SAX: A Privacy Preserving General Purpose Method applied to Detection of Intrusions
François Troussot, Pascal Poncelet, Florent Masseglia

To cite this version:
François Troussot, Pascal Poncelet, Florent Masseglia. SAX: A Privacy Preserving General Purpose Method applied to Detection of Intrusions. ACM First International Workshop on Privacy and Anonymity for Very Large Datasets, join with CIKM 09, Nov 2009, Hong Kong, China. pp.17-24. lirmm-00430646

HAL Id: lirmm-00430646
https://hal-lirmm.ccsd.cnrs.fr/lirmm-00430646
Submitted on 9 Nov 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
SAX: A Privacy Preserving General Purpose Method applied to Detection of Intrusions

ABSTRACT
To overcome the problem of attacks on networks, new Intrusion Detection System (IDS) approaches have been proposed in recent years. They consist in identifying signatures of known attacks to compare them to each request and determine whether it is an attack or not. However, these methods are set to default when the attack is unknown. However, it is frequent that an attack has already been detected by another organization and it would be useful to be able to benefit from this knowledge to enrich the database of signatures. Unfortunately, this information is not so easy to obtain. In fact, organizations do not necessarily want to spread the information that they have already faced this type of attack. In this paper, we propose a new approach to intrusion detection in a collaborative environment but by preserving the privacy of collaborative organizations. Our approach works for any signature even if it needs a complex program to be detected and insures that no information is disclosed on the content of any of the sites. For this purpose, we have developed a general method (SAX) that allows to compute any algorithm while preserving privacy of data and also of the program code which is computed.

Categories and Subject Descriptors
D.4.6 [Security and Protection]: Algorithms, Privacy

General Terms
Algorithms, Security

Keywords
Intrusion Detection, Attacks, Collaborative Organizations, Privacy

1. INTRODUCTION
The fast growing computational Grid environments has increased risk of attack and intrusion. Thus misuse detection has become a real concern for companies and organizations. Whereas earlier attacks focused on Web servers (often misconfigured or poorly maintained), the most recent ones take advantage of Security service and Web application weaknesses which become more vulnerable [4, 2]. To overcome this problem, new approaches called Intrusion Detection Systems (IDS) have been developed. Installed on networks, they aim to analyze traffic requests and detect malicious behavior (e.g. Prelude-IDS, Snort). They can be classified into two broad categories (e.g. [8, 9]): the Anomaly Detection Systems which attempt to detect attacks and the Abuse Detection Systems which detects unknown comportment so called abuse from a specification of allowed ones.

Recently approaches called Collaborative Intrusion Detection Systems (CIDS) (e.g. [1, 12, 5, 7, 11]) have been proposed. In comparison with isolated IDS, CIDS significantly improve time and efficiency of misuse detections by sharing information on attacks between distributed IDS from one or more organizations. The main principle of these approaches is to exchange information using peer to peer links. However, the exchanged information are mostly limited to IP addresses of requests (e.g. [1, 5, 7]) and consider that data can be freely exchanged among the peers. The last constraint is very strong: companies, for reasons of confidentiality, do not want to spread out that they were attacked and therefore are unwilling to give any information on it.

In this article we propose a secure collaborative detection approach, called SAX (Secure Algorithm Execution), which ensures that private data and programs will not be disclosed. Via our approach, any program from the various collaborative sites can be executed without disclosing any information from the local IDS to the outside. Collaborative Sites are free to work with signatures of attacks or non-attacks and may give information on the type of intrusion detected. Thus, when new request is checked, the response will be one of: it is an attack (with its type if available), it is a non-attack, or unknown (if none of the IDS data leads to a positive or negative conclusion). To our knowledge, very little studies are concerned with this topic of security in such collaborative environment. The only works [10, 7] consider both collaborative and security aspects. In its context, security mainly concerns information on IP addresses and ports. It uses Bloom’s filters to manage data exchanges. Our problem is different in that, we want to exchange data, i.e. more complex than IP addresses and ports. In fact we wants to be able to exchange and execute any algorithm of anomaly detection on the full request.

In previous studies, we focused on those anomalies whose
signature detection may be expressed as a regular expressions. But there is still a lot of cases that cannot be expressed like that and which need specific programs to be detected. Instead of handling them one by one, we decided to study a general purpose method able to execute any program while preserving privacy of data and also of the program code (once it has been translated into the adequate formalism). This method is described in this article.

The article is organized as follows. In section 2, we present the problem. An overview of our approach is given in section 3 and 4. The various algorithms are described in section 5. Finally section 6 concludes and presents various perspectives.

2. PROBLEM STATEMENT

$DB$ is a database such as $DB = DB_1 \cup DB_2 \ldots \cup DB_D$. Each database $DB_i$ is equivalent to a tuple $< id, S_{Alg} >$, where $id$ is the identifier of the database and $S_{Alg}$ is a set of anomaly detection programs expressed as a finite state automaton and initial program data. The details are described in section 3. The other part of input data of the program is the request string $R$ owned by the client site $S$. Each program will provide two kinds of value: a three states flag (True/False/Unknown) specifying whether the request is an attack or not or has not been identified at all and an integer (the type of the attack) when the request is effectively identified as an attack (True state).

Definition 1. Given a database $DB = DB_1 \cup DB_2 \ldots \cup DB_D$ and a request string $R$, the secure approach in such a collaborative environment consists in finding a program $Alg$ from $DB$ such that execution of $Alg$ upon $R$ gives a False or True result (identify an attack or a non-attack) while ensuring that the client site does not provide the request string $R$ to anyone and that none of the databases $DB_i$ provided any information from its content to anyone (programs and data).

3. THE SAX APPROACH

This section will provide an overview of the secure architecture SAX (Secure Algorithm eXecution). It is designed to answer the problem of privacy preserving in a collaborative environment. Inspired by the work of [6], this architecture offers the advantage of achieving the various operations while ensuring that neither party may have access to private data contained in the initial databases. In addition to the client site $S$ which is responsible to provide the request to be tested, the architecture requires four non-collaborative and semi honest sites [3]: they follow the protocol correctly, but are free to use the information they have collected during the execution of the protocol. These independent sites collect, store and evaluate information in a secure way. The different functions provided by these sites are:

- The Control Site CTRL: CTRL is used to rule the various operations needed to execute the program To do this, it interacts with the two non colluding sites $NC_1$ and $NC_2$.
- Non Colluding Sites $NC_1$ and $NC_2$: These two symmetric sites collects garbled data from all databases as well as the garbled request to be tested from $S$.

Under the control of CTRL and by interaction with $PS$, they perform several secure operations in order to ensure that none of them will be able to infer any of the intermediate results or the final result which is returned to site $S$.

- The Processing Site $PS$: This site is used both by $NC_1$ and $NC_2$ to process, in a secure way, the various needed operations. Like $NC_1$ and $NC_2$, $PS$ also cannot deduce any pertinent value of intermediate or final result from the data it processes.

Remark: The data kept by $NC_1$ ($\hat{b}$) and $NC_2$ ($\bar{a}$) are random data such that $\hat{b} \oplus \bar{a} = V$ where $V$ is the real value. But as $NC_1$ and $NC_2$ are non-colluding, the value of $V$ will never be known by any of the sites. (Details will be explained in the following sections).

3.1 Encoding of the program

To perform a secure computation of client site $S$ data by a program comming from an external site $DB$, (which may be anonymized), we consider the program as a deterministic automaton $< State, Trans, Init, Final >$ where State is the set of states of the automaton, Init is the initial state, Final is the set of final states and $Trans$ is the set of transitions. Each transition is a quadruplet $(S_{initial}, Action, S_{final}, V_{final})$ meaning that if the automaton is in state $S_{initial}$ and that $Action$ is performed and returns a boolean value then the automaton’s current state changes to $S_{final}$ if the boolean value is false or to $V_{final}$ if it is true. The automaton starts with state $Init$ and ends when reaching any of the final states from $Final$.

Remark: To ensure that computation is secure, none of the performed action shall return any data to the site $DB$, which gave the program to be computed. This means that only the client site $S$ gets results.

This encoding implies that all of the states (but final ones) are connected to two other states in the graph of the automaton (in fact $S_{final0}$ and $V_{final}$). These links are designated by $\oplus$ leading to $S_{final0}$ and $\odot$ leading to $V_{final}$.

Each action is in fact a list of 3-uptlets $(OpNum, Aff1, Aff2)$ where $OpNum$ is the number of the operation to be processed, and $Aff1$ and $Aff2$ are two random sequences of bits of of same length. Each 3-uptlet of the list performs the same action by using different operators which will be effectively executed by $NC_1$ and $NC_2$. There is a list to avoid $NC_1$ and $NC_2$ from deducing any link between the value of $OpNum$ and a “corresponding state” in CTRL.

The automaton is managed by CTRL while $NC_1$, $NC_2$ maintain the memory and performs the operations on the values through $PS$. If CTRL maintains the list of operation numbers to be performed for each node, the protocol to be performed when executing this operation is aimed by $NC_1$ and $NC_2$ and data used as input and output of this operation will only be kept (garbled) in $NC_1$ and $NC_2$. Thus CTRL will know nothing about the real operation that is performed and no more on the values used and produced by the operation. $NC_1$ and $NC_2$ also owns a list of Registers which contains address in the memory. These register will be used to perform indirect and based memory access. Instead of the data kept in their memory which are garbled, the real values of registers are known by both $NC_1$ and $NC_2$. 

After Performing the Action associated to a state, CTRL receives two bits (one from NC1 and one from NC2). If these two bits are identical then CTRL follows the link $\oplus$ otherwise it follows the link $\odot$.

For each operation OpNum of CTRL, NC1 and NC2 have a corresponding 5-uplet (Protos, NCA, Arg$^1$, Arg$^2$, Res, Incr) where Protos is a list of binary operations ($p$ elements), NCA is a random number ($1+m+n+p+q$ bits), Arg$^1$ is the list of values from the memory that may potentially be used as first argument of operation ($n$ elements), Arg$^2$ is similar to Arg$^1$ but for the second argument ($m$ elements), Res is the list of potential storages for the result in the memory ($q$ elements), and Incr is the list ($r$ elements) of couples (Register, IncrVal) giving increments to perform on registers at the end of the operation. In fact the Arg$^1$, Arg$^2$ and Res are couples (Position, Length) where Length is the number of bits of the value and Position is either a position in the memory or a a register number Reg and an Offset to perform indirect or based memory addressing.

**Remark:** there is several ways to implement an action that does only register increments, but it also exists a non operation prototype named $NOP^S$ designed for this purpose.

**Remark:** $Aff_1$, $Aff_2$ are such that $Aff = Aff_1 \oplus Aff_2 \oplus NCA \oplus NCA$ where $NCA$ is the part of $NCA$ owned by $NC_1$ and $NCA$ is the part owned by $NC_2$. $Aff$ is a list of boolean flags (true/false) that are used to specify which affectation shall be effectively done (see section 3.3). We will use the notations $\hat{Add}_1 = Aff_1 \oplus NCA$ owned by $NC_1$ and $\hat{Add}_2 = Aff_2 \oplus NCA$ owned by $NC_2$.

In fact $Aff_1 = \hat{Add}_1 \oplus RND_1$, $Aff_2 = \hat{Add}_2 \oplus RND_2$, $NCA = RND_1 \oplus RND_3$ is owned by $NC_1$ and $NCA = RND_2 \oplus RND_3$ is owned by $NC_2$ where $RND_1$, $RND_2$ and $RND_3$ are generated by $DB$, and $Aff = \hat{Add}_1 \oplus \hat{Add}_2$.

### 3.2 Initialization of the processus and global processing

The exchange of data between the different sites is done by using the secure method $SEND^S(V[\hat{v}][\hat{v}])$ which sends the vector of bits $V = \hat{v} \oplus \hat{v}$ to $NC_1$ and $NC_2$. It is defined in order to send $\hat{v}$ to $NC_1$ and $\hat{v}$ to $NC_2$ (or vice versa).

A random vector $R$ is used to garble transmitted data such that $\hat{v} = R$ and $\hat{v} = V \oplus R$ or vice versa. This method is used in particular to send the data from the databases $DB$, and to send the request from site $S$. Thus, the process described in figure 1 starts in the following way:

* First, the site $S$ sends its request to NC1 and NC2 using the $SEND^S$ method (See arrow number 1 in figure 1). More precisely, the request $R$ is taken in its boolean form: a vector of bits.

  * A random vector of bits $A_p$ is then generated with the same size as the request $R$ to compute the new vector $Z = A_R \oplus R$. $Z$ is sent to NC1 and $A_R$ to NC2 (or vice versa).

* Each database $DB_i$ sends the transition matrix of its automaton to CTRL (See arrows number 2) The size of the needed memory, the number of needed adress registers and their initial value are naturally sent to NC1 and NC2 as their real values are known by both sites. The initial value of the memory is sent to NC1 and NC2 by using $SEND^S$. $DB_i$ also sends the list of actions to be performed ($5$-uplets) to $NC_1$ and $NC_2$.

From this point, the computation of the request is done under the control of CTRL. It will ask NC1 and NC2 execute action (See arrow number 3).

* The action is effectively performed securely by NC1 and NC2 through PS (See arrow number 4 and 5).

* The result of the action is a boolean divided in two parts, one is owned by NC1 and the other by NC2 such that none are able to infer its real value. Both parts are then securely returned to CTRL (see arrows 5 and 6).

* While NC1 and NC2 are updating the value of the different registers as needed by the action, CTRL compares the two values he received and updates the current state of the automate in consequence.

* The process is repeated under control of CTRL unless the automaton moves into a final state.

* When entering a Final state, the result value previously stored by the program by executing the secure action $AFFECT^S$ is aggregated or sent to the client site $S$ by NC1 and NC2 (See arrow number 7) under the control of CTRL.
This choice depends of the configuration of the anonymity behaviour (See section 4).

### 3.3 Execution of action

When CTRL enter a non final state, he starts by randomly selecting one of the 3-uplets (OpNum, Affs1, Affs2) from the list associated to this state. It then generates a random number ROP of same length than Affs1 and Affs2.

Then CTRL sends \( \hat{\alpha}n = Affs2 \oplus ROP \) to NC1 and \( \hat{\alpha}n = Affs1 \oplus ROP \) to NC2.

NC1 (respectively NC2) computes the vectors of bits \( \hat{\alpha} = \hat{\alpha}R \oplus NC1 \) (respectively \( \hat{\alpha} = \hat{\alpha}R \oplus NC2 \)) and extract \( \hat{\beta} = \hat{\alpha}[0] \).

Now let \( \hat{\nu}^1 \) and \( \hat{\nu}^2 \) (respectively \( \hat{\nu}^1 \) and \( \hat{\nu}^2 \)) be two distinct vectors of bits of length greater or equal to the lengths on any of the values in \( \lambda_{\hat{\nu}^1} \) for \( \hat{\nu}^1 \) and \( \lambda_{\hat{\nu}^2} \) for \( \hat{\nu}^2 \) used to store the actual arguments of operations. We shall also use \( \hat{\kappa} \) and \( \hat{\mu} \) (respectively \( \hat{\kappa} \) and \( \hat{\mu} \)) to store the computed results of the operation.

Now NC1 (respectively NC2) computes the following secure instructions (See notations in section 5):

**Property 1**: This implementation allows to compute securely any program while encoded in the correct form.

**Property 2**: This secure execution of a program prohibits DB to know any of the client’s data.

**Property 3**: CTRL does not know any of the incoming and outgoing data and does not even know what processing has been performed.

**Property 4**: None of NC1, NC2 or PS can infer what processing has been performed.

**Property 5**: None of NC1, NC2 or PS can deduce any real value of the incoming or produced data.

### 3.4 Proof of privacy

**Property 1**: This implementation allows to compute securely any program while encoded in the correct form.

**Property 2**: This secure execution of a program prohibits DB to know any of the client’s data.

**Property 3**: CTRL does not know any of the incoming and outgoing data and does not even know what processing has been performed.

**Property 4**: None of NC1, NC2 or PS can infer what processing has been performed.

**Property 5**: None of NC1, NC2 or PS can deduce any real value of the incoming or produced data.

4. **ANONYMISATION**

There is two main ways to do anonymization: in buffering inputs programs issued by databases DB, and execute them randomly or in buffering results before sending them.
randomly to the client. The first one essentially introduce costs in term of space (we need to store the automata in CTRL and associated data in NC1 and NC2). The second one essentially introduce time costs (we may process other DB's programs while the desired result has already been obtained. And we will do that until the client gets this result). A third way is to mix the two approaches and adapt parameters (size of both buffers) to adjust anonymization process according to the needs and bearable costs.

As we are dealing with anomaly detection, the result is not so complex: an algorithm may detect an attack and then give the type of the detected attack, or identify a non attack request or may not be able to identify the request. In fact only the first positive or negative result has to be kept. We then only need to keep one result in NC1 and NC2 in an Accumulator. Any new result may be aggregated in the accumulator by computing securely

\textbf{if (Accumulator == unknown) then Accumulator = Result} which may be easily computed by using the operators \textit{COMPARE} and \textit{AFFECT}. At the origin, the accumulator is set securely with the value “unknown”. Thus, when anonymization is done by buffering results, we only get time costs, and the parameter is the number of aggregations to be performed before sending an aggregated result to the client.

5. **THE SECURE ALGORITHMS**

In this section, we present the various algorithms used in SAX approach. In order to simplify writing, we consider the following notations:

Let \((\chi | \bar{x}) \leftarrow h^S(\bar{y}_1 \cdots \bar{y}_n | \bar{z}_1 \cdots \bar{z}_n)\) be a tripartite computation of any function \(h^S\) between NC1, NC2 and PS where NC1 owns some of the inputs \(\bar{y}_1 \cdots \bar{y}_n\) and gets part of the result \(\bar{z}\) and similarly NC2 owns some of the entries \(\bar{z}_1 \cdots \bar{z}_n\) and gets part of the result \(\bar{x}\).

The final result is obtained by applying the binary operator XOR (\(\oplus\)) between \(\bar{x}\) and \(\bar{z}\). However, this does not mean that NC1 sends exactly \(\bar{y}_1 \cdots \bar{y}_n\) to PS an receives the result \(\bar{z}\) from PS. In fact, NC1 garbles its inputs \(\bar{y}_1 \cdots \bar{y}_n\) by adding random noise and gets \(\bar{y}_1 \cdots \bar{y}_n\) which are securely sent to PS. Similarly, NC2 sends its garbled inputs to PS.

At the end of the process, both sites receives a part of the garbled result from PS (respectively \(\bar{x}\) and \(\bar{z}\)). This intermediate result may now be used as input of further computation.

We will also use the following simplifications:

1. \(g^S(\bar{x}, \bar{y}) \leftrightarrow g^S(\bar{x} | \bar{y})\)
2. \(h^S()\) is a 2 argument function then \(h^S(\bar{x}_1, \cdots, \bar{x}_n | \bar{z}_1, \cdots, \bar{z}_n)\) will correspond to \(h^S(h^S(\cdots h^S(\bar{x}_1 | \bar{z}_1), \cdots | \bar{z}_n)) = (\bar{x}_1 | \bar{z}_1, \cdots | \bar{z}_n)\)

5.1 **The algorithm AFFECT\(^S\)**

The operator \textit{AFFECT}\(^S\) implements a conditional affectation of value to a variable. The non secure equivalent operator AFFECT could be specified as follows:

\[
\text{AFFECT } (\text{Var, Cond, Value}) \\
\iff \text{(cond) then Var } \leftarrow \text{Value; else Var } \leftarrow \text{Var; fi}
\]

The secure implementation of this operation (which is shown in algorithm 1 ensure that no one can infer whether the affection has effectively been done or not. That means that no one can predict the value stored in \(\text{Var}\) unless it already know the value of \(\text{Cond}\) (but NC1 and NC2 only knows random values).

The secure operator \textit{AFFECT}\(^S\) shall be used in the following way:

\[
\text{AFFECT}(\text{Var, Cond, Value})\rightarrow (\text{Var, Cond, Value})
\]

where \(\dagger\) are owned by NC1 and \(\ast\) are owned by NC2.

The implementation of \textit{AFFECT}\(^S\) require the secure operators \(V^S\) (shown in section 5.2), NC1 and NC2 therefore perform 34n aleatory bits, send 12n bits and receive 10n bits (including parameters). PS performs 12n binary operations, generates 3n + 1 aleatory bits, receives 12n bits (6n from NC1 and NC2 each) and sends 6n bits (3n to NC1 and NC2 each). Obviously this has to be compared with the length of inputs \((n \text{ bits})\).

**Algorithm 1:** Algorithm \textit{AFFECT}\(^S\)

\[
\text{Data: Var } = \text{Var}^\dagger \oplus \text{Var}^\ast \text{ of length } n \text{ is the value of the variable.}
\]

\[
\text{// Value } = \text{Var}^\dagger \oplus \text{Var}^\ast \text{ is the value to be conditionally affected (of length } n).\n\]

\[
\text{// Cond } = \text{Cond}^\dagger \oplus \text{Cond}^\ast \text{ is the bit condition true if the affection shall be done and false otherwise.}
\]

1. \(\forall k \in \mathbb{1}..n\) NC1, NC2 and PS compute

\[
(w_k | w_k^\dagger) \leftarrow \land^S(\text{Cond}^\dagger, \text{Var}^\dagger | \text{Cond}^\ast, \text{Var}^\ast);
\]

\[
\land^S(\neg \text{Cond}^\dagger, \text{Var}^\dagger | \text{Cond}^\ast, \text{Var}^\ast)
\]

2. NC1 and NC2 respectively computes \(\text{Var}^\dagger = \bar{w}^\dagger\) and \(\text{Var}^\ast = \bar{w}^\ast\).

**Property 6.** \textit{AFFECT}\(^S\) prohibits NC1 and NC2 to access the value stored in the variable \text{Var}. They even do not know if the value stored in \text{Var} has changed or not.

**Proof:** All the values stored in NC1 and NC2 are randomized. This count also for \text{Var}, \text{Cond} and \text{Value}. That means that none knows the real value stored in \text{Var} and \text{Value}. To know whether \text{Value} is affected to \text{Var} or not they shall not know the real value of \text{Cond} but as it is also randomized they cannot infer whether the value of \text{Var} has been changed or not. In any case, the new value of \text{Var} is \text{Var} and \text{Var} is new random values whatever the affection has been effective or not.

**Complexity:** The methods \(\land^S\) and \(\land^S\) are used 2n and \(n\) times respectively on one bit. By reusing the complexity of operators \(\land^S\) and \(\land^S\) (see section 5.2), NC1 and NC2 therefore perform 34n binary operations, generate 6n aleatory bits, send 12n bits and receive 10n bits (including parameters). PS performs 12n binary operations, generates 3n + 1 aleatory bits, receives 12n bits (6n from NC1 and NC2 each) and sends 6n bits (3n to NC1 and NC2 each). Obviously this has to be compared with the length of inputs (\(n\) bits).
$\Lambda^S$ and $\lor^S$

In this section, we define algorithms used to implement the secure operator $\Lambda^S$ and $\lor^S$. The basic principle of these algorithms is to add uniform random noise to the data which could be deleted from the final result.

The $\Lambda^S$ protocol begins with NC1 and NC2 who modify their data by doing XOR them with random values (see step 1 in algorithm ). NC1 and NC2 share these random values (also see step 1). Garbled data are then send to PS (step 2 and 3) which is now able to compute $\Lambda$ in a secure way (step 4). In fact, PS gets only garbled inputs indistinguishable from random and unrelated to each others and thus calculates random values from its point of view.

To avoid NC1 and NC2 from inferring the final result, it does XOR with random noise to the values it calculates before sending them back to NC1 and NC2 (step 5). Now NC1 and NC2 may both obtain their part of the final result by removing the random noise they added on step 1 (see step 6 and 7). The final result is obtained by computing:

$$A^R \oplus B^R = A'_{PS} \oplus (\tilde{x} \land R'_A) \oplus (\tilde{y} \land R_B) \oplus (\tilde{x} \land \tilde{y}) \oplus (R_B \land R'_A) \oplus B'_{PS} \oplus (\tilde{x} \land R'_A) \oplus (\tilde{y} \land R_A) \oplus (\tilde{x} \land \tilde{y}) \oplus (R_A \land R'_B)$$

where:

$$A'_{PS} \oplus B'_{PS} = (\tilde{x} \land R_B) \oplus (\tilde{y} \land R_A) \oplus (\tilde{x} \land R'_B) \oplus (\tilde{y} \land R'_A) \oplus (\tilde{x} \land \tilde{y}) \oplus (R_A \land R'_B) \oplus (R_B \land R'_A) \oplus R_{PS} \oplus R_{PS}.$$ 

Using the property of the XOR operator: $R \oplus R = 0$, we get the desired result: $A^R \oplus B^R = \tilde{x} \land \tilde{y} \oplus \tilde{x} \land \tilde{y} \oplus \tilde{x} \land \tilde{y}$.

The $\lor^S$ protocol is identical to the $\Lambda^S$ protocol except for the last two steps (steps 6 and 7) performed by NC1 and NC2. Thus we get the final result:

$$A^R \oplus B^R = \tilde{x} \lor \tilde{y} \oplus (\tilde{x} \land \tilde{y}) \oplus (\tilde{x} \land \tilde{y}) \oplus (\tilde{x} \land \tilde{y}) \oplus (R_B \land R'_A) \oplus B'_{PS} \oplus (\tilde{x} \land R'_A) \oplus (\tilde{y} \land R_A) \oplus (\tilde{x} \land \tilde{y}) \oplus (R_A \land R'_B).$$

This reduce to the desired result: $A^R \oplus B^R = \tilde{x} \lor \tilde{y} \oplus (\tilde{x} \land \tilde{y}) \oplus (\tilde{x} \land \tilde{y}) \oplus (\tilde{x} \land \tilde{y}) \oplus (\tilde{x} \land \tilde{y})$ which is a re-written form of $(\tilde{x} \oplus \tilde{y}) \lor (\tilde{y} \oplus \tilde{y})$.
5.3 The Algorithm COMPARE$^S$

Algorithm 4: The Algorithm COMPARE$^S$

Data: Hall part of $V$ and $W$ is owned by NC$_1$ and the other part is owned by NC$_2$

Result: $(\hat{n} | \hat{n})$ is such that $\hat{n} \oplus \hat{n} = 1$ if $V = W$ else 0
1. NC$_1$ computes $X \leftarrow \hat{v} \oplus \hat{w}$ where $X = (X_1, X_2, \ldots, X_l)$ and $l$ is the length of vector $V$ and $W$.
2. NC$_2$ computes $Y \leftarrow \hat{v} \oplus \hat{w}$ where $Y = (Y_1, Y_2, \ldots, Y_l)$. 
3. $(\hat{n}_1 | \hat{n}_2) \leftarrow OR^S(X_1, X_2, \ldots, X_l | Y_1, Y_2, \ldots, Y_l)$
4. There are two ways to get the result either NC$_1$, NC$_2$ perform $(\hat{n}_1 | \hat{n}_2) \leftarrow (\neg \hat{n}_1 | \hat{n}_2)$ or $(\hat{n}_1 | \hat{n}_2) \leftarrow (\hat{n}_1 | \neg \hat{n}_2)$.

Property 8. NC$_1$ and NC$_2$ gain no information of the real values which are compared and of the result of the comparison.

Proof: The input data sent to NC$_1$ and NC$_2$ are garbled with random values. Thus they cannot distinguish them from random values. In the same way, all values returned by $V^S$ are also garbled with unrelated random bits. Thus NC$_1$ and NC$_2$ only gets random values and then cannot infer the actual values of the inputs or results. If PS keeps history of intermediate results, it might deduce a part of the aleatory bits that were used to encode its results sent to NC$_1$ and NC$_2$. However, this gives no information of actual data.

Complexity: Length of bit vector is $l$: COMPARE$^S$ executes $l \oplus$ operations and $l - 1 \sqrt{S}$. Thus NC$_1$ and NC$_2$ compute $13l - 12$ binary operations (plus 1 negation in NC$_1$ or NC$_2$), generate $2l - 2$ aleatory bits, receive $4l - 3$ bits (including inputs) and send $5l - 4$ bits (including the result). On PS side, $PS$ computes $4l - 4$ binary operations, generates $l - 1$ aleatory bits, receives $4l - 4$ bits and sends $2l - 2$ bits.

5.4 The Algorithm $-S$ and NOP$^S$

Both are very simple to implement, by negating or not the value in NC$_1$ and/or NC$_2$: If none or both negate the value it is a NOP$^S$ operation but if only one does it, it is a $-S$ operation. In the point of view of NC$_1$ (respectively NC$_2$), there is no way to know which operation is really executed as none of them knows whether or not the other site negates or not its value. We also may implement these operation by involving $PS$ in the process to increase the confusion of NC$_1$ and NC$_2$ but it will consume extra time.

6. CONCLUSION

In this paper, we proposed a new approach to do any computation in a collaborative environment while preserving privacy and applied it to the intrusion detection problematic. Via our approach an application can use knowledge from foreign databases to identify whether a request corresponds to an attack or not. We have demonstrated that the proposed architecture ensured that it is impossible to identify which database has given the answer and that none of the internal components of the architecture can infer knowledge on the databases or on the request from the data they got. Our approach may also provide the type of the attack when they are specified in the databases. We also demonstrate that the cost of this secure computation is linear with the one of the same computation executed in a non-secure way. The approach also seems to be very scalable and one of our current focus is to study in which cases each site is required depending of the needs of programs code privacy or anonymity of sites. The second one concern the ability to detect when one or more of the non-colluding site does not conform to the edicted protocol. And another one concern the efficiency of operators (i.e introduce more powerful and efficient operators like arithmetic ones) and the definition of a programming language upon the SAX architecture to simplify the task of encoding.

7. REFERENCES