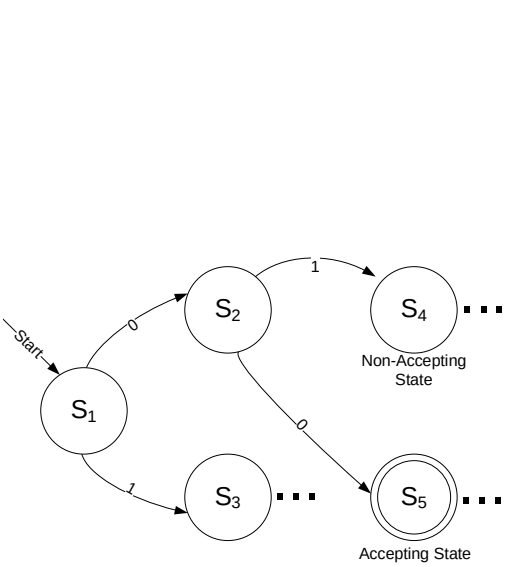
Algorithm 2 describes the function $\mathsf{DfaMat}(\Gamma, n)$ which takes a DFA Γ and the input size n, as its input and generates the DFA matrix M_Γ.

Algorithm 2 $\mathsf{DfaMat}(\Gamma, n)$

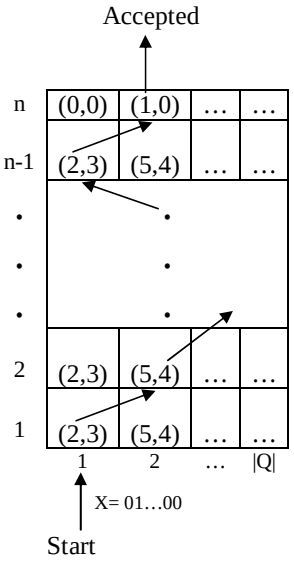
```
for 1 ≤ i ≤ n do
  for 1 ≤ j ≤ |Q| do
    if i ≤ n − 1 then
      M_Γ[i, j] ← (Δ(j, 0), Δ(j, 1))
    else if i = n then
      res_0 ← (Δ(j, 0) ∈ F)?1 : 0
      res_1 ← (Δ(j, 1) ∈ F)?1 : 0
      M_Γ[n, j] ← (res_0, res_1)
    end if
  end for
end for
return M_Γ
```

Evaluation using the DFA Matrix. One can use M_Γ to efficiently evaluate Γ on any n bit input X. We start at $M_\Gamma[1, 1]$. If $x_1 = 0$, the first index of the pair $M_\Gamma[1, 1]$ is used to located the next cell to visit at row 2. If $x_1 = 1$, the second index of $M_\Gamma[1, 1]$ is used instead. Then by considering the chosen pair in row 2 and the value of x_2, one can find the next pair to visit in row 3. This process is repeated until we reach row n and read either 0 or 1 which will be the result of the evaluation of X on Γ.

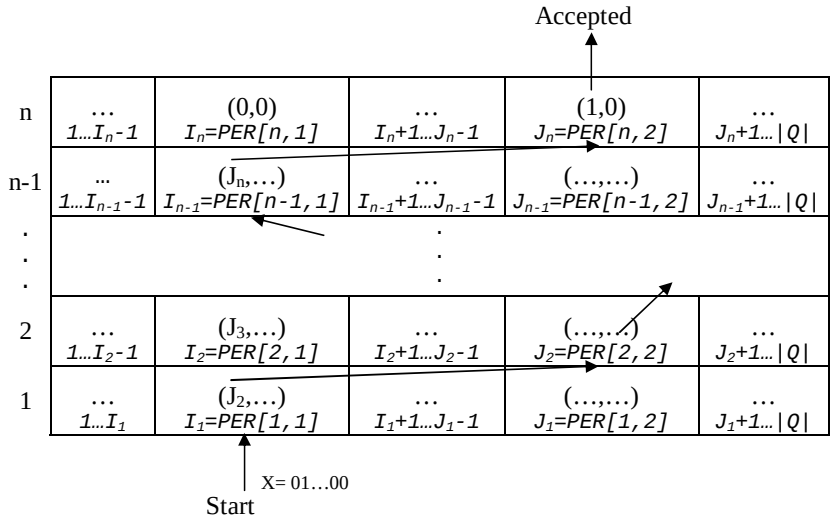
When evaluating an input string X using a DFA matrix, we call the set of pairs visited starting from row 1 upto row n a *transit path* for X. A transit path either ends with 1 which shows that X is accepted by Γ or ends with 0 which shows that X is not accepted by Γ. A sample transit path of $X = 10\dots$ for the sample DFA matrix is depicted in Figure 1.



(a) A general DFA Γ



(b) M_Γ (DFA Matrix of Γ)



(c) PM_Γ (Permuted DFA Matrix of Γ)

Figure 1: DFA Γ, DFA matrix M_Γ, Permuted DFA Matrix PM_Γ

3.3 Permuted DFA Matrix

A permuted DFA matrix PM_Γ is generated by randomly permuting the elements in each row i of M_Γ and updating the associated indices in row $i-1$ accordingly to point to the new permuted indices of row i. In order to do this, we first generate a permutation matrix PER of size $n \times |Q|$ using the GenPerm algorithm 1. Then, algorithm 3 is invoked to convert a DFA matrix M_Γ to an equivalent permuted DFA Matrix PM_Γ. Figure 1(c) depicts the DFA permuted Matrix for the DFA in Figure1.

Algorithm 3 PermDfaMat(M_Γ, PER)

for $1 \leq i \leq n$ **do**
  for $1 \leq j \leq |Q|$ **do**
    if $i \leq n-1$ **then**
      $PM_\Gamma[i, PER[i,j], 0] \leftarrow PER[i+1, M_\Gamma[i,j,0]]$
      $PM_\Gamma[i, PER[i,j], 1] \leftarrow PER[i+1, M_\Gamma[i,j,1]]$
    else if $i = n$ **then**
      $PM_\Gamma[n, PER[n,j], 0] \leftarrow M_\Gamma[i,j,0]$
      $PM_\Gamma[n, PER[n,j], 1] \leftarrow M_\Gamma[i,j,1]$
    end if
  end for
end for
return PM_Γ

Evaluating an input using the permuted DFA matrix is almost identical to the normal DFA matrix with the exception that the evaluation begins at $PM_\Gamma[1, PER[1,1]]$.

4 An Efficient Protocol for Oblivious DFA Evaluation

Let the server hold a *private* deterministic finite automata (DFA) $\Gamma = (Q, \{0,1\}, \Delta, s_1, F)$ and the client hold a private string $X = x_1 x_2 \dots x_n \in \{0,1\}^n$. Our goal is to let the client discover whether his string X is accepted by the server's DFA Γ or not without revealing anything about X and Γ to server and client, respectively. In this section we propose a new and efficient protocol for Oblivious DFA evaluation. The main version of our protocol is a single-round construction that only requires $O(n)$ exponentiations for both server and client. Considering the fact that exponentiation is the computational bottleneck (the other operations are XORing and indexing), for practical purposes, parties' computation only depends on the length of client's input and is independent of the size of DFA. In the random oracle model and using the OT extension of [11], we can make this number independent of client's input by further reducing the number of exponentiations to $O(k)$, where k is the security parameter (at the cost of adding an extra round). In situations where $n \gg k$ which is the case in many applications of DFAs in practice, this leads to a noticeable improvement in efficiency.

We prove the security of our proposed protocol using the standard simulation-based definitions of security for two-party computation (see Section 2.4).

4.1 A High Level Overview

Before describing our protocol in more detail we start with a high level overview.

Client gets his input keys. For every bit of client's input x_i, server and client engage in an oblivious transfer where server's inputs are two random key strings (K_i^0, K_i^1) corresponding to bit values 0 and 1. As a result client learns one of the keys in each pair.

Server computes a garbled DFA Matrix. In this stage, server (the holder of the DFA Γ) first computes a permuted DFA matrix PM_Γ corresponding to her DFA by calling DfaMat(), GenPerm() and PermDfaMat() algorithms (See Section 3). The permutations are done to help with hiding the structure of the DFA from client.

Server then garbles the permuted DFA matrix in a special way. To garble the matrix server first generates a $n \times |Q|$ matrix PAD filled with random strings. Consider a pair (a_0, a_1) stored in the cell $PM_\Gamma[i, j]$ of the permuted matrix. Each value in the pair is encrypted using a one-time pad encryption where the pad is a combination of the strings in PAD (i.e. $PAD[i, j]$) and the input key strings K_i^0, and K_i^1. More specifically, a_0 is encrypted using K_i^0 while a_1 is encrypted using K_i^1. Then, the resulting ciphertexts are concatenated and encrypted using an expanded version of $PAD[i, j]$ using a PRG G. All the encryptions are one-time pad encryptions.

Note that client can only decrypt a_b if he knows both the correct input key K_i^b and the random string $PAD[i, j]$. Client will learn one of the two input keys through the oblivious transfer, but this is not sufficient for decrypting either value in the pair. Client learns $PAD[i, j]$ only if he is visiting from a legitimate previous state in the DFA. In order to enforce the latter, $PAD[i, j]$ is concatenated to the appropriate value (i.e. index) already stored in $PM_\Gamma[i-1, j']$, where j' is the permuted index (at row $i-1$) of a legitimate previous state. It is only after this concatenations that the matrix is garbled using the one-time pads described above.

Server sends the resulting garbled DFA matrix GM_Γ plus the index and the pad of starting cell row 1 to client. Note that the PAD matrix is not sent to client.

Client evaluates the garbled DFA matrix. Client uses the input keys he retrieves at the OT stage, to decrypt one of the two values in the starting pair. As a result, he learns the index to a single pair in the next row in addition to a random pad that he uses to partially decrypt the values in that pair. He then decrypts exactly one of the values in the pair (completely) using the retrieved key for his second input bit.

He repeats this process, moving along the *transit path* for input X until he reaches the last row and recovers the final output. First, note that for all the elements not on his transit path, client does not learn the corresponding random string in the PAD matrix and hence those elements remain garbled to him. For those pairs that appear on his path, he can only decrypt one of the two values using the single input key he has retrieved at the OT stage. This rough intuition behind the security against a malicious client is formalized in the security proof.

4.2 The Protocol 1

Server's Input: A DFA $\Gamma = (Q, \{0,1\}, \Delta, s_1, F)$.
Client's Input: A bitstring $X = x_1 x_2 ... x_n \in \{0,1\}^n$.
Common Input: The security parameter k, the OT security parameter κ and the size of DFA $|Q|$. We let $k' = k + \log|Q|$ throughout the protocol. Parties also agree on a 1-out-of-2 OT protocol $OT = (\mathrm{G_{OT}}, \mathrm{Q_{OT}}, \mathrm{A_{OT}}, \mathrm{D_{OT}})$ and a PRG $G : \{0,1\}^k \to \{0,1\}^{2k'}$.

1. **Client encrypts his inputs using OT queries, and sends them to server.**

Sending OT Queries to server

Client computes $(pk, sk) \leftarrow \mathrm{G_{OT}}(1^\kappa)$
for $1 \le i \le n$ **do**
  client computes $q_i \leftarrow \mathrm{Q_{OT}}(pk, 1^2, 1^{k'}, x_i)$
end for
Client sends pk and $\vec{q} = (q_1, q_2, \ldots, q_n)$ to server.

2. **Server Computes a Garbled DFA matrix GM_Γ.**

Generating random pads and a permuted DFA matrix PM_Γ

SERVER GENERATES n RANDOM KEY PAIRS FOR THE OTs:
for $1 \leq i \leq n$ **do**
  $(K_i^0, K_i^1) \xleftarrow{\$} \{0,1\}^{k'}$
end for
SERVER GENERATES A RANDOM PAD MATRIX $PAD_{n\times|Q|}$:
for $i = 1$ to n and $j \in Q$ **do**
  $PAD[i,j] \xleftarrow{\$} \{0,1\}^k$
end for
SERVER GENERATES A DFA MATRIX M_Γ:
$M_\Gamma \leftarrow \mathsf{DfaMat}(\Gamma, n)$
SERVER GENERATES A RANDOM PERMUTATION MATRIX $PER_{n\times|Q|}$:
$PER \leftarrow \mathsf{GenPerm}(n, Q)$
SERVER GENERATES A PERMUTED DFA PERMUTED MATRIX PM_Γ:
$PM_\Gamma \leftarrow \mathsf{PermDfaMat}(M_\Gamma, PER)$

Computing the Garbled DFA Matrix GM_Γ from PM_Γ

for each row $i = 1$ to n **do**
  for each $j \in Q$ **do**
    if $1 \leq i \leq n-1$ **then**
      $GM_\Gamma[i,j,0] \leftarrow PM_\Gamma[i,j,0]||PAD[i+1, PM_\Gamma[i,j,0]]$
      $GM_\Gamma[i,j,1] \leftarrow PM_\Gamma[i,j,1]||PAD[i+1, PM_\Gamma[i,j,1]]$
    else if $i = n$ **then**
      $GM_\Gamma[n,j,0] \leftarrow (PM_\Gamma[n,j,0])^{k'}$
      $GM_\Gamma[n,j,1] \leftarrow (PM_\Gamma[n,j,1])^{k'}$
    end if
    $GM_\Gamma[i,j,0] \leftarrow GM_\Gamma[i,j,0] \oplus K_i^0$
    $GM_\Gamma[i,j,1] \leftarrow GM_\Gamma[i,j,1] \oplus K_i^1$
    $pad_0||pad_1 \leftarrow G(PAD[i,j])$
    $GM_\Gamma[i,j,0] \leftarrow GM_\Gamma[i,j,0] \oplus pad_0$
    $GM_\Gamma[i,j,1] \leftarrow GM_\Gamma[i,j,1] \oplus pad_1$
  end for
end for

3. **Server computes the OT answers $\vec{a}$, and sends $(\vec{a}, GM_\Gamma, PER[1,1], PAD[1, PER[1,1]])$ to client.**

Sending OT Answers and the Garbled Matrix to client

for $1 \leq i \leq n$ **do**
  $a_i \leftarrow \mathrm{A}_{\mathrm{OT}}(pk, q_i, K_i^0, K_i^1)$
end for
Server sends $(\vec{a}, GM_\Gamma, PER[1,1], PAD[1, PER[1,1]])$ to client where $\vec{a} = (a_1, a_2, \ldots, a_n)$.

4. **Client retrieves the keys and computes the final result.**

Computing the Final Output

$state \leftarrow PER[1,1]$
$pad \leftarrow PAD[1, PER[1,1]]$
for $i = 1$ to $n-1$ **do**
  $K_i^{x_i} \leftarrow \mathrm{D}_{\mathrm{OT}}(sk, a_i)$
  $pad_0||pad_1 \leftarrow G(pad)$
  $newstate||newpad \leftarrow K_i^{x_i} \oplus pad_{x_i} \oplus GM_\Gamma[i, state, x_i]$
  $pad \leftarrow newpad$
  $state \leftarrow newstate$
end for
$pad_0||pad_1 \leftarrow G(pad)$
Client outputs $GM_\Gamma[n, state, x_n] \oplus pad_{x_n} \oplus K_n^{x_n}$ as his final output.

It is easy to verify that if both parties behave honestly, the protocol correctly evaluates server's DFA Γ on client's input X. In particular, client has the secret information necessary to decrypt one of the two values in each pair on the transition path for input X (in the garbled DFA matrix). Next, we focus on the proof of security of the protocol and a careful analysis of its efficiency.

4.3 Security Proof

We show that as long the oblivious transfer protocol used is secure, so is our protocol. Particularly, if the OT is secure against malicious (semi-honest) adversaries when executed in parallel, our oblivious DFA evaluation protocol described above is also secure against malicious (semi-honest) adversaries. The following Theorem formalizes this statement.

Theorem 1. *In the OT-hybrid model, and given a computationally secure PRG G, the above protocol is* fully-secure *against a malicious client (see definition 1 of Section 2.4) and is* private *against a malicious server (definition 2 of Section 2.4).*

Proof. In this proof we show full-security against a malicious client and privacy against a malicious server.

Full-security against a malicious client. Our proof follows the ideal/real world simulation paradigm. In particular, for any PPT adversary $\mathcal{B}$ controlling client in the real world, we describe a simulator $\mathcal{S}_{\mathcal{B}}$ who simulates $\mathcal{B}$'s view in the ideal world.

Simulation. $\mathcal{S}_{\mathcal{B}}$ runs $\mathcal{B}$ on inputs $X, k, |Q|$. Since we operate in the OT hybrid model, $\mathcal{B}$ sends an input $X' = x'_1 \dots x'_n$ to the OT's trusted party. $\mathcal{S}_{\mathcal{B}}$ generates n random key pairs (K_i^0, K_i^1) for $1 \leq i \leq n$ and sends $K_i^{x_i}$ back to $\mathcal{B}$. $\mathcal{S}_{\mathcal{B}}$ also sends X' to the trusted party and gets $\Gamma(X')$ back.

Let $k' = k + \log|Q|$. $\mathcal{S}_{\mathcal{B}}$ generates a Garbled DFA matrix GM' as follows.

$state \xleftarrow{\$} \{1...|Q|\}$
$pad \leftarrow \{0,1\}^k$
$firststate \leftarrow state$
$firstpad \leftarrow pad$
for each row $i = 1$ to n **do**
  $nextstate \xleftarrow{\$} \{1...|Q|\}$
  $nextpad \xleftarrow{\$} \{0,1\}^k$
  $pad_0||pad_1 \leftarrow G(pad)$
  for each $j \in Q$ **do**
    $GM'[i,j,0] \xleftarrow{\$} \{0,1\}^{k'}$
    $GM'[i,j,1] \xleftarrow{\$} \{0,1\}^{k'}$
    if ($j = state$ and $1 < i \leq n-1$) **then**
      $GM'[i,j,x'_i] \leftarrow (nextstate||nextpad) \oplus pad_{x'_i} \oplus K_i^{x'_i}$
    end if
    if ($j = state$ and $i = n$) **then**
      $GM'[n,j,x'_n] \leftarrow (\Gamma(X'))^{k'} \oplus pad_{x'_n} \oplus K_i^{x'_i}$
    end if
  end for
  $state \leftarrow nextstate$
  $pad \leftarrow nextpad$
end for

Intuitively, all the pair values in GM' are set to random k'-bit strings except for the values on the transition path for $\mathcal{B}$'s input X'. S then sends $(GM', firststate, firstpad)$ to $\mathcal{B}$. Note that $\mathcal{S}_{\mathcal{B}}$ did not need the description of Γ to generate the above garbled DFA matrix. Also note that since we work in the OT hybrid model, $\mathcal{S}_{\mathcal{B}}$ does not need to send the OT answers to $\mathcal{B}$. $\mathcal{S}_{\mathcal{B}}$ outputs whatever $\mathcal{B}$ does and halts. This ends the description of the simulation.

Indistinguishability of views. We now prove that $\mathcal{B}$ cannot distinguish between his view during the real execution and his interaction with $\mathcal{S}_{\mathcal{B}}$.

To prove this we consider a sequence of distributions $D_0, \cdots, D_n$ where D_0 is $\mathcal{B}$'s view in the real execution while D_n is his view during his interaction with $\mathcal{S}_\mathcal{B}$. Our goal is to show that $D_0 \stackrel{c}{\equiv} D_n$. We do so through a simple hybrid argument where we show $D_t \stackrel{c}{\equiv} D_{t+1}$ for $1 \leq t < n$.

1. D_0 is generated using the real execution of the protocol (i.e. $\mathcal{B}$'s view in the protocol).
2. For D_t, the garbled DFA matrix is generated in the same way as D_{t-1} except that we place uniformly random strings for all values in row t except for the single value (one of the two values in a pair) that appears on X's transit path.

Based on the above, it is easy to see that $\mathcal{B}$'s view when interacting with $\mathcal{S}_\mathcal{B}$ is the same as D_n. Also note that due to the security of the PRG-based one-time encryption scheme we use in our construction (i.e. XORing the output of PRG with the message), it is not hard to argue that $D_t \stackrel{c}{\equiv} D_{t-1}$. The only requirement we need to satisfy for this to be true is for the seed to the PRG to be uniformly random and independent of the other seeds. But note that for distribution D_t, all the values in the previous rows $1, \cdots, t-1$ (except for the one value on the transition path) are replaced by uniformly random values and hence the seeds to the PRG invocations at level t are all uniformly random.

The above argument completes our argument that $D_n \stackrel{c}{\equiv} D_0$.

Privacy against a malicious server. Since the only message client sends to server during the protocol execution is his OT queries, client's privacy against a malicious server readily follows from the assumed privacy of the OT protocol used. The proof is automatic when we work in the OT hybrid model. □

4.4 Using OT Extension

In our protocol, the main computational overhead for the client is the $O(n)$ exponentiations required for invoking $n \times OT_2^1$. However, using the extended OT protocol of [11] we can reduce the number of exponentiations from $O(n)$ to $O(k)$ at the expense of security in the *random oracle model*. This improvement is significant in those applications of oblivious DFA evaluation where $n \gg k$. This is particularly the case in the secure pattern matching applications where n represents the size of the text being searched which is often rather large. Using the OT extension also leads to a slight increase in the number of transferred messages (from 2 to 3). In other words, the number of rounds increase from 1 to 1.5.

4.5 A Different Presentation of Our Protocol

An alternative presentation of our construction is to describe it as a generalization of Yao's garbled circuit protocol, where the gates to the circuit can take non-boolean inputs, and return non-boolean outputs. This presentation was pointed out to us by one of the reviewers of our paper at CT-RSA 2012.

More specifically, one can evaluate a DFA D with Q states on a (boolean) input string $x = x_1 ... x_n$ by repeatedly evaluating a "gate" g that takes as input the current state q_i after reading the first i bits of x (so q_0 is just the initial state) and x_i and outputs the next state q_{i+1}. After n applications of the gate g, we obtain the final state q_n of the DFA (explicitly, $q_n = g(g(...g(q_0, x_1), ...), x_n)$), and then we add one more gate to check whether q_n is an accepting state or not. One can generalize Yao's garbled circuit construction to handle such non-boolean gates. In particular, each gate is garbled by constructing $2Q$ ciphertexts, two for each row. Similar to Yao's protocol, each ciphertext is a "double-key" encryption where one of the keys determines x_i's value and the other determines the input state q_i (in each gate g, a unique key is assigned to each state). Each ciphertext encrypts the key for the next state which is determined using the transition function. Hence, each garbled gate g contains $O(|Q|)$ ciphertexts, and requires $O(|Q|)$ symmetric-key operations by the server to compute. Note that the ciphertexts also need to embed the (after permutation) index of the next row of ciphertexts to consider in the upcoming gate. With this approach, the circuit evaluator only needs to perform $O(1)$ symmetric-key operations per gate to decrypt the output key for each gate.

4.6 Efficiency

In this section we present the complexity analysis of our basic protocol.

Table 1: A Comparison of Complexities

	Round Complexity	client Computations		server Computations		Communication Complexity
		Asymmetric	Symmetric	Asymmetric	Symmetric	
Troncoso [21]	$O(n)$	$O(n\|Q\|)$	None	$O(n\|Q\|)$	$O(n\|Q\|)$	$O(n\|Q\|k)$
Frikken [3]	2	$O(n+\|Q\|)$	$O(n\|Q\|)$	$O(n+\|Q\|)$	$O(n\|Q\|)$	$O(n\|Q\|k)$
Gennaro [4]	$min(O(\|Q\|), O(n))$	$O(n\|Q\|)$	None	$O(n\|Q\|)$	None	$O(n\|Q\|k)$
Yao's protocol [22]	1	$O(n)$	$O(n\|Q\|\log\|Q\|)$	$O(n)$	$O(n\|Q\|\log\|Q\|)$	$O(n\|Q\|k)$
Ishai [12]	1	$O(n)$	None	$O(n\|Q\|)$	None	$O(kn^2)$
Protocol 1(PRG)	1	$O(n)$	$O(n)$	$O(n)$	$O(n\|Q\|)$	$O(n\|Q\|k)$
Protocol 1 (PRG+Extended OT)	1.5	$O(k)$	$O(n)$	$O(k)$	$O(n\|Q\|)$	$O(n\|Q\|k)$

Rounds of Communication: Our protocol runs in one round which consists of a message from client to server and vice versa.

Asymmetric Computation: We have tried to minimize the number of required asymmetric computation in our protocol since asymmetric operations are significantly more expensive. The only asymmetric computation we perform in our protocol is for the OTs. Since each OT requires a constant number of exponentiations and there are n invocations of such OTs, the overall number of exponentiation in our protocol is bounded by $O(n)$ for both server and client. Using the amortized OT protocol of Naor and Pinkas [18], server and client have to perform one and two exponentiations per OT, respectively. Our use of OT extension further reduces this bound to $O(k)$, where k is the security parameter.

Symmetric Computation: The only symmetric computation in our protocol is the PRG invocations. Server performs $2n|Q|$ PRG invocations to build GM_Γ, so the symmetric computation for the server is $O(n|Q|)$. Client performs n PRG invocations for computing the final output and hence the number of symmetric operations by the client is only $O(n)$.

Communication Complexity: The communication complexity of the protocol is dominated by the number of bits stored in the garbled DFA matrix GM_Γ which is bounded by $O(n|Q|k)$ where k is the security parameter.

Comparison to Previous Work: Table 1 summarizes and compares the computational and communication costs of our proposed protocol with the related work. The complexity for a Yao's-based construction is borrowed from the analysis given in [4]. The complexity of the ODFA protocol based on the construction of [12] is derived by considering a branching program corresponding to evaluation of a input of size n on a DFA of size Q. Note that in all the existing constructions except for the one base on Yao's garbled circuit protocol, the number of asymmetric operations (exponentiations) by the server is at least linear in both the input size n and the DFA size Q. In our protocol, on the other hand, this number is $O(n)$ in the standard model and $O(k)$ in the random oracle model. This is a significant improvement in efficiency when dealing with large DFA sizes. Another efficiency criteria we are interested in is small computation by the input holder (client). In all the previous constructions except for the one based on [12], the client's work is at least linear in the DFA size which is undesirable in applications with large DFAs.

5 Counting Accepting States and Secure Pattern Matching

Modified versions of our proposed protocol for Oblivious DFA evaluation can be used to efficiently solve multiple variants of the *Secure Pattern Matching* problem. This problem has been the focus of several recent works (e.g. see [4, 8, 13]). In this section we use the notion of Alice/Bob in which Alice has the role of the server and Bob has the role of the client in our protocol. This notion helps us to better explain the secure pattern matching application. In the three main variants we consider here, one party (Bob) holds a text T while the other party (Alice) holds a pattern p and the aim is for Alice to learn one of the following: (i) whether or not p appears in T, (ii) all the locations (if any) where p occurs as a pattern in T, or (iii) the number of occurrences of a pattern p in T, while Bob learns nothing about the pattern.

In this section we discuss how to adapt our oblivious DFA evaluation protocol in order to solve all three variants without any additional computational overhead. The first two variants can be instantiated through a relatively straightforward application of our ODFA protocol from Section 4. Nevertheless, a few small

modifications and considerations are necessary to make things work and we discuss them in this section.

The more interesting and challenging problem to tackle is the third and last variant of secure pattern matching where parties are interested in counting the number of occurrences of the pattern in a text but nothing else. Counting the number of occurrences is a natural measure of how related or essential a pattern is to a studied text. While solving the second variant of the problem would also provide the number of occurrences of the pattern, it reveals significantly more information than just the count. Hence, if the number of occurrences is all that the parties are interested in, a solution for the second variant is not a suitable solution.

It is not clear how to modify existing secure pattern matching protocols to solve the third variant without a significant increase in their computation. It is also not clear how to formulate this problem as an oblivious DFA evaluation protocol and then apply our construction from Section 4 to it. We observe that what is really needed to solve this variant of the secure pattern matching problem, is a modified oblivious DFA evaluation protocol that counts the number of accepting states visited during an input evaluation and outputs this count as the final result as opposed to a single bit indicating whether the final state was an accepting or a rejecting one. This modified version of the ODFA protocol, when applied to the pattern-specific DFA of KMP [14], yields exactly a secure protocol for the third variant of the pattern matching problem. Our solution for this variant uses a novel trick (see Section 5.3 for details) that allows Alice to learn the number of occurrences of the pattern p without having to perform any additional computation.

Below, we describe how to efficiently modify our oblivious DFA protocol in order to implement the three variants of the secure pattern matching discussed above.

5.1 First Variant: Existence of p in T

In this simplest variant of pattern matching, Alice only learns whether her pattern p exists in T or not while Bob learns nothing. To solve this problem using the protocol of Section 4, Alice first converts her pattern p to a DFA Γ that accepts an input T if it contains the pattern p (using well-known transformations [20]). Bob's input is his string T. If we wanted Bob to learn the result of computation we would simply run our protocol to evaluate Γ on X without any modifications. But, since we only want Alice to learn the result, we modify the original protocol as follows:

Modifications. When generating the *last row* of the DFA matrix M_Γ (before it is garbled or permuted), instead of filling the non-accepting cells with 0 and accepting ones with 1, Alice generates two uniformly random strings $rand_0$ and $rand_1$ of length k and uses these random strings for accepting and non-accepting states, respectively (instead of bits 0 and 1). The DFA matrix is then garbled as before. Note that with this modification, Bob learns one of the two random strings $rand_0$, or $rand_1$ instead of the actual result of the computation. Bob send this value to Alice who can translate the random string to its actual value. This modification only adds one extra message to the protocol (half a round) and does not effect either parties' computation.

Security. It is easy to show that the resulting protocol is still fully-secure against a malicious Bob and private against a malicious Alice. Proofs of security are closely related to those for our protocol of Section 4 and hence omitted.

5.2 Second Variant: All Locations of p in T

In this version of the problem, which is also the most common variant studied in the previous work, Alice needs to learn all the locations in T where p appears. As before, Alice first converts the pattern p to a DFA Γ and Bob's input is his string T. However, the DFA we need is slightly different from that of the first variant since we want to compute all the locations where p appears in T. In particular, during the evaluation of T we need the property that we visit an accepting state, whenever a pattern p occurs in T. The KMP transformation [14] yields exactly such a DFA.

Modifications. Alice then selects two uniformly random strings $rand_i^0$ and $rand_i^1$ for each row $1 \leq i \leq n$ of M_Γ. For each row of M_Γ, Alice concatenates the values in each cell with $rand_i^1$ ($rand_i^0$) if the corresponding state is an accepting state (non-accepting state). M_Γ is then garbled as usual. While evaluating

the garbled DFA matrix, Bob learns a vector of random strings (with $rand_i^b$s as its components) corresponding to the transit path he traverses. Bob sends this vector back to Alice who translates the random vector to a corresponding bit vector (mapping $rand_i^1$ to 1 and $rand_i^0$ to 0). The location of 1s in this bit-vector are exactly the locations of p in T, and the output we are after.

Efficiency comparison. This modification only requires an extra message from Bob to Alice and does not affect the complexity of our protocol. Assuming simple string patterns, protocol of [4] requires $O(|p||T|)$ exponentiations, the protocol of [8] requires $O(|p| + |T|)$ exponentiations while our solution only requires $O(|T|)$ such operations in the standard model and $O(k)$ exponentiations in random oracle model. This is a significant improvement specially when $|T| \gg k$, which is likely to be the case for real inputs to the pattern matching problem.

5.3 Third Variant: Number of Locations of p in T

Now consider the more challenging variant where the goal is to only reveal the number of occurrences to Alice or Bob but nothing else. First consider the case where Bob is to learn the number of matches while Alice learns nothing. The pattern-specific DFA we need is again generated using the KMP algorithm [14]. The main observation is that for the KMP-transformed DFA, the number of accepting states visited in one evaluation of a text T, is exactly the number of times a pattern p occurs in a text T. Hence, all we need to do is to design a protocol for counting the the number of accepting states visited during a DFA evaluation of the input. Such an oblivious DFA protocol might find other applications in future.

Modifications. Alice generates n uniformly random values $s_i \in F$ for $1 \leq i \leq n$ where F is a finite field of size $|F| > |T|$. Alice then computes $S = \sum_{i=0}^{n} s_i$. When generating the DFA matrix, for each row i of M_Γ, Alice concatenates the values in each cell by s_i except for the cells corresponding to accepting states for which the value $s_i + 1$ is concatenated instead. The DFA matrix is then garbled as usual. Alice sends S along with the garbled DFA matrix to Bob. When computing the final output, Bob collects all the values $s'_i \in F$ for $1 \leq i \leq n$ on his transit path. Finally, Bob computes the sum of those values ($S' = \sum_{i=0}^{n} s'_i$), and outputs $S' - S$ as the number of occurrences of p in T.

Now if we only want Alice to learn the result, we do not send the value S to Bob. Instead, in the above protocol when Bob calculates S', he sends it back to Alice who computes $S' - S$ on his own in order to learn the number of matches.

Correctness. The intuition behind the correctness of the algorithm is that for each location i where p appears in T, the value $s_i + 1$ is retrieved by Bob and for all other locations the value s_i. Hence, the number of additional 1s is exactly equal to the number of locations of p in T.

Security. In order to prove the security of the scheme, we need to show that Bob cannot distinguish between s_i and $s_i + 1$ values he retrieves since they both are uniformly random values in F. In other words, we need to show that his view is simulatable given just the final output.

The proof of security in this case is slightly more subtle, since it does not automatically follow from our original ODFA protocol. Hence, we outline the intuition behind it next. Our main observation is that the following two distributions (D_V and D'_V) are *identically* distributed.

Let V be an arbitrary subset of size t of $\{1, \ldots, n\}$. Here, V represents the locations in T where matches occur. Consider the following two distributions:

1. (D_V) Choose n uniformly random values $\{s_1, \ldots\ s_n\} \in F$. Compute $S = \sum s_i$. For every i in V, let $s'_i = s_i + 1$. For the rest, let $s'_i = s_i$. Output $(s'_1, ..., s'_n)$.

2. (D'_V) Choose a uniformly random value S in F. Let $S' = S + t$. Generate $n-1$ random values $s'_1, ..., s'_{n-1}$. Let $s'_n = S' - \sum_{i=0}^{n-1} s'_i$. Output $(s'_1, ..., s'_n)$.

It is relatively easy to show that the above two distributions D_V and D'_V are identical for any subset $V \subset \{1, \ldots, n\}$ of size t. Given this property, we can modify the original proof of security for our oblivious DFA evaluation protocol such that the simulator in the proof of Theorem 1 samples from the second distribution (D'_V) while the first distribution (D_V) represents the distribution of the corrupted party's view in the real world execution of the protocol. Since sampling from D'_V only requires knowledge of t (i.e. the number

of occurrences of p in T), our simulator can simulate the real world adversaries's view given only the final output. A complete proof of security for the above protocol closely follows the proof of Theorem 1, and hence is omitted.

6 Implementation and Experimental Results

To demonstrate that our proposed protocol as well as its variants are practical, we have implemented and evaluated our protocol. Implementation is done using C++ and the methods in Crypto++ library v.5.61. The experiments were run on two machines one as the client (input holder) and the other as the server (DFA holder). Each of these systems has an Intel Core i7 processor, with 4GBs of RAM. They systems are connected using a 1 GB ethernet.

6.1 OT Implementation

For our OT protocol, we have implemented the Naor-Pinkas amortized OT (See section 3.1 of [18]) which requires one exponentiation for each transfer. We implemented their protocol over Elliptic Curves (EC) for better efficiency. The EC curve we use is the NIST recommended curve P-192 (see Section D.1.2.1 of [2]).

We have also implemented the OT extensions of [11] for improved efficiency. Two extensions are discussed in [11]. The first one is concerned with extending the number of OTs efficiently (Section 3) while the second extension one (Appendix B [11]) reduces oblivious transfer for long strings to oblivious transfer of shorter strings.

Both extensions mentioned above rely on the use of a hash function (in the random oracle model). We have chosen SHA-256 for this implementation. When the number of OT invocations is lower than $k = 80$, we make a direct call to our base OT protocol, but otherwise employ the first extension to reduce the number of OT invocations. When we encounter an OT with message sizes larger than 256 bits (equivalently 32 bytes) we reduce them to an OT with message size of 256 bits using the second extension. Note that since in this protocol the message for each OT is XORed with the output of the random oracle (hash function), we are able to handle varying message sizes for each OT by simply adjusting the output size of the random oracle to the corresponding message size.

The PRG is also implemented using sufficient invocations of the random oracle (i.e. SHA-256).

Table 2: Empirical Results for Experiment 1 (ms)

n	Client Ungarbling	Client OT	Server Garbling	Server OT	Client Total	Server Total	Communication (MB)
100	0.13	67.10	6.85	127.60	67.23	134.44	0.06
500	0.61	69.65	34.04	130.01	70.26	164.05	0.28
1500	1.92	74.40	102.22	137.20	76.33	239.42	0.83
5000	6.32	92.05	340.10	159.89	98.37	499.99	2.72
10000	12.66	118.36	674.59	191.05	131.02	865.64	5.44
20000	25.43	166.50	1352.59	254.48	191.93	1607.07	10.88
50000	64.35	323.48	3409.71	448.51	387.83	3858.22	27.19
75000	96.44	451.53	5056.24	610.65	547.97	5666.88	40.78
100000	128.89	576.53	6794.88	782.05	705.43	7576.93	54.38
150000	194.66	837.60	10244.40	1087.57	1032.26	11331.90	81.55

6.2 Experiments

We have designed two experiments to analyze the effect of the input size and the DFA size on the performance of our protocol. In the first experiment we fix the DFA size and increase the input size, while in the second experiment we fix the input size and increase the DFA size. In what appears next, we refer to the input holder as the client while referring to the DFA holder as the server.

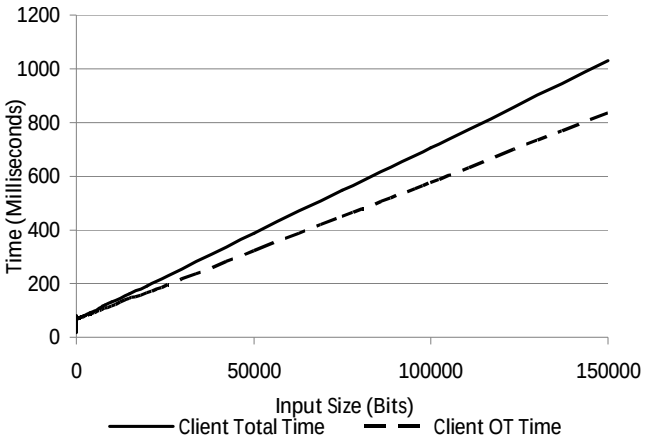


Figure 2: Client computation for experiment 1

Table 3: Empirical Results for Experiment 2 (ms)

Q	Client Ungarbling	Client OT	Server Garbling	Server OT	Client Total	Server Total	Communication (MB)
100	0.02	32.14	6.89	17.76	32.17	24.65	0.05
500	0.02	31.80	33.37	18.54	31.82	51.91	0.27
1500	0.02	31.90	99.74	18.26	31.92	118.00	0.80
5000	0.03	32.36	331.47	18.44	32.39	349.91	2.67
10000	0.03	31.86	666.18	18.51	31.89	684.69	5.34
20000	0.03	31.56	1332.67	18.39	31.59	1351.06	10.68
50000	0.03	32.12	3373.53	17.92	32.14	3391.45	26.71
75000	0.03	32.22	5169.60	18.88	32.24	5188.49	42.92
100000	0.03	32.57	6949.88	18.56	32.60	6968.43	57.22
150000	0.03	32.29	10640.30	18.44	32.32	10658.70	85.83

Experiment 1. In the first experiment, an arbitrary DFA with 20 states is considered. We have chosen a low number of states in order to draw a clear conclusion on the effect of the input size. We then increase the input size starting from 10 bits all the way up to 150000 bits. This experiment is of interest for applications such as DNA matching where the input can be large while the number of DFA states (related to the pattern size) is often low. It is noteworthy to mention that by fixitng the state size, the DFA transitions does not have any effect on computation or communication costs and hence we just selected a DFA with arbitrary transitions. The results of this experiment are presented in Figure 2 and Figure 3. more detailed empirical measurements are also presented in Table 2.

From Figure 2, it can be observed that the client time is dominated by the client's OT time, and the client's evaluation (ungarbling) time is almost negligible for even large input sizes. This is due to the fact that the client's ungarbling is limited to only one PRG evaluation per input bit. On the other hand, based on Figure 3, for large input sizes, server time is dominated by the server's garbling time. The reason for this is partly due to our use of OT extension, which prevents the number of exponentiations from increasing as the input size grows. We also note that that for input size of 2000 bits or more, OT time is no longer the bottleneck for the overall protocol. Furthermore, the server's garbling time is dependent on the size of the DFA matrix (unlike client's evaluation time which only depends on the number of rows of the matrix) and hence grows as we increase the input size.

Experiment 2. In the second experiment, for a fixed input size of 20 bits, we produce arbitrary DFAs with increasing number of states (10 to 150000 states). The result of this experiment is presented in Figure 4. More detailed empirical measurements are also presented in Table 3.

Based on Table 3 we note that the server time is dominated by the server garbling time, since the number of OTs remain the same. The OT time for 20-bit inputs is approximately 32 milliseconds for both the client

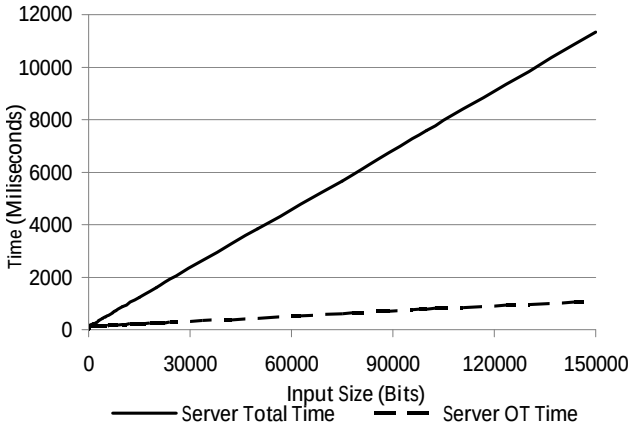


Figure 3: Server computation for experiment 1

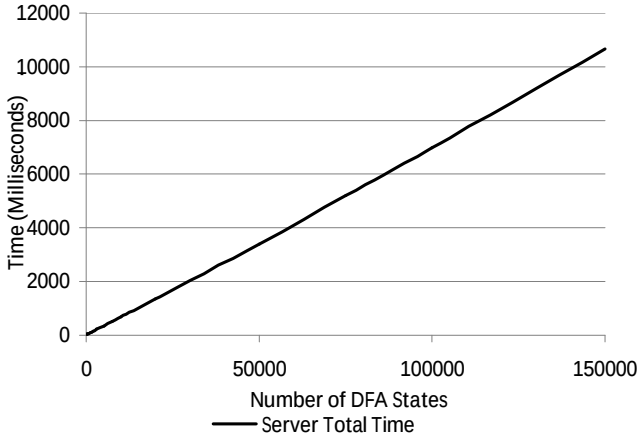


Figure 4: Server computation for experiment 2

and the server. Client's ungarbling time is negligible because it does not depend on the number of states. When the input size is 20, for DFA sizes of over 200, the OT is no longer the bottleneck. Again we have a negligible computation time for the client and a total time (client + server) of under 1 second for DFAs with number of states as large as 15000.

Finally, we note that the communication time only constituted a small portion of the total time in our experiments and hence we only report the size of communication in the tables.

References

[1] R. Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, 2000.

[2] PUB FIPS. 186-3. Digital signature standard (DSS), 2009.

[3] K. Frikken. Practical private DNA string searching and matching through efficient oblivious automata evaluation. *Data and Applications Security XXIII*, pages 81–94, 2009.

[4] R. Gennaro, C. Hazay, and J. Sorensen. Text search protocols with simulation based security. *Public Key Cryptography–PKC 2010*, pages 332–350, 2010.

[5] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st annual ACM symposium on Theory of computing*, pages 169–178. ACM, 2009.

[6] O. Goldreich. *Foundations of cryptography: Basic applications*. Cambridge Univ Pr, 2004.

[7] C. Hazay and Y. Lindell. Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. *Theory of Cryptography*, pages 155–175, 2008.

[8] C. Hazay and T. Toft. Computationally secure pattern matching in the presence of malicious adversaries. *Advances in Cryptology-ASIACRYPT 2010*, pages 195–212, 2010.

[9] Wilko Henecka, Stefan K ögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. Tasty: tool for automating secure two-party computations. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 451–462, New York, NY, USA, 2010. ACM.

[10] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security Symposium*, 2011.

[11] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending oblivious transfers efficiently. *Advances in Cryptology-CRYPTO 2003*, pages 145–161, 2003.

[12] Y. Ishai and A. Paskin. Evaluating branching programs on encrypted data. *Theory of Cryptography*, pages 575–594, 2007.

[13] J. Katz and L. Malka. Secure text processing with applications to private DNA matching. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 485–492. ACM, 2010.

[14] D.E. Knuth, J.H. Morris Jr, and V.R. Pratt. Fast pattern matching in strings. *SIAM journal on computing*, 6:323, 1977.

[15] Y. Lindell and B. Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. pages 52–78. Springer, 2007.

[16] Y. Lindell and B. Pinkas. A proof of Yao's protocol for secure two-party computation. *Journal of Cryptology*, 22(2):161–188, 2009.

[17] H. Lipmaa. An oblivious transfer protocol with log-squared communication. In *Proceedings of the 8th Information Security Conference (ISC' 05)*, volume 3650, pages 314–328. Springer, 2005.

[18] Moni Naor and Benny Pinkas. Efficient oblivious transfer protocols. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, SODA '01, pages 448–457, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics.

[19] C. Peikert, V. Vaikuntanathan, and B. Waters. A framework for efficient and composable oblivious transfer. *Advances in Cryptology–CRYPTO 2008*, pages 554–571, 2008.

[20] M. Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1996.

[21] 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*, pages 519–528. ACM, 2007.

[22] A.C. Yao. Protocols for secure computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, pages 160–164. Citeseer, 1982.