ZIDS - A Privacy-Preserving Intrusion Detection System using Secure Two-Party Computation Protocols

Salman Niksefat[1], Babak Sadeghiyan[1], Payman Mohassel[2] and Saeed Sadeghian[2]

[1]*Computer Engineering and Information Technology Department, Amirkabir University of Technology, Tehran, Iran*
[2]*Computer Science Department, University of Calgary, Calgary, AB, T2N1N4, Canada*
Email: basadegh@aut.ac.ir

We introduce ZIDS, a client-server solution for private detection of intrusions that is suitable for private detection of zero-day attacks in input data. The system includes an IDS server that has a set of sensitive signatures for zero-day attacks and IDS clients that possess some sensitive data (e.g. files, logs). Using ZIDS, each IDS client learns whether its input data matches any of the zero-day signatures, but neither party learns any additional information. In other words, the IDS client learns nothing about the zero-day signatures and the IDS server learns nothing about the input data and the analysis results.
To solve this problem, we reduce privacy-preserving intrusion detection to an instance of secure two-party *oblivious deterministic finite automata evaluation* (ODFA). Then, motivated by the fact that the DFAs associated with attack signature are often *sparse*, we propose a new and efficient ODFA protocol that takes advantage of this sparsity. Our new construction is considerably more efficient than the existing solutions and at the same time does not leak any sensitive information about the nature of the sparsity in the private DFA.
We provide a full implementation of our privacy-preserving system which includes optimizations that lead to better memory usage and evaluate its performance on rule sets from the Snort IDS.

Keywords: Secure Two-Party Computation; Oblivious DFA Evaluation; Privacy-Preserving Intrusion Detection; Zero-day Signatures

1. INTRODUCTION

Vulnerabilities are typically found and reported by "white hat" hackers through a range of techniques such as fuzz testing, reverse engineering or source code analysis. Some organizations are even willing to offer financial incentives in return for new vulnerabilities. An example of a program encouraging hackers to report their bugs is the TippingPoint's Zero Day Initiative (ZDI) [1], which pays for new vulnerabilities discovered in existing systems. Other examples include Mozilla's Security Bug Bounty Program [2], or a similar program by Google for security of its web applications [3].

New (zero-day) vulnerabilities are often discovered months before an attack outbreak. But, due to the complexity of the existing software products, or simply due to bad security practices, designing the appropriate patches and incorporating them into the existing systems takes a long time. In fact, patch development times can vary from a few days to several years in worst cases, and several months of delay between the time a vulnerability is reported and when a patch becomes available to the users seems to be a reasonable estimate (e.g. see the following statistics from Trapkit [4]).

Hence, it is critical for security vendors and service providers to closely watch the development of new vulnerabilities and to provide their customers with the necessary protection in a timely manner. Commonly, once a new vulnerability is discovered and reported, security vendors generate and distribute the relevant filters (signatures) to their customers through the deployed intrusion prevention and detection systems (IPS/IDS).

Privacy of Signatures. There are strong reasons for keeping IDS signatures private. Firstly, the signatures can be analyzed by attackers who could potentially use them to learn the vulnerabilities and design their own exploits. This is specially the case for signatures of zero-day exploits. In other words, the IPS/IDS devices can become a source of information for attackers who wish to learn and design new attacks. The following quote from ZDI's disclosure policy demonstrates that security experts are well aware of the sensitivity of such information:

> "TippingPoint may share technical details of the vulnerability with other security vendors ... Such a security vendor must show they are able to provide security protection for vulnerabilities, while at the same time not revealing the technical vulnerability details in their product updates."

Secondly, many security vendors consider their protection mechanisms *proprietary* and are not willing to make it accessible to their competitors. However, ensuring that the product updates do not reveal the sensitive details of the vulnerabilities and/or hide the information considered proprietary by the vendors can be a challenging task in the traditional setting.

Zero-Day Detection as a Remote Service. A natural solution is to provide a customized intrusion protection/detection service for zero-day vulnerabilities, through a *remote server* that allows the security vendors to keep the sensitive information on their side and hide the details of the vulnerabilities from their users.

Privacy of Client's Data. On the other hand, a remote solution would require the users to share their suspicious data with the service provider, which raises privacy concerns of its own. Users may not be willing to share their input with the IPS/IDS providers since this data could include private network activities, URLs, personal files and emails. The client side privacy concerns are valid even if the IDS device is not managed remotely, since there is the possibility of exporting users' data by untrusted IDS devices.

Privacy of the Analysis Results. Besides the privacy of client's input data, the outcome of the intrusion analysis itself can be considered sensitive information, specially for zero-day vulnerabilities. Many companies prefer not to reveal the specifics of the vulnerabilities present in their systems since such information could be misused to penetrate and attack the company's internal infrastructure. Hence, a desirable feature of such a solution for clients is to keep the outcome of the analysis hidden even from the service provider.

In this paper, we aim for a solution that meets the privacy needs of both the service providers and their clients. We emphasize that, not surprisingly, a solution with such a strong privacy guarantee comes with its own costs and overheads, and hence should be utilized only in cases when privacy concerns necessitate. In particular, we believe that such a solution should only be considered as a special service and be applied to a small subset of sensitive signatures (e.g. associated with zero-day attacks), while still allowing the rest of the IDS service to operate in the normal fashion.

1.1. The Problem Statement

Motivated by the above discussion, we consider the following problem. An IDS server who holds a set of signatures (rule set), and an IDS client who wants to check his data against this rule set. The goal is for the IDS client to receive the appropriate alert if his input data is infected or anomalous, but nothing else. The IDS server should not learn any information about the client's input data (See figure 1).

The security and efficiency requirement are as follows.

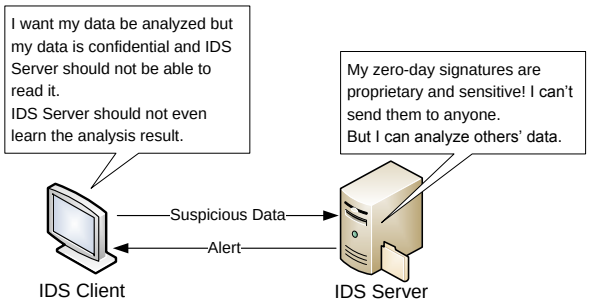

FIGURE 1. The basic architecture of ZIDS

Security. We borrow our security requirements from *secure two-party computation* and require the protocols to be secure against malicious clients and servers who may deviate from the steps of the protocol. In essence, these requirements guarantee that any information that a malicious IDS client can learn in the privacy preserving IDS protocol is simulatable given only the final alert (full security against malicious clients). It also implies that a malicious server's view in the protocol is indistinguishable for any two inputs held by the client (privacy against a malicious server).

Efficiency. Since the size of the server's rule set turns out to be the main contributor to the complexity of the protocol, our two main efficiency items are: (i) the client's computation be small and independent of the size of the server's rule set, and (ii) the total online computation of both parties combined be small. Finally, to keep the protocols non-interactive, we aim for a small round complexity and ideally, only a *single round*.

General-purpose solutions. The above problem can be formulated as an instance of secure two-party computation. When solving any problem in secure two-party computation, an important point of comparison is a generic construction based on Yao's garbled circuit protocol [5]. In recent years it has been demonstrated [6, 7, 8] that secure two-party protocols based on Yao's garbled circuit construction may be practical when the function being computed can be expressed with a *reasonably sized* boolean circuit. As we will see in section 6, however, this is not the case for our IDS applications where the circuits can be very large. Yao's garbled circuit protocol has the additional drawback of requiring both the client and the server to perform the same amount of computation (i.e. proportional to the size of the circuit they evaluate) which renders it unsuitable for client-server applications.

1.2. Our Contributions

Transforming the Problem to Oblivious DFA Evaluation. Our first contribution is to reduce the privacy-preserving IDS problem introduced above to an instance of the oblivious DFA evaluation problem (ODFA). In an ODFA protocol, one party holds an input string X, while the other party holds a DFA Γ. Their goal is for the input holder to learn $\Gamma(X)$ and for the DFA holder to learn nothing. Recently, several constructions of ODFA protocols were introduced [9, 10, 11, 12] but their motivating application revolves around the string/pattern matching problem and its use in processing DNA data. To the best or our knowledge, our work is the first to consider the use of ODFA protocols in privacy-preserving IDS applications.

As will be shown in section 3, our privacy-preserving IDS construction can be naturally divided into an *offline* and an *online* phase. In the offline phase, the client's input has not entered the system yet, but the IDS's rule set is known. In the online phase, however, a specific client input is analyzed for intrusion. The online phase turns out to be extremely efficient in our construction, for both the server and the client. We elaborate on the different components of the construction next.

From Attack Signatures to DFAs. We propose techniques for converting each rule in an IDS's rule set into a DFA. In other words, we generate a DFA that transits to an accepting state if the attack patterns corresponding to the rule exists in the input string and transits to a non-accepting state otherwise. As we will discuss in section 4, this is done by chaining the content processing elements associated with each rule in a serial and/or parallel manner.

The generated DFAs are then fed to a *DFA Combiner module* that combines them into a single DFA that has multiple accepting states, each labeled with a specific attack pattern's ID. The alternative of invoking the ODFA protocol separately for each attack pattern has two main drawbacks: (i) it leaks additional information since the client learns a different output each time, and (ii) it is less efficient, since the DFA combiner algorithm yield DFAs with much smaller sizes than the sum of the sizes of each input DFA.

Finally, we reduce the number of states of the DFA via *DFA-minimization* techniques. This turns out to yield a significant reduction in computation and communication, since the efficiency of the ODFA protocol is directly dependent on the DFA size.

A new ODFA Protocol for Sparse DFAs. Our empirical studies show that the DFAs generated via combining multiple attack patterns are quite sparse. See section 5.1.2, and Table 2 for the sparsity parameters we extracted for DFAs generated for various number of Snort rules. Our experiments show that the maximum number of outgoing edges in the resulting DFAs is a factor of 4-9 smaller than the alphabet size. We design a new ODFA protocol that takes advantage of this reduction in the outdegree of the states in the DFAs. To show the efficiency of this new construction, we first extend the protocol of [12] (originally designed for binary alphabets) to work with ASCII characters. A naive extension of their construction results in a factor of 128 increase in both the computation and the communication. We then propose the new protocol that takes advantage of DFA's sparsity. The resulting protocol is considerably (a factor of 3-9) more efficient both in terms of communication and computation compared to the naive extension, and at the same time, does not leak any useful information about the nature of the sparsity in the DFA.

Implementation and Evaluation. We fully implement our privacy-preserving IDS system outlined above and evaluate its performance on real data. Our implementation can process arbitrary rule sets from Snort and analyze ASCII strings for potential intrusion. We significantly reduce the memory usage of our ODFA construction during offline phase by a semi-pipelining technique. In particular, every time the server processes and garbles a single row of the DFA matrix and sends it over to the client. As a result, the server only stores a single row at each point in time, therefore memory usage becomes a non-issue in our implementation (a similar technique was used in a recent implementation of Yao's garbled circuit protocol [8]). We report on the performance of our system on different input sizes in section 6.

1.3. Related Work in Privacy Preserving IDSes

To the best of our knowledge, our paper is the first work that proposes a privacy preserving IDS in a two-party setting with such strong privacy requirements. That is,

both the privacy of server sensitive signatures and the client input data should be preserved simultaneously.

Most of the research in the field of privacy preserving intrusion detection consider the preserving of user identities during intrusion analysis [13, 14, 15, 16]. That is the IDS should not identify the identity of users while analyzing their behavior. This is generally done by replacing the user identity fields with pseudonyms. Apparently in these works, the privacy of the IDS signatures or client input data (other than identity fields) are not preserved. There are other works that focus on collaborative and privacy preserving techniques for building signatures [17] rather than private detection.

2. BACKGROUND

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 and κ to denote the computational security parameters for the symmetric and asymmetric encryptions respectively. We use s to denote the statistical security parameter for the OT extension. We denote an element at row i and column j of a matrix by $M[i, j]$. If the element itself is a tuple we use $M[i, j, z]$ to denote z-th value of the tuple. 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. By a^b we mean concatenating a by itself b times. We use λ to denote an empty string. Also let "$\leftarrow$" denote assignment and "$s \xleftarrow{\$} S$" denote the assignment to s of a randomly chosen element of set S.

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. We denote the i-th row of PER by $PER[i]$.

Algorithm 1 $\mathsf{GenPerm}(n, Q)$

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

2.2. DFA

A deterministic finite automaton (DFA) [18] 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 $\mathbb{Z}_{|Q|}$. $\Delta(j, \alpha)$ returns the next state when the DFA is in state $j \in Q$ and sees an input $\alpha \in \Sigma$. A string $X = x_1 x_2 \ldots x_n \in \Sigma^n$ is said to be accepted by Γ if the state $s_n = \Delta(\ldots \Delta(\Delta(s_1, x_1), x_2) \ldots, x_n)$ is a final state $s_n \in F$. A binary DFA is a DFA with $\Sigma = \{0, 1\}$.

2.3. Oblivious Transfer

Our protocols use Oblivious Transfer (OT) as a building block. We mostly focus on protocols that run in a single round. 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 that 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[3] 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 such that $1 \leq i \leq t$. The parameters t and τ are given as inputs to both parties.

The algorithms $\mathrm{G_{OT}}$, $\mathrm{Q_{OT}}$ and $\mathrm{D_{OT}}$ are invoked by the client for the operations of *OT key generation*, *OT query generation* and *OT answer decryption* respectively. The algorithm $\mathrm{A_{OT}}$ is used for *OT answer generation* and is invoked by the server. The protocol proceeds as follows:

1. The client generates a pair of keys, i.e. one public and one private, $(pk, sk) \leftarrow \mathrm{G_{OT}}(1^\kappa)$, computes an OT query $q \leftarrow \mathrm{Q_{OT}}(pk, 1^t, 1^\tau, i)$, and sends (pk, q) to the server.
2. The server computes an OT answer $a \leftarrow \mathrm{A_{OT}}(pk, q, s_1, \ldots, s_t)$ and sends a to the client.
3. The client computes and outputs $\mathrm{D_{OT}}(sk, a)$ which equals to s_i.

When $t = 2$, in case of semi-honest adversaries many of the OT protocols in the literature are one-round protocols (e.g. see [19, 20]). In case of malicious adversaries (CRS model), one can use the one-round OT protocols of [21].

For $t > 2$ we use a construction of Naor and Pinkas [22] to reduce the computation to $\log t$ invocations of OT_1^2 and t symmetric-key operations. We also take advantage of OT extension of [23] to perform many OT_1^2 protocols simultaneously and more efficiently.

2.4. Pseudorandom Generator

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

[3] Probabilistic Polynomial Time

and for any PPT distinguisher D the following should be negligible in k

$$|\Pr[D(U_n) = 1] - \Pr[D(G(U_k)) = 1]|$$

where U_n denotes a uniformly random string in $\{0,1\}^n$. Here the string U_k is called the "seed".

3. SYSTEM OVERVIEW

In this section we present an overview of our proposed privacy preserving IDS system (ZIDS) and the steps performed by both participants. Figure 2 depicts the architecture as well as the different components of the proposed system. IDS client's input is typically a suspicious payload extracted from a log file, or captured from a network. The IDS server's input is a set of attack signatures or patterns of malicious behavior. The aim is for the IDS client to learn whether the suspicious input matches any of the attack signatures. In fact the client learns 0 (No attack) or the ID of the attack pattern found. The IDS server should not learn anything about the client's payload. Moreover, the IDS client should not learn any additional information about the attack signatures. Optionally, the result of the intrusion analysis could only be revealed to the client in which case, the server would not learn whether the input matches any signatures or not.

Please note that as we discussed earlier, the client's output in the ZIDS is the ID of the attack pattern found. However, this could help a client who runs the protocol multiple times on slightly modified inputs to learn additional information about the attack pattern with a specific id. Alternatively, we can also consider the case that the client learns only whether any of the attack patterns matches or not. This can be accomplished by having the client to learn a "1" instead of the ID of the attack pattern.

A Note on the IDS Client's Input. We note that the actual input data processed by IDSes often contains additional metadata, such as user identities and IP headers in case of captured network traffic. The privacy concerns with such metadata have been studied in prior work and can be addressed using well-known techniques such as pseudonymization and omission [13, 16, 14]. Therefore, in this paper we focus on preserving the privacy of the payload data rather than the accompanying metadata.

In particular, the obvious solution to protect privacy within such metadata is that the client and server have an initial interaction in which the client hides such identifying fields by omission and pseudonymization and sends the IP headers to server. Then for additional content processing on payload the client and server proceed with ZIDS. It is noteworthy to mention that the existing techniques fail to address the privacy concerns with payload data since the IDS needs to analyze the payload in order to detect intrusions, and omission or pseudonymization techniques limit the ability of the IDS in performing such analysis.

We divide the steps of the protocol into the *offline* and the *online* phases, and give a high level description of the different components involved in each step.

3.1. The Offline Phase

The offline phase occurs before the client's suspicious input enters the system, and can be precomputed by the server without the knowledge of the client's input. The different components of the offline phase are numbered in the order they are used in the system.

- Parameter Setup (Step 0 - offline): In this step, the required cryptographic keys (for the base OTs) and parameters used by the primitives are exchanged.
- Snort Rule-to-DFA Module (Step 1 - Offline): This component converts the content processing elements of a Snort rule to an equivalent DFA. Each DFA transits to an accepting state if the corresponding attack pattern is present in the input string and will transit to a non-accepting state, otherwise. We describe the details of the Rule-to-DFA module in section 4.1.
- DFA Combiner Module (Step 2 - Offline): This component takes as input a number of DFAs and combines them into a single DFA. This step is necessary to prevent multiple invocation of the ODFA protocol, one for each DFA. As a result, a single DFA that represents all attack signatures will be evaluated obliviously by the IDS client and server. The resulting DFA has a number of accepting states, each labeled with a specific attack pattern ID. More details on combining multiple DFAs into a single DFA are discussed in section 4.2.
- DFA Optimizer Module (Step 3 - Offline): This component is responsible for optimizing the resulting DFA of step 2 by reducing its size and extracting the sparsity parameters. This is quite important since both the computation and the communication of the ODFA protocol are directly affected by it. We discuss the details of the DFA minimization and sparsity analysis in sections 4.3 and 5.1 respectively.
- Key Generator Module (Step 4 - Offline): This component generates the random keys used in the garbled DFA builder component. The keys are also used as input strings in the OT response builder.
- GDFA Builder (Step 5 - Offline) This component builds a garbled DFA according to the description of the oblivious DFA evaluation protocol discussed in section 5.

3.2. The Online Phase

These components are performed only after the client's suspicious input enters the system. Note that before starting the online phases some additional parameters

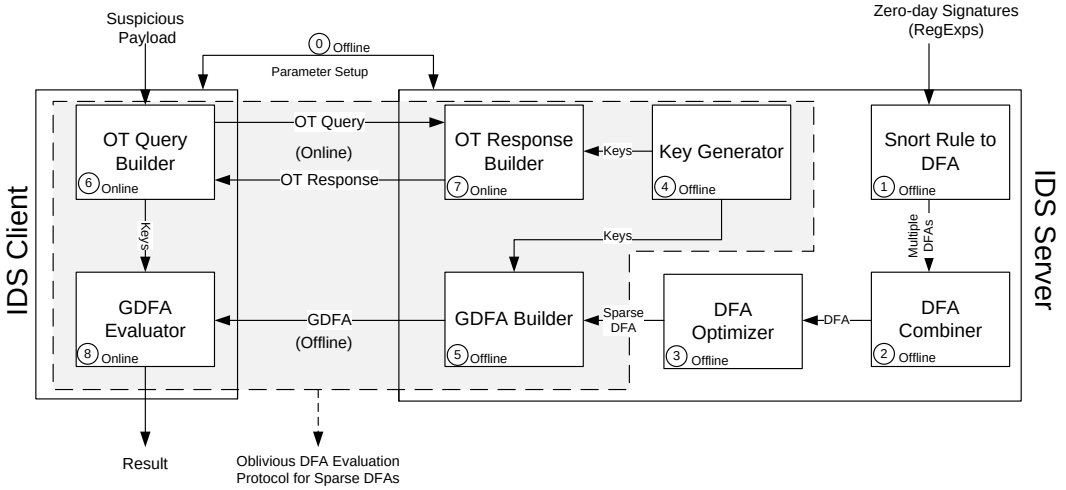

FIGURE 2. The architecture of ZIDS

(in addition to step 0) such as DFA sparsity parameters need to be exchanged.

- OT Query Builder (Step 6 - Online): This component is run as part of the ODFA protocol. It generates the OT queries based on the IDS client's input. (Refer to section 2 for definition of the OT primitive and section 5 for its use in the ODFA protocol). The OT queries are then sent to the IDS server.
- OT Response Builder (Step 7 - Online): This component which is also a part of the ODFA protocol, builds the OT responses upon receiving the OT queries from the client. The OT responses are sent back to client. The client can retrieve a key per each input symbol.
- GDFA Evaluator (Step 8 - Online): Using this component, the client can decrypt a "transit path" in the GDFA matrix using the keys received through the OT. The last element of the transit path decrypted by the client is the result of DFA evaluation. This is either equal to 0 which would mean there is no attack pattern in the suspicious payload, or equals to an ID of an attack pattern found in the suspicious payload.

Note that steps 1, 2 and 3 can be amortized over multiple protocol runs, while steps 0, 4 and 5 cannot. This is because for each protocol run, a new set of symmetric keys has to be used in the garbling phase to ensure the security of the protocol. However as we mentioned earlier, these computations can be done in the offline phase.

4. REPRESENTING RULE SETS USING DFAS

As we noted in section 3, in our proposed privacy preserving IDS, the attack patterns are converted to DFAs so that they can be processed by the oblivious DFA evaluation protocol. The attack patterns are generally part of the rule set of an IDS such as Snort [24]. Each rule in Snort's rule set consists of both heading and content processing elements. The conditions such as protocol type (tcp, udp, icmp), source and destination IP addresses, and ports are examples of header processing fields. The *pcre* (perl compatible regular expressions) and *content* elements are the main content processing elements. The *content* option specifies a string to be matched in the payload. The *pcre* is used to find more complex patterns in a payload and is expressed using regular expressions. Table 1 depicts the *pcre* and *content* elements in three sample rules of the Snort IDS. These rules are used to detect various types of web application attacks.

In order to generate a single DFA from the rule set for an IDS such as Snort, the following three steps are performed by the IDS server.

4.1. Generating each rule's DFA from its *content* and *pcre* fields.

In this step, our goal is to build a unique DFA for each Snort rule. To do this, we first convert the *pcre* and *content* fields into their equivalent DFA representations using standard RegExp-to-DFA conversion algorithms [25, 18]. Note that the content field can be considered as a simple regular expression that only contains a string.

In order to build the unique DFA of each rule, our idea is to chain the *pcre* and *content* DFAs by connecting the accepting states of the first DFA to the starting state of the second one, with an ϵ-transition. This yields a non-deterministic finite automata (NFA) that only accepts a string if it is accepted by both DFAs (Figure 3). The resulting NFA can then be converted to an equivalent DFA using standard NFA-

TABLE 1. Sample Snort rules and their content processing elements (*pcre* and *content*)

Snort Rule-set	*Content* Field	*pcre* Field	Attack Description
web-php	graph_image.php	*/rra_id=(?!(\d+|all))([\x26\s]|$))/smi*	cacti graph_image SQL injection attempt
sql	xp_cmdshell	-	SQL xp_cmdshell attempt
web-misc	/sharepoint/	/sharepoint[\n] * \x22\s * \x29\s * \x3b/Ui	Sharepoint cross site scripting attempt

to-DFA algorithms [18, 26].

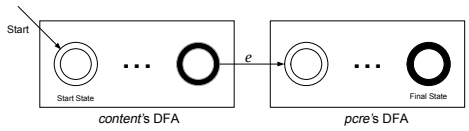

FIGURE 3. The DFA generated by chaining content's DFA before pcre's DFA

In the above, we have implicitly made the assumption that the *content* string appears before the *pcre* pattern in a malicious payload, so we place the *content*'s DFA as the first DFA in the chain. If for a specific rule, the *pcre* should appear first, then the DFAs can be chained in the opposite order. For rules where the order of the elements is not important, we can construct the two chains (pcre-content and content-pcre) and generate a new starting state that connects to the starting states of both chains by ϵ-transitions. This can be viewed as OR-ing the two chains (Figure 4).

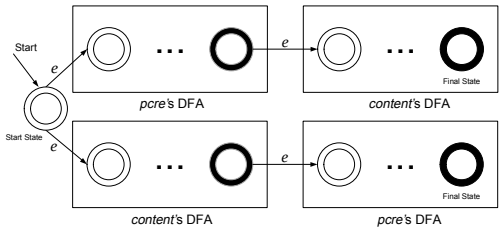

FIGURE 4. The DFA generated by chaining and ORing the pcre and content DFAs

4.2. Combining the DFAs for all rules to a single DFA.

As discussed before, to avoid a separate invocation of our ODFA protocol for each rule, we combine the DFAs into a single DFA for the whole rule set. To function as intended, for any input string that matches an attack pattern in the rule set, the new DFA should transit to an accepting state. Hence, the new DFA can be viewed as an OR of the DFAs for each rule. Similar to the previous transformation, to create this new DFA, we generate a new starting state and connect it to the starting state of each rule's DFA by an ϵ-transition. Then we convert the resulting NFA to an equivalent DFA using standard NFA-to-DFA algorithms.

To distinguish the different accepting states in the newly generated DFA, we assign a label to each state that specifies the attack ID of the matching rule.

4.3. Optimizing the generated DFA.

In this step we reduce the size of the DFA generated in the previous step and also perform sparsity analysis to extract sparsity parameters of the generated DFA. This step helps to improve the efficiency of the ODFA protocol. DFA minimization is the task of transforming a given DFA into an *equivalent* DFA that has the least number of states possible. Two DFAs are equivalent if they describe the same regular language [18].

Several different algorithms for DFA minimization are known such as Hopcroft's algorithm [27] and Watson and Daciuks's algorithm [28]. According to the empirical results of [29], Watson and Daciuks's algorithm performs better than the others when the alphabet size is more than 10 symbols, which is the case for our DFAs since Snort uses ASCII characters as its alphabet.

The details for the sparsity analysis come in the next section.

5. AN OBLIVIOUS DFA EVALUATION PROTOCOL FOR SPARSE DFAS

Existing solutions [9, 11, 12, 10] for oblivious DFA evaluation mainly consider a binary alphabet. Although these solutions can be extended to support an ASCII alphabet, these extensions generally yield schemes that have a higher computation or communication complexity. In particular, increasing the size of the alphabet increases the complexity of these protocols by a factor of $O(\Sigma)$ which is a significant overhead. For example, consider the protocol of [12] which is computationally quite efficient, due to the limited use of public-key operations. When the alphabet size is Σ, the communication and the server-side computation of their protocol is $O(N \cdot |Q| \cdot |\Sigma| \cdot k)$ bits where N is the client's input size, $|Q|$ is the number of DFA states, and k is the security parameter. When the alphabet size grows from 2 (binary) to 256 (ASCII), this translates to a factor 128 times increase in the complexity of the protocol.

In this section, we propose a new protocol that is significantly more efficient than existing ODFA solutions when the DFA is sparse and has almost the same computation and communication complexities when the DFA is dense. In fact, the more sparse a DFA is, the more efficient our new protocol is, compared

to existing solutions. Our goal is to design this more efficient protocol without having to reveal any sensitive information about the structure of the sparse DFA. In particular, the only additional information leaked to the input holder about the DFA is two sparsity parameters, *outmax* and *cmax* which we explain in more detail, shortly.

Our work is based on the recent construction of [12] who design an efficient protocol for binary alphabets. Their protocol runs in one round and only requires a small number of exponentiations ($O(s)$) for both parties, where s is a statistical security parameter. The basic idea is to represent the DFA as a DFA matrix, and then have the server permute and encrypt this matrix and send the resulting garbled matrix to the client. Given his input keys, the client can decrypt a single *transit path* in this garbled matrix and compute the final result. As this solution is proposed for binary alphabet, an extension to ASCII alphabet is needed. To extend it to ASCII alphabet, the elements in each matrix cell should be increased from 2 to 256 and a 1-out-of-256 OT should be used to retrieve each input key. This yields a protocol with high communication complexity, as the size of the garbled matrix will be $256 \cdot N \cdot |Q| \cdot k$, compared to the original size of $2N \cdot |Q| \cdot k$.

5.1. Using Sparsity of the DFA to Gain Efficiency

We take advantage of the sparsity of the input DFA to design a more efficient protocol. In sparse DFAs, a group of characters in the alphabet can transit from a single state to the same next state. This is unlike a *dense* DFA in which each character transits the DFA to a different state. Intuitively, if a DFA is sparse, the number of outgoing edges from each state is small compared to the size of Σ. As discussed in section 5.1.2, generally, the DFAs used to represent attack patterns are sparse. Figures 5 and 6 depict samples of sparse and dense DFAs, respectively.

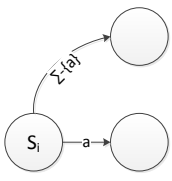

FIGURE 5. Sample of a sparse DFA

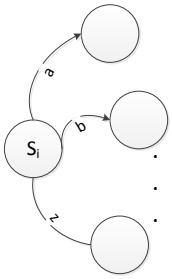

FIGURE 6. Sample of a dense DFA

5.1.1. DFA Sparsity Parameters

For the DFA $\Gamma(Q, \Sigma, \Delta, s_0, F)$, we define the notion of a character group as the set of characters in Σ, for which, there exists a state q_i where the transition from q_i using any of the characters in the group yields the same destination state q_j. More formally, we say that a and b are in the same character group if $\Delta(q_i, a) = \Delta(q_i, b) = q_j$. We denote the set of character groups of a state i as $C_i = \{C_i[1], C_i[2], ...\}$ where $C_i[1]$ denote the first character group of state i, $C_i[2]$ the second, and so on. We denote by $C = C_1 \cup ... \cup C_{|Q|}$, the set of all character groups of the DFA Γ.

We denote the set of character groups that have the character $x \in \Sigma$ as a member with $C^x = \{C^x[1], C^x[2], ...\}$ where $C^x[1]$ is the first character group of x, $C^x[2]$ the second, and so on.

outmax is the maximum out-degree of states in Γ. Formally, $outmax = max_{i \in Q}(|C_i|)$. Also *cmax* is the maximum number of character groups that a character is member of. Formally, $cmax = max_{x \in \Sigma}(|C^x|)$. We call $(C, cmax, outmax)$ the sparsity parameters of Γ.

For example the sparsity parameters for the DFA of figure 7 are as follows. $C_1 = \{\{a\}, \{\Sigma - a\}\}$, $C^a = \{\{a\}, \{\Sigma\}, \{\Sigma - c\}\}$, and $C = \{\{a\}, \{b\}, \{c\}, \{d\}, \{\Sigma\}, \{\Sigma - a\}, \{\Sigma - c\}, \{\Sigma - \{a, b\}\}, \{\Sigma - \{a, d\}\}\}$. Furthermore, $outmax = 3$, and $cmax = 5$ since c is member of 5 groups in C.

5.1.2. DFAs for Attack Patterns are Small

We start with showing that for any regular expression corresponding to a fixed pattern or patterns containing only the $*$ operator, the out-degree of the resulting DFA is at most 3. In other words, the *outmax* parameter for such DFAs is at most 3 regardless of the size of the alphabet (figure 7 is one such example for the regular expression $/ab*cd/$). The following Theorem formalizes this statement.

Theorem 1. For any regular expression corresponding to a fixed pattern or patterns containing only the $*$ operator, the out-degree of its equivalent DFA is at most 3.

Proof. To prove this, consider a general regular expression $p = p_1 p_2 \cdots p_m \cdots p_n$ that has a wildcard operator at location m ($p_m = *$ and $p_i \in \Sigma$ for $i \neq m$). Here we construct its equivalent DFA Γ_p that has $outmax = 3$. Note that Γ_p has n states $S_1 \cdots S_m \cdots S_n$. The first state has a transition with label $\Sigma - \{p_1\}$ to itself and another transition with label p_1 to S_2. Each state S_i ($i < n$) has one transition with label p_i to S_{i+1}, the second transition with label p_1 to S_2 for ($i \neq m, n$ and $p_1 \neq p_i$). Finally states can have a third transition with label $\Sigma - \{p_1, p_{i+1}\}$ to S_1 for ($i \neq m, n$). S_m has a transition with label $\Sigma - \{p_{m+1}\}$ to itself and a transition with label p_{m+1} to S_{m+1} ($outmax = 2$). The last state has a transition with label Σ to itself ($outmax = 1$). Therefore the *outmax* parameter for the resulting DFA does not exceed 3. □

For more general patterns, it is not easy to have a formal study on the sparsity of signatures due to the fact that they don't have any obvious mathematical

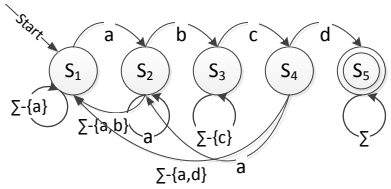

FIGURE 7. The DFA for the pattern $/ab*cd/$

TABLE 2. Sparsity Parameters for some DFAs representing IDS signatures

Sig File	DFA States	DFA Edges	cmax	outmax
Sig_2	16	4096	11	4
Sig_6	45	11520	17	6
Sig_10	87	22272	26	8
Sig_20	176	45056	46	13

characteristics. Therefore, we only rely on experimental studies. Our empirical studies reveal that the DFAs generated via combining multiple attack patterns are also quite sparse, even if the patterns are described using more general regular expressions. Table 2 shows the *outmax* and *cmax* parameters for DFAs generated by combining various number of rules, from a sample snort rule set (2-20 signatures)[4]. Note that outmax degrees are a factor of 20-60 smaller than the alphabet size of 256. Next, we design a protocol that takes advantage of this reduction in outdegree while only revealing the values *outmax* and *cmax* but nothing else.

5.2. DFA Matrix

Our proposed oblivious evaluation protocol for sparse DFAs takes advantage of a *matrix representation* of DFAs. Next we briefly review the notions of a *DFA matrix* and a *permuted DFA matrix* which we use throughout the paper. The binary version of these notions are first introduced in [12]. Here we extend these notions to ASCII alphabet and incorporate the concept of character group into them.

5.2.1. DFA Matrix

Assume that the input string of a DFA $\Gamma = (Q, \{\Sigma\}, \Delta, s_1, F)$ is a string $X = x_1x_2 \dots x_n \in \{\Sigma\}^n$. 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 $1 \le i \le n$, the ith row of M_Γ represents the evaluation of x_i.

In particular, the element $M_\Gamma[i, j]$ stores the tuple $(\Delta(j, C_j[1]), \Delta(j, C_j[2]), \cdots, \Delta(j, C_j[|C_j|]))$ which encodes the indices of the next states to be visited (at row $i + 1$) for each character group of state j.[5] At row n where the last character x_n is processed, instead of storing the indices of the next states, we place a 1 (or possibly a positive integer of a specific length) if the next state is an accepting state and a 0 otherwise.

[4]We used the first 20 rules (with *pcre* field) in *web-php.rules* of snortrules-snapshot-2904[30].

[5]Note that here we assume the transition function (Δ) can take a character group as its second argument.

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
    for 1 ≤ z ≤ |C_j| do
      if i ≤ n − 1 then
        M_Γ[i, j, z] ← Δ(j, C_j[z])
      else if i = n then
        if Δ(j, C_j[z]) ∈ F then
          M_Γ[i, j, z] ← 1
        else
          M_Γ[i, j, z] ← 0
        end if
      end if
    end for
  end for
end for
return M_Γ
```

Evaluation using the DFA Matrix. One can use M_Γ to efficiently evaluate Γ on any input X of length n. We start at $M_\Gamma[1, 1]$. If x_1 is a member of the z'th character group of the first state ($x_1 \in C_1[z]$), the z'th index of the tuple $M_\Gamma[1, 1]$ ($M_\Gamma[1, 1, z]$) is used to locate the next cell to visit at row 2. Then by considering the chosen tuple in row 2 and the value of x_2, we 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 tuples 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.

5.2.2. 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 Algorithm 1 ($\mathsf{GenPerm}$). Then, algorithm 3 is invoked to convert a DFA matrix M_Γ to an equivalent permuted DFA Matrix 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 $PM_\Gamma[1, PER[1, 1]]$.

Next, we present an overview of our proposed protocol.

Algorithm 3 PermDfaMat(M_Γ, PER)

```
for 1 ≤ i ≤ n do
  for 1 ≤ j ≤ |Q| do
    for 1 ≤ z ≤ |C_j| do
      if i ≤ n − 1 then
       PM_Γ[i, PER[i, j], z] ← PER[i + 1, M_Γ[i, j, z]]
      else if i = n then
       PM_Γ[n, PER[n, j], z] ← M_Γ[i, j, z]
      end if
    end for
  end for
end for
return PM_Γ
```

5.3. Overview of the Protocol

We start with a high level overview of our protocol. As discussed before, the idea is to use the sparsity of the DFA to our advantage. We start with the construction of [12] which transforms a DFA to a DFA matrix before garbling it. In our proposed scheme, we generate a significantly smaller size garbled DFA matrix. As discussed above, the trivial extension of [12] to ASCII alphabet has to have 256 elements in each matrix cell. We reduce this number to *outmax* elements in each matrix cell, which reduces the total size of the DFA matrix from $O(256 \cdot |Q| \cdot N \cdot k)$ bits to $O(outmax \cdot |Q| \cdot N \cdot k)$ bits where $outmax << 256$. The challenge is to do so without compromising the privacy of the DFA holder. In particular, we do not want to leak any information besides the *outmax* and *cmax* parameters to the client while still enabling him to evaluate his input on the DFA.

In contrast to the construction of [12], we assign a unique random key to each character group instead of each character in the alphabet, and then use this key to encrypt each element in the DFA matrix. The content of each element in the DFA matrix is the same as the basic protocol: an index to the next cell in row $i + 1$ and a random pad.

In the basic protocol, the client receives a key for each input character, while in our proposed protocol, the client receives a set of keys corresponding to all the character groups that have the input character as a member. As discussed earlier, we denote the maximum number of such character groups by *cmax*. This set of keys is sent to the client by invoking a 1-out-of-256 OT protocol between the client and the server. One of these keys can be used to decrypt the next element in conjunction with its pad. The client tries these keys one by one, finds the appropriate key and learns the next state index at row $i+1$ and its pad (Table 3 shows basic differences between our proposed protocol with [12]).

In the following, we give a more detailed (but still informal) description of the steps involved in our protocol. In each step, we also point to the related lines of the formal protocol (section 5.4) to help reader better follow the protocol steps.

TABLE 3. A comparison of our proposed protocol with [12]

	[12]	Our Protocol
No. of elements per GDFA cell	$\|\Sigma\|$	*outmax*
GDFA size (bit)	$\|\Sigma\| \cdot \|Q\| \cdot n \cdot k$	$outmax \cdot \|Q\| \cdot n \cdot k$
No. of keys in each OT string	1	*cmax*

Client encrypts his inputs using OT queries, and sends them to the server. Client invokes $\mathrm{G_{OT}}$ to generate a key pair (Line 1 of step 1 in the formal protocol), and then invokes $\mathrm{Q_{OT}}$ for each of its input characters to compute the corresponding OT query q_i (Lines 2-4 of step 1). The inputs to the $\mathrm{Q_{OT}}$ algorithm are the public key, number of server strings, length of each string (Each string contains *cmax* keys, each of length k') and the value of x_i. The client then sends the public key and the query answer vector to the server (Line 5 of step 1).

Server garbles the DFA by forming a garbled DFA matrix. The idea behind this step is to garble the DFA matrix before sending it to the client. This idea was used previously in [12] for DFAs with binary alphabets, but here we propose a different garbling technique. A simple extension of [12] for ASCII alphabet requires Σ elements in each matrix cell. As discussed before, this would impose a high communication and computation cost. In our proposed method, we reduce the number of elements in each cell to *outmax* ($outmax << \Sigma$).

The garbling phase proceeds as follows: First for each row we assign a key to each character group (Lines 1-6 of step 2.1 in the formal protocol). Also we generate a random pad for each cell of the DFA matrix (Lines 7-12 of step 2.1).

Then we generate a permuted DFA matrix PM_Γ from Γ by invoking the algorithms DfaMat,GenPerm and PermDfaMat respectively (Lines 13-18 of step 2.1).

Next, the garbled DFA matrix is computed (Step 2.2 of the protocol). First, for the states that have fewer character groups than *outmax*, we add random elements to their corresponding cell in DFA matrix so all cells have exactly *outmax* elements (lines 4-8 of step 2.2). For the row $1 \leq i \leq n-1$ of the garbled matrix, each element of each cell is filled with the permuted next index and pad at row $i + 1$. The trailing zeros are also added to detect correct decryption (Lines 9-13 of step 2.2). For the row n, we just expand the elements of PM_Γ that have the output values of 0 or 1, to the appropriate length (Lines 14-17 of step 2.2). Then each element is encrypted by its corresponding character group key (Line 19 of step 2.2). Finally all the elements of each cell are randomly permuted and further encrypted by the appropriate pad from the PAD matrix (Lines 21-26 of step 2.2).

Server computes the OT answer vector and sends it as well as the garbled DFA matrix to the client. For each OT query q_i, the server prepares its corresponding OT answer a_i. To do this, the server should first computes its $|\Sigma|$ strings $(S_{1,i} \cdots S_{|\Sigma|,i})$ used in OT (one for each $x \in \Sigma$). Each $S_{x,i}$ is formed by concatenating and permuting the keys of the character groups that contains character x (Lines 3-7 of step 3 of the formal protocol). If a character x is member of less than *cmax* character groups, we expand $S_{x,i}$ with random bits so each $S_{x,i}$ has exactly $cmax \cdot k'$ bits (Lines 8-11 of step 3). Then server computes the OT answers by invoking $\mathrm{A_{OT}}$ (Line 12 of step 3). Finally, the server sends the resulting OT answer vector plus the garbled DFA matrix and the index and pad of the starting cell in row 1 to the client (Line 14 of step 3).

Client retrieves the keys and computes the final result. In this step the client first invokes $\mathrm{D_{OT}}$ for each OT answer a_i to learn $KEYS$, the set of character group keys for x_i (Line 4 of step 4).

Then the client starts evaluating the garbled DFA matrix. First it uses the pad of the starting cell to partially decrypt it (Line 6 of step 4). Then the client uses the *cmax* keys it retrieved through OT, to decrypt one of the *outmax* values in the starting cell. To to this, it tries these keys one by one, finds the appropriate key and learns the next state index at row 2 and its pad. Note that if a key is found the client stops and goes to next cell (Lines 15-18 of step 4). The client repeats this process, moving along the transit path for input X until it reaches the last row and recovers the final output (Lines 8-22 of step 4).

First, note that for all the cells 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 cells that appear on his path, he can only decrypt one of the *outmax* values using one of the *cmax* keys he has retrieved at the OT stage. Next, we present the formal description of our proposed protocol.

5.4. The Protocol

Server's Input: A DFA $\Gamma = (Q, \Sigma, \Delta, s_1, F)$ and the set of character groups C.
Client's Input: A string $X = x_1x_2...x_n \in \Sigma^n$.
Common Input: The computational security parameter k, the OT security parameter κ and the size of DFA $|Q|$. The sparse DFA parameters *cmax* and *outmax*. We let $k' = 2k + \log|Q|$ throughout the protocol. Parties also agree on a 1-out-of-$|\Sigma|$ OT protocol $OT = (\mathrm{G_{OT}}, \mathrm{Q_{OT}}, \mathrm{A_{OT}}, \mathrm{D_{OT}})$ (we explain how we implement the 1-out-of-$|\Sigma|$ OT protocol, shortly) and a PRG $G : \{0,1\}^k \to \{0,1\}^{outmax \cdot k'}$.

1. **Client encrypts his inputs using OT queries, and sends them to server. (OT Query Builder Component)**

Sending OT Queries to server

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

2. **Server Computes a Garbled DFA matrix GM_Γ. (Key Generator and GDFA Builder Components)**

2.1. Generating random pads, character groups keys and a permuted DFA matrix PM_Γ

1: Server generates random keys for the character groups:
2: **for** $1 \le i \le n$ **do**
3: $\quad$ **for** $1 \le j \le |C|$ **do**
4: $\quad\quad (K_{C[j],i}) \xleftarrow{\$} \{0,1\}^{k'}$
5: $\quad$ **end for**
6: **end for**
7: Server generates a random pad matrix $PAD_{n \times |Q|}$:
8: **for** $1 \le i \le n$ **do**
9: $\quad$ **for** each $j \in Q$ **do**
10: $\quad\quad PAD[i,j] \xleftarrow{\$} \{0,1\}^k$
11: $\quad$ **end for**
12: **end for**
13: server generates a DFA matrix M_Γ:
14: $M_\Gamma \leftarrow \mathsf{DfaMat}(\Gamma, n)$
15: server generates a random permutation matrix $PER_{n \times |Q|}$:
16: $PER \leftarrow \mathsf{GenPerm}(n, Q)$
17: server generates a permuted DFA matrix PM_Γ:
18: $PM_\Gamma \leftarrow \mathsf{PermDfaMat}(M_\Gamma, PER)$

2.2. Computing the Garbled DFA Matrix GM_Γ from PM_Γ

1: **for** $1 \le i \le n$ **do**
2: $\quad$ **for** each $j \in Q$ **do**
3: $\quad\quad$ **for** $1 \le z \le outmax$ **do**
4: $\quad\quad\quad$ **if** $z > |C_j|$ **then**
5: $\quad\quad\quad\quad GM_\Gamma[i,j,z] \xleftarrow{\$} \{0,1\}^{k'}$
6: $\quad\quad\quad\quad$ //Skip to next z
7: $\quad\quad\quad\quad$ continue;
8: $\quad\quad\quad$ **end if**
9: $\quad\quad\quad$ **if** $1 \le i \le n-1$ **then**
10: $\quad\quad\quad\quad$ //Add the next index, next pad and trailing zeros
11: $\quad\quad\quad\quad nextindex \leftarrow PM_\Gamma[i,j,z]$
12: $\quad\quad\quad\quad nextpad \leftarrow PAD[i+1, nextindex]$
13: $\quad\quad\quad\quad GM_\Gamma[i,j,z] \leftarrow nextindex||nextpad||0^k$
14: $\quad\quad\quad$ **else if** $i = n$ **then**
15: $\quad\quad\quad\quad$ //Expand Last Row to Appropriate Length
16: $\quad\quad\quad\quad GM_\Gamma[n,j,z] \leftarrow (PM_\Gamma[i,j,z])^{(k+\log|Q|)}||0^k$
17: $\quad\quad\quad$ **end if**
18: $\quad\quad\quad$ //Encrypt GM_Γ with the character groups keys.
19: $\quad\quad\quad GM_\Gamma[i,j,z] \leftarrow GM_\Gamma[i,j,z] \oplus K_{C_j[z],i}$
20: $\quad\quad$ **end for**
21: $\quad\quad$ //Permute each cell and encrypt it using pads
22: $\quad\quad randperm \leftarrow Perm(\{1, ..., outmax\})$
23: $\quad\quad$ **for** $1 \le z \le outmax$ **do**
24: $\quad\quad\quad GM_\Gamma[i,j,randperm[z]] \leftarrow GM_\Gamma[i,j,z]$
25: $\quad\quad$ **end for**
26: $\quad\quad GM_\Gamma[i,j] \leftarrow GM_\Gamma[i,j] \oplus G(PAD[i,j])$
27: $\quad$ **end for**
28: **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. (OT Response Builder Component)**

Sending OT Answers and the Garbled Matrix to client

for $1 \le i \le n$ **do**
  for $1 \le x \le |\Sigma|$ **do**
    //Concatenating and permuting keys for the character groups of character x:
    $randperm \leftarrow Perm(\{1, ..., |C^x|\})$
    $z' \leftarrow randperm[z]$
    $S_{x,i} \leftarrow \lambda$
    for $1 \le z \le |C^x|$ **do**
      $S_{x,i} \leftarrow S_{x,i} || K_{C^x[z'],i}$
    end for
    //Expanding $S_{x,i}$ if its size is less than $cmax \times k'$:
    $KR \xleftarrow{\$} \{0,1\}^{k' \cdot (cmax - |C^x|)}$
    $S_{x,i} \leftarrow S_{x,i} || KR$
  end for
  $a_i = \mathrm{A}_{\mathrm{OT}}(pk, q_i, S_{1,i}, \cdots, S_{|\Sigma|,i})$
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. (GDFA Evaluator Component)**

Computing the Final Output

$state \leftarrow PER[1,1]$
$pad \leftarrow PAD[1, PER[1,1]]$
for $1 \le i \le n$ **do**
  $KEYS \leftarrow \mathrm{D}_{\mathrm{OT}}(sk, a_i)$
  $K_1 || \cdots || K_{cmax} \leftarrow KEYS$
  $GM_\Gamma[i, state] \leftarrow GM_\Gamma[i, state] \oplus G(pad)$
  //Trying to decrypt $GM_\Gamma[i, state]$ using different K_z's:
  for $1 \le j \le outmax$ **do**
    for $1 \le z \le cmax$ **do**
      if $i < n$ **then**
        $newstate || newpad || tail \leftarrow K_z \oplus GM_\Gamma[i, state, z]$
      else if $i = n$ **then**
        $result || tail \leftarrow K_z \oplus GM_\Gamma[i, state, z]$
      end if
      //Exit from the two outer FORs if decryption is successful:
      if $tail = 0^k$ **then**
        break 2
      end if
    end for
  end for
  $pad \leftarrow newpad$
  $state \leftarrow newstate$
end for
Client outputs *result* as his final output.

5.5. Complexity Analysis

Round Complexity: The protocol consists only 1 round of communication between the client and server.

Asymmetric Computations: 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 exponentiations in our protocol is bounded by $O(n)$ for both server and client. We reduce this to $O(s)$ using the OT extension of [23], where s is the statistical security parameter (at the cost of adding an extra round). This improves the efficiency significantly when $n >> s$, which is the case for our IDS application.

Symmetric Computations: The only symmetric computation in our protocol is PRG invocations. The server and client perform $O(n \cdot Q)$ and $O(n)$ PRG invocations where Q is the DFA size. This is due to the fact that the garbled DFA matrix has $n \cdot Q$ cells and for each cell the server performs one PRG invocation. On the other hand, the client invokes one PRG for each row of the garbled matrix to decrypt exactly one element per each row.

Communication Complexity: The communication overhead of our proposed protocol consists of OT communication and garbled DFA transfer. The OT communication is $O(n \cdot cmax \cdot k)$ bits and the size of the garbled DFA matrix is $O(n \cdot Q \cdot outmax \cdot k)$ bits. Therefore the total communication cost would be $O(n \cdot k \cdot (cmax + Q \cdot outmax))$ bits.

5.6. Security

The security proof of our proposed protocol is almost the same as the ODFA protocol of [12]. Specifically, as long as 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 2. In the OT-hybrid model, and given a computationally secure PRG G, the above protocol is *fully-secure* against a malicious client and is *private* against a malicious server.

Proof skecth. We sketch the ideas for proving 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. Since the proof and the simulator are almost the same as [12], here we describe a sketch of the proof and highlight the differences:

- $\mathcal{S}_\mathcal{B}$ runs $\mathcal{B}$ on inputs $X, k, |Q|$ as well as the sparsity parameters $outmax$ and $cmax$. This is one difference with proof of [12], since here the sparsity parameters are learned by the parties.
- Since we operate in the OT hybrid model, $\mathcal{B}$ sends an input $X' = x'_1 \ldots x'_n$ to the OT's trusted party. $\mathcal{S}_\mathcal{B}$ generates n random set of keys

$(S_{1,i}, S_{2,i}, ..., S_{|\Sigma|,i})$ for $1 \leq i \leq n$ and sends $S_{x_i,i}$ back to $\mathcal{B}$.

- $\mathcal{S}_\mathcal{B}$ generates a Garbled DFA matrix GM' that has *outmax* elements in each cell instead of the Σ elements as in [12]. Intuitively, all the element values in GM' are set to random k'-bit strings except for the values on the transit 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.
- Now it is straightforward to prove that $\mathcal{B}$ cannot distinguish between his view during the real execution and his interaction with $\mathcal{S}_\mathcal{B}$. To prove this one can 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 $\mathcal{S}_\mathcal{B}$. The goal is to show that $D_0 \stackrel{c}{\equiv} D_n$. This can be done through a simple hybrid argument where it is shown that $D_t \stackrel{c}{\equiv} D_{t+1}$ for $1 \leq t < n$. 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 that appears on X's transit path. Readers are refered to [12] for the full discussion on the indistinguishability of views.

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.

□

6. IMPLEMENTATION AND EVALUATION

In this section we explain the details of our implementation, evaluate the proposed system and provide some comparison with alternative implementations.

6.1. Implementation Details

The implementation of ZIDS is done using C++ and the Crypto++ library v.5.61 [31].

OT Implementation. Our ODFA protocol uses a 1-out-of-256 OT as a sub-protocol. To reduce the number of public-key operations needed for the OTs, we use the construction of Naor and Pinkas [22] to efficiently reduce n invocations of the 1-out-of-256 OT to $8n$ invocations of 1-out-of-2 OT.

For our 1-out-of-2 OT protocol, we implement the Naor-Pinkas amortized OT protocol (See section 3.1 of [19]) which requires one exponentiation for each transfer. We implement 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 [32]).

We have taken advantage of the OT extension of [23] in our implementation for improved efficiency. There are two different extensions in [23]. The first one is concerned with extending the number of OTs efficiently (section 3 of [23]) while the second extension (Appendix B of [23]) reduces oblivious transfer for long strings to oblivious transfer of shorter strings. Both implementations of the mentioned extensions rely on the use of a hash function (in the random oracle model). We have chosen SHA-256 for this implementation. As a result, regardless of the input or the DFA sizes, the number of exponentiations necessary for the OTs does not exceed $O(s)$ where s is a statistical security parameter. We let $s = 80$.

Offline Computations. For efficiency reasons, we precompute several components of the protocol that do not depend on the client's input, such as the DFA garbling phase. Note that the IDS server does not need to have the IDS client's OT queries in order to calculate the garbled matrix. Therefore, the IDS server precomputes the garbled DFA and sends it to the client in an offline phase. This leads to a significant decrease in the online overhead of our protocol.

We also use the OT precomputation technique of [33] to precompute the OT's expensive operations (exponentiations) and only perform XOR operations in the online phase.

Reduction of Memory Usage. Since the DFA matrix corresponding to real-life rule sets can be quite large, it is critical to avoid the storage of the whole DFA matrix by the server. Note that the server might have several clients, each with a separate DFA matrix, and therefore optimizing the memory usage for the server seems very useful. To achieve this, we implement our protocol such that in the offline phase, the server generates and garbles exactly one row of the matrix; sends it to the client over the network and discards the computed row. So at each point, the server only stores a single row of the matrix. Note that here the client has to retain the full DFA matrix in order to later use it in the online phase.

6.2. Data Sets

IDS Server attack patterns. For the attack patterns, we used the latest Snort signature set (snortrules-snapshot-2904) [30] and extracted the content processing elements from *web-php*, *web-misc*, *web-cgi*, *sql.rule*, *netbios* and *exploit* rule files.

We have converted the content processing elements of each rule to its equivalent DFA. The rules' DFAs are then combined and optimized to form a unique DFA that can be fed to our proposed ODFA protocol.

IDS Client Input. In our experiments the content of client's input does not affect the efficiency of the

protocols. The only important factor is the size of his input string. As a result, various random strings of different sizes are considered as the IDS client's input in our experiments.

6.3. Experiment Set Up

We have used two equivalent PCs as the IDS client and IDS server connected via a Gigabit Ethernet. Each PC has an Intel Core2 Duo E6750 with @2.66Ghz CPU, and 4GB of RAM. Both PCs have Linux CentOS 5.5 installed as their Operating System.

6.4. Experiment Results

We performed a number of experiments to evaluate the efficiency of our proposed system. In our experiments, we combined a different number of signatures, leading to a different number of states in the resulting DFAs. As discussed, we have tried to push the computation and communication to the offline phase. Garbling the DFA by server and the OT precomputation by both parties is done in the offline phase. As a result, the online time complexity of our protocol is quite low (usually less than 1 second).

In our experiment we use the combination of two and six signatures respectively. For the zero-day detection scenario this is usually a reasonable number of signatures. For each of these combinations, we increase the size of the input and record the result. Since the target of this paper is to detect zero-day type patterns, we have considered an input size suitable for detecting the common vulnerabilities. It is known that beside IP packets, many of the new targets for zero-day attacks are PDF or office documents. The vulnerability in these types of files usually resides in the first segments of the files, which has a fairly small size. Hence we consider input sizes ranging from 500 to $4k$ bytes. Table 4 shows the results of this experiment.

According to this table, the total online time of the protocol is less than 100 ms for the 16 states DFA and less than 5 seconds for the 45 states DFA, which is quite efficient in practice. For the online interaction, the client and server should perform the OT's online computation and the client has to evaluate the garbled matrix to learn the result. Note that the online component of the OT only requires XOR operations. As for the offline computation, the client's time is still below 5 seconds, while the server's ranges from 23 seconds to 5 minutes. Our experiments confirm that client's work is fairly small in both the online and the offline phase, and that due to the precomputation phase, we can optimize the online response time of the server to a large extent.

Communication Results. Table 5 shows the communication overhead of ZIDS and compares it with alternative solutions. According to this table, the online communication for a DFA of 45 states, and a 512 byte input is around 6 MB. Although the communication size is still far better than the alternative solution, it can grow with the input size and become the bottleneck of the protocol when network resources are limited. For these scenarios, we have considered a tradeoff that considerably reduces the online communication, in exchange for one symmetric-key encryption on the client side for each input byte. Considering the fast implementations of AES in software and also in modern CPUs, it is not a computationally expensive task for the client to perform an additional encryption per each input character. Next we explain this technique.

6.5. Online Phase Key Size Optimization

The idea is to send the encryption of long keys in the offline phase, and in the online phase send the shorter keys required to decrypt the corresponding long keys. This technique proceeds as follows. Let's denote the keys which are normally used as input to OT in online phase by K. In the offline phase, server uses the smaller keys K' to encrypt all such keys, $ek = E_{K'}(K)$ and sends ek's to the client. In the online phase client already possesses the encrypted version of all keys. So instead of performing the OTs on the original keys K, server will use much smaller keys K' in the online phase. Client will use K'_i that received with OT to decrypt the corresponding ek_i and recover the key K_i. This yields a factor of $cmax$ improvement in the online communication.

6.6. Comparison with Garbled-circuit-based Implementations.

As we discussed in the introduction, one can use the general-purpose protocols for secure two-party computations as a tool for building privacy preserving IDSes. However, it turns out that the general solutions are too inefficient for the IDS applications we consider in this paper. To show this, we compare our modified ODFA protocol with a number of most recent implementations of Yao's garbled circuit protocol [7, 8, 34].

In order to do a somewhat concrete comparison, we first generate the garbled circuit required for evaluation of a DFA with various input sizes using the compiler of [34] and then counted the number of non-XOR gates. Note that in these new implementations the XOR gates can be evaluated for free, therefore we do our analysis based on the number of non-XOR gates only. Then, given that these papers use machines with similar (or even stronger) CPU, and RAM capabilities to ours we calculated approximate computation and communication costs involved in evaluating these garbled circuits using the estimates and performance numbers given in these papers. Authors of [8] estimated a computation time of 10 micro seconds per gate,

TABLE 4. Computation Times

DFA size	Input Size (Bytes)	cmax	outmax	OT Time (s)	Client. Eval. (s)	Total online (s)	Client offline (s)	Server offline (s)	tasty[7] (s)	fastgc[8] (s)	[12] (s)	KSS[34] (s)
16	512	11	4	0.004	0.006	0.010	0.62	23.62	1892	430	557	164
16	1024	11	4	0.007	0.010	0.017	1.10	47.25	3783	860	1112	360
16	2048	11	4	0.014	0.017	0.031	2.07	94.31	7567	1720	2217	792
16	4096	11	4	0.043	0.031	0.074	4.09	188.65	15134	3439	4434	1743
45	512	17	6	0.557	0.007	0.564	0.64	39.40	6984	1587	600	604
45	1024	17	6	1.069	0.012	1.081	1.14	78.66	13967	3174	1197	1330
45	2048	17	6	2.093	0.022	2.115	2.27	162.20	27934	6349	2467	2925
45	4096	17	6	4.141	0.042	4.183	4.54	320.3	55868	12697	4965	6435

TABLE 5. Communication Comparison

DFA size	Input size (Bytes)	Ours (MB)	[12] (MB)	Circuit Size (non-XOR Gates)	Tasty [7] (MB)	KSS [34] (MB)
16	512	4	55	2^{25}	2340	768
16	1024	8	110	2^{26}	4680	1689
16	2048	16	221	2^{27}	9361	3716
16	4096	32	441	2^{28}	18721	8176
45	512	6	86	2^{27}	6582	2835
45	1024	12	172	2^{28}	13163	6236
45	2048	24	345	2^{29}	26327	13720
45	4096	49	688	2^{30}	52654	30183

while [7] and [34] provide result tables with performance numbers for different circuit sizes.

The results show that in some cases, our proposed scheme performs up to 1000 times better than the general-purpose solutions both in terms of communication and computation.

7. CONCLUSION

In this paper we proposed ZIDS, a client-server solution for private detection of intrusions. The system consists of an IDS server that has a set of sensitive signatures for zero-day attacks and IDS clients that possess some sensitive data. Using ZIDS, each IDS client learns whether its input data matches any of the zero-day signatures, but neither party learns any additional information.

ZIDS needs an efficient cryptographic protocol for oblivious evaluation of DFAs in order to privately check the IDS client input data against sensitive zero-day signatures of the IDS server. We proposed such a protocol that meets the standard security requirements of *secure two-party computation* and is also quite efficient compared to existing ODFA protocols. To the best or our knowledge, our work is the first to consider the use of ODFA protocols in privacy-preserving IDS applications.

Our implementation can process arbitrary rule sets from the Snort's IDS and analyze input strings for potential intrusions, privately. We evaluate the performance of our implementation on varying input sizes and provide both analytical and concrete comparisons to alternative solutions, in some cases showing *several orders of magnitude* (several thousand times) improvement in computation and communication cost.

REFERENCES

[1] TippingPoint. Zero day initiative. `http://www.zerodayinitiative.com/`.

[2] Mozilla. Security bug bounty program. `http://www.mozilla.org/security/bug-bounty.html`.

[3] Google. Web application security research. `http://googleonlinesecurity.blogspot.com/2010/11/rewarding-web-application-security.html`.

[4] TRAPKIT. Patch development time statistics. `http://www.trapkit.de/advisories/pdts.php`.

[5] Yao, A. C. (1982) Protocols for secure computations. *Proceedings of SFCS '82*, Chicago, IL, 3-5 November, pp. 160–164. IEEE Computer Society, Washington, DC.

[6] Malkhi, D., Nisan, N., Pinkas, B., and Sella, Y. (2004) Fairplay - a secure two-party computation system. *Proceedings of USENIX Security '04*, San Diego, CA, 9-13 August, pp. 20–20. USENIX Association, Berkeley, CA.

[7] Henecka, W., Kögl, S., Sadeghi, A.-R., Schneider, T., and Wehrenberg, I. (2010) TASTY: tool for automating secure two-party computations. *Proceedings of CCS '10*, Chicago, IL, 4-8 October, pp. 451–462. ACM, New York.

[8] Huang, Y., Evans, D., Katz, J., and Malka, L. (2011) Faster secure two-party computation using garbled circuits. *Proceedings of USENIX Security '11*, San Francisco, CA, 8-12 August, pp. 35–35. USENIX Association, Berkeley, CA.

[9] Troncoso-Pastoriza, J. R., Katzenbeisser, S., and Celik, M. (2007) Privacy preserving error resilient DNA searching through oblivious automata. *Proceedings of CCS '07*, Alexandria, VA, 28-31 October, pp. 519–528. ACM, New York.

[10] Frikken, K. B. (2009) Practical private DNA string searching and matching through efficient oblivious automata evaluation. *Proceedings of DBSec '09*, Montreal, P.Q., Canada, 12-15 July, pp. 81–94. Springer-Verlag, Berlin.

[11] Gennaro, R., Hazay, C., and Sorensen, J. S. (2010) Text search protocols with simulation based security. *Proceedings of PKC '10*, Paris, France, 26-28 May, pp. 332–350. Springer-Verlag, Berlin.

[12] Mohassel, P., Niksefat, S., Sadeghian, S., and Sadeghiyan, B. (2012) An efficient protocol for oblivious DFA evaluation and applications. *Proceedings of CT-RSA '12*, San Francisco, CA, 27 February – 2 March, pp. 398–415. Springer-Verlag, Berlin.

[13] Biskup, J. and Flegel, U. (2000) Transaction-based pseudonyms in audit data for privacy respecting intrusion detection. *Proceedings of RAID '00*,

Toulouse, France, 2-4 October, pp. 28–48. Springer-Verlag, London, UK.

[14] Flegel, U. (2007) *Privacy-Respecting Intrusion Detection (Advances in Information Security)*. Springer-Verlag, New York.

[15] Ringberg, H. A. (2009) Privacy-preserving collaborative anomaly detection. PhD thesis Princeton University Princeton, NJ.

[16] Zhang, J., Borisov, N., and Yurcik, W. (2006) Outsourcing security analysis with anonymized logs. *Proceedings of Securecomm '06*, Baltimore, MD, 28 August – 1 September, pp. 1–9. IEEE Computer Society, Washington, DC.

[17] Kim, H. (2010) Privacy-preserving distributed, automated signature-based detection of new Internet worms. PhD thesis Carnegie Mellon University Pittsburgh, PA.

[18] Sipser, M. (1996) *Introduction to the Theory of Computation*. International Thomson Publishing, Stamford, CT.

[19] Naor, M. and Pinkas, B. (2001) Efficient oblivious transfer protocols. *Proceedings of SODA '01*, Washington, D.C., 7-9 January, pp. 448–457. Society for Industrial and Applied Mathematics, Philadelphia, PA.

[20] Lipmaa, H. (2005) An oblivious transfer protocol with log-squared communication. *Proceedings of ISC '05*, Singapore, 20-23 September, pp. 314–328. Springer-Verlag, Berlin.

[21] Peikert, C., Vaikuntanathan, V., and Waters, B. (2008) A framework for efficient and composable oblivious transfer. *Proceedings of CRYPTO '08*, Santa Barbara, CA, 17-21 August, pp. 554–571. Springer-Verlag, Berlin.

[22] Naor, M. and Pinkas, B. (2005) Computationally secure oblivious transfer. *Journal of Cryptology*, **18**, 1–35.

[23] Ishai, Y., Kilian, J., Nissim, K., and Petrank, E. (2003) Extending oblivious transfers efficiently. *Proceedings of CRYPTO '03*, Santa Barbara, CA, 17-21 August, pp. 145–161. Springer-Verlag, Berlin.

[24] Roesch, M. (1999) Snort - - lightweight intrusion detection for networks. *Proceedings of USENIX System Administration '99*, Seattle, Washington, 6-11 June, pp. 229–238. USENIX Association, Berkeley, CA.

[25] Berry, G. and Sethi, R. (1986) From regular expressions to deterministic automata. *Theory of Computer Science Journal*, **48**, 117–126.

[26] Hopcroft, J. E., Ullman, J., and Motwani, R. (2000) *Introduction to automata theory, languages, and computation*. Addison-Wesley Longman Publishing, Boston, MA.

[27] Hopcroft, J. (1971) An $n \log n$ algorithm for minimizing states in a finite automaton. *Proceedings of Theory of machines and computations*, Haifa, Israel, 16-19 August, pp. 189–196. Academic Press, New York.

[28] Watson, B. and Daciuk, J. (2003) An efficient incremental DFA minimization algorithm. *Journal of Natural Language Engineering*, **9**, 49–64.

[29] Almeida, M., Moreira, N., and Reis, R. (2007) On the performance of automata minimization algorithms. Technical report. Universidade do Porto, Porto, Portugal.

[30] Snort rule set - http://www.snort.org/snort-rules/.

[31] Crypto++ library 5.6 - http://www.cryptopp.com/.

[32] FIPS (2009). 186-3. Digital signature standard (DSS).

[33] Beaver, D. (1995) Precomputing oblivious transfer. *Proceedings of CRYPT0 '95*, Santa Barbara, CA, 27-31 August, pp. 97–109. Springer-Verlag, London, UK.

[34] Kreuter, B., Shelat, A., and Shen, C. (2012) Billion-gate secure computation with malicious adversaries. *Proceedings of USENIX Security '12*, Bellevue, WA, 8-10 August, pp. 285–301. USENIX Association, Berkeley, CA.