KTH Computer Science
and Communication

Privacy Preserving Collaborative Anomaly Detection Using Secure Multi-party Computation

ABU HAMED MOHAMMAD MISBAH UDDIN

Master's Thesis
Supervisors:

Dr. Peeter Laud, University of Tartu
Dan Bogdanov, Cybernetica AS
Dr. Mads Dam, The Royal Institute of Technology (KTH)

Abstract

The increasing volume of cyber attacks has become a major problem to the Internet world. Collaborative intrusion detection systems can help mitigating the problem to some extent. A mechanism to design such a system is aggregating attack traffic from victim organizations and applying anomaly detection systems on the aggregated data. To protect privacy of the users, the organizations should aggregate in a secure environment. Secure multiparty computation may be applied to such a task, but the general consensus is that the computation and communication overhead of such protocols makes them impractical for aggregation of large datasets.

In our work, we present a novel way to aggregate attack traffic in a privacy preserving manner using the primitives of secure multiparty computation. Specifically, we have devised a protocol independent algorithm that computes fast and secure set union and intersection. We implemented our algorithm in Sharemind, a fast privacy preserving virtual computer and support our claims by experimental results.

Contents

1 Introduction **1**

2 Background and Related Work **3**
2.1 Homomorphic Secret Sharing . . . 3
2.1.1 A trivial secret sharing scheme . . . 4
2.1.2 Shamir's Secret Sharing . . . 4
2.1.3 Homomorphism of Secret Sharing . . . 4
2.2 Secure Multiparty Computation . . . 5
2.3 Sharemind . . . 6
2.4 Sorting Networks . . . 8
2.5 Outlier Detection . . . 8
2.6 Secure Set Operation . . . 10

3 Algorithms **13**
3.1 Problem Statement . . . 13
3.2 Solution Formulation . . . 15
3.2.1 Privacy and Performance Goals . . . 17
3.3 Outlier Detection by LOCI . . . 18
3.4 Important Preliminaries for Privacy Preserving Algorithms . . . 19
3.4.1 Setup . . . 20
3.4.2 Notational Convention . . . 20
3.4.3 Oblivious Operations . . . 20
3.4.4 Data Structure . . . 21
3.5 Privacy Preserving Algorithms . . . 21
3.5.1 Secure Set Union Algorithm . . . 21
3.5.2 Secure Set Intersection Algorithm . . . 23
3.5.3 Oblivious Aggregation . . . 23
3.5.4 Oblivious Sorting . . . 26
3.5.5 Secure Set Reduction . . . 30
3.6 Merging the Pieces . . . 31
3.7 Privacy Level of the Algorithms . . . 32

4 Performance Evaluation **35**
4.1 Introduction . . . 35
4.1.1 On Performance Analysis of Anomaly Detection System . . . 35
4.1.2 On Performance Analysis of Privacy Preserving Algorithms . . . 35
4.2 Outlier Detection by LOCI . . . 36
4.2.1 Feature Construction . . . 36
4.2.2 Receiver Operating Characteristic Curve . . . 37

4.3 Evaluation of LOCI 38
4.4 Evaluation of the Privacy Preserving Algorithms 40
4.4.1 Test Setup 40
4.4.2 Performance Evaluations 41

5 Discussions **47**

Bibliography **49**

Appendix:

A Comparison Sequence Generators **53**
A.1 Python script for odd-even merge sort sequence generator 53
A.2 Python script for modified odd-even merge sorting network sequence generator 54

B Python Script for LOCI scheme **55**

C Privacy Preserving Algorithms **57**
C.1 SecreC code for 'oblivious aggregation algorithm' 57
C.2 SecreC code for 'vectorized oblivious aggregation algorithm' 57
C.3 SecreC code for 'oblivious bubble sorting network' 58
C.4 SecreC code for 'vectorized oblivious odd-even transposition sorting network' 59
C.5 SecreC code for 'vectorized oblivious odd-even merge sorting network' . . . 60
C.6 Secure Set Reduction Operation 62

Chapter 1

Introduction

Human communications have been revolutionized by the advent of the Internet. Public and private service providers are increasingly transferring their services online for improved availability and convenience. Low startup barriers, high consumer privacy and global outreach of this open network have made it possible to run completely online businesses and services.

Unfortunately, these benefits have attracted all kinds of adversaries. The number of cyber crimes are increasing at an alarming rate. According to MacAfee, "corporations around the world face millions of cyber-attacks everyday" [25]. Some of these attacks are orchestrated to harm a group of organizations e.g distributed network scanning. These attacks can be defined as multi domain attacks. Multi domain attacks are relatively more organized than the single domain attacks and often driven by political or ideological agendas. Usually, they start by surveying the vulnerability of the networks and then mount large scale attacks in the later stages. Therefore, it is important to detect the multi domain attacks at the early stages to prevent further damages.

Usually, traffic logs of a victim of multi domain attack contain only the partial visibility of the attacks. Therefore, when victim organizations apply intrusion detection system on the local traffic, they often fail to differentiate between multi and single domain attacks. To accurately detect multi domain attacks, the organizations need to aggregate traffic logs containing attacks samples, so that intrusion detection systems have the complete picture of the attacks.

But the organizations should not engage in such collaboration in non-privacy preserving environment. Non-privacy preserving aggregation of network logs reveals crucial information about the organizations, e.g. network topology, and crucial information about their clients, e.g. internet usage behavior. Disclosing such information may arise various kinds of complications such as legal ones.

Various methods exist for privacy preserving aggregation: one way hashing, commutative encryption, homomorphic encryption, secure multiparty computation (MPC) etc. Out of all, MPC protocols have been unpopular for aggregation of large datasets even though it provides a high level of privacy. MPC protocols suffer from high computation and communication overhead, which make them inapplicable in processing of large datasets.

In this project, we have focused on improving the speed of performance of MPC based aggregation. Particularly, we tried to use MPC framework for fast aggregation of large datasets. We have devised several algorithms that allow us to compute some secure set operations using MPC framework within an acceptable time. Specifically, we have designed privacy preserving aggregation, sorting and set reduction algorithm that can be assembled

to compute secure set union and intersection of multiple data sets. The algorithms are of generic nature. That means they can be implemented on any MPC framework provided that the framework supports some requirements. We state these requirements in the later discussions. We used these privacy preserving set operation protocols to build our privacy preserving anomaly detection system. Using the secure set operations protocols, we can aggregate the attack traffic from multiple organizations.

Our privacy preserving anomaly detection system allows us to aggregate the attack traffic from multiple organizations using the secure set operation protocols and to detect the multi domain attacks using an anomaly detection system on the aggregated data. In our project, we have chosen to use an outlier detection system named LOCI [31] as a network anomaly detection system. LOCI is a density based nearest neighbor algorithm that can be easily merged with the secure set operation algorithms to build the privacy preserving system.

We have implemented the building blocks of our system to evaluate their performance. The privacy preserving algorithms are implemented in Sharemind [9], a fast privacy preserving MPC framework. We have also implemented LOCI scheme to detect outliers from univariate attack features. The performance evaluation showed that our privacy preserving algorithms have successfully achieved our goal. The evaluation also helped us to choose the right configuration parameters for the LOCI scheme.

Rest of the report is organized as follows. In chapter two, we discuss the background and related work of our project. In chapter three, we discuss the problem formulation and design of the solution. In chapter four, we present the performance evaluation of our algorithms. We conclude in chapter five by discussing the achievements and limitations of our project.

Chapter 2

Background and Related Work

In this chapter, we discuss the theoretical concepts and related works necessary to understand our project. Our privacy preserving multi domain anomaly detection system is designed using the functionalities of secure multiparty computation (MPC). We assume that the MPC framework is based on secret sharing scheme and the MPC functionalities are implemented using share computing methods. Therefore, we start by discussing the basic concepts of secret sharing and secure multiparty computation in Sections 2.1 and 2.2.

We have devised some generic privacy preserving algorithms to design our anomaly detection system. The algorithms are portable to any MPC framework provided that the framework supports some requirements mentioned in chapter three. In this project, we implemented our scheme in the Sharemind framework [9]. We provide a brief description of the Sharemind framework in Section 2.3.

Our privacy preserving anomaly detection system needs to sort a sequence of values. Due to privacy constraint, the sorting operation should not depend on the value of the data. Sorting network is a data independent sorting model that can be easily incorporated to our system. Hence, we discuss the basic concepts of sorting network and some examples of sorting network in Section 2.4.

For anomaly detection, we have used the concept of outlier detection. Particularly we have used an outlier detection scheme named LOCI [31]. Therefore, in Section 2.5 we provide a brief discussion on outlier detection systems. Finally, we have assembled our privacy preserving algorithms to design secure set union and set intersection protocols to aggregate multi-domain attack data. In Section 2.6, we provide a brief overview of some earlier scientific researches in the area of secure set intersection computation.

2.1 Homomorphic Secret Sharing

Homomorphic secret sharing plays an important role in the project. Our secure set intersection protocol assumes private data is in the secret shared format and implements the subprotocols accordingly. In this section, we give a brief overview of secret sharing schemes and their homomorphisms.

Secret sharing is a useful method for protecting the privacy of sensitive data, independently proposed by Adi Shamir [34] and G.R. Blakley [7] in 1979. In simple terms, a secret sharing scheme includes a dealer who divides a secret value into n number of shares and distributes them to n parties. A predefined group of participants of size m ($m \leq n$) can cooperate to reconstruct the shares at any given time, while ensuring that no group of less than m can learn the secret. This group is called an access structure. We give a brief

overview of some secret sharing methods in the following section.

2.1.1 A trivial secret sharing scheme

A secret sharing scheme is considered trivial when all shares are necessary to recover the secret. This is also known as a (n, n) secret sharing protocol. A simple, yet efficient, trivial secret sharing scheme is given as follows. Let us assume there are n number of parties, each denoted by P_i, where $i = 1 \ldots n$. One of the parties acts as a dealer, who holds a secret S from a group G, and picks a set of random numbers s_i $(i = 1 \ldots n-1)$, where each s_i is as large as S. The dealer calculates $s_n = S - \sum s_i$ and distributes the shares by giving each s_i to P_i. When the secret needs to be reconstructed, all P_i send their shares s_i, to the dealer which adds them to obtain S. Since each share is random, they carry no information about the secret. An adversary cannot deduce the secret unless it manages to corrupt all of the parties. This scheme is, secure but not convenient, as it requires all parties to be present during the reconstruction phase.

2.1.2 Shamir's Secret Sharing

Shamir introduced the concept of secret sharing with the scheme described in [34]. It is a (t, n) threshold scheme, where only t out of n shares are needed to reconstruct the secret. Shamir's scheme is based on polynomial evaluation. Let S be a secret from some $\mathbf{Z}_P$, owned by the dealer. The dealer selects a random polynomial, $f(x) = f_0 + f_1 x + f_2 x^2 + \ldots + f_t x^{t-1}$, where $f(0) = S$. The dealer evaluates the polynomial $s_i = f(i)$, and gives each s_i to a party P_i, where $i = 1 \ldots n$. Therefore each party obtains a share of the secret.

The size of the access structure is t, which is same as the degree of the polynomial, where $t \leq n$. For reconstruction of the secret, the Lagrange interpolation formula [6] is used. Given t points, (x_i, y_i), $i = 1 \ldots t$, the polynomial can be regenerated by using the formula presented in Equation 2.1.

$$f(x) = \sum_{i=1}^{t} y_i \prod_{j=1, j \neq i}^{t} \frac{x - x_j}{x_i - x_j} \tag{2.1}$$

Thus, k parties can recreate the secret S with Equation 2.2.

$$f(x) = \sum_{i=1}^{t} f(i) \prod_{j=1, j \neq i}^{t} \frac{j}{j - i} \tag{2.2}$$

Any party holding a share can construct the secret by collaborating with $(t - 1)$ other parties, whether the dealer is included or not. As any subset of up to $(t-1)$ shares does not leak any information about the secret, it is secure in presence of computationally bounded adversaries. This scheme is more convenient than the trivial one, since the secret can be reconstructed by a subset of the parties.

2.1.3 Homomorphism of Secret Sharing

A homomorphism is a structure preserving map between two algebraic structures, such that for every kind of manipulation of the original data, there is a corresponding manipulation of the transformed data. In [11], Benaloh showed that Shamir [34], Blakley [7] and some other secret sharing schemes are homomorphic for some basic operation. Based on Benaloh's discussion, we define such homomorphic secret sharing as follows.

Let us assume two secrets, S and T, each being divided into n shares, denoted by $s_1 \dots s_n$ and $t_1 \dots t_n$. For any binary operations $\oplus$ and $\otimes$, a secret sharing scheme is $(\oplus, \otimes)$-homomorphic, if the reconstructed value of the shares $s_1 \otimes t_1 \dots s_n \otimes t_n$ is same as the value of $S \oplus T$.

We can also define a homomorphic secret sharing scheme with a constant as follows.

Let us assume a secret S and a constant C, where S is divided into n shares denoted by $s_1 \dots s_n$. For any binary operations $\oplus$ and $\otimes$, a secret sharing scheme is $(\oplus, \otimes)$-homomorphic, if the reconstructed value of the shares $s_1 \otimes C \dots s_n \otimes C$ is same as the value of $S \oplus C$.

The homomorphism properties of secret sharing schemes allows integrity preserving operations on a secret by computations on the shares. Benaloh successfully argued that it is much more secure to exchange and compute on the shares than to do the same on the secret itself, as the secret cannot be retrieved from less than t shares in (t, n) threshold schemes. Therefore, it is possible to efficiently implement a multi party computation protocol with homomorphic secret sharing scheme such as Shamir's.

2.2 Secure Multiparty Computation

Our privacy preserving scheme is built upon the secure multiparty computation (MPC) framework. The MPC problem was initially suggested by Andrew C. Yao in 1982, in terms of the millionaires' problem [38]. According to the millionaires' problem, Alice and Bob are two millionaires, who are trying to find out who is richer, without revealing information about their wealth. Yao proposed a two party protocol, that solves the problem with the given constraints. The solution of the millionaire problem lead to a generalization to multi-party protocols.

Based on Goldreich's discussion in [20], we give a brief overview of the basics of MPC as follows. MPC is an M party cryptographic protocol that maps M inputs to M outputs using a random process. The M inputs are local inputs of the parties and M outputs are their corresponding expected local outputs. The random process is the desired functionality of the protocol. The functionality allows distrustful parties to emulate by themselves the behavior of some external trusted third party who computes the outcome of the process using M inputs and returns each party the corresponding outputs.

To elaborate the emulation process, we introduce two distinct settings: real and ideal. The real setting is the actual execution of the protocol, whereas the ideal setting is an imaginary execution of the ideal protocol for computing desired functionality with the help of a trusted third party. The protocol is deemed secure, and hence emulates the ideal setting, if whatever the adversaries can feasibly obtain from the real setting can also be drawn from the ideal setting. Here, an adversary is a malicious party, whose objective is to prevent the users from achieving their goals by corrupting a set of parties. The security objectives are preservation of privacy of the local input by the parties and correctness of the local output by the honest parties.

The extent of emulation of the trusted third party by the mutually distrustful parties in MPC protocols varies according to adversary and communication channel models. Primarily, there are two main classes of adversary models: passive and active. In the passive adversary model, an adversary only gathers information from the corrupted parties without modifying their behavior. On the other hand, in the active adversary model, adversaries not

only read the messages, but also can modify the messages of the corrupted parties. Further, active and passive adversaries can be adaptive or non-adaptive. A non-adaptive adversary controls an arbitrary but fixed, set of corrupted parties before execution of the protocol, whereas an adaptive adversary can choose which party to corrupt during the execution of the protocol, based on the information gathered so far.

There are two basic models of communication. The first one is a cryptographic model, where an adversary is able to access all messages exchanged between the parties, and modify messages of the corrupted parties. The second one is an information-theoretic model where parties communicate with each other over pairwise secure channels. A secure channel prevents an adversary from reading any messages exchanged between the honest parties, even when the adversary is computationally unbounded.

Some models for general secure multi party protocol is given as follows:

- Models for passive and active adversary for any number of dishonest parties, assuming that adversary is non-adaptive, and computationally bounded and communication channels are cryptographically secure.

- Models for passive and active adversary that may control only a strict minority of the parties, assuming that adversary is adaptive, and computationally unbounded and communication channels are information theoretically secure.

The models can be easily applied to a reactive computational model, where a high level application interacts with the parties. The parties adaptively receive some inputs from the application and return the corresponding outputs. The application iterates this process reactively for some time. The outputs of each iteration may also depend on some global state, which may consist of the inputs and outputs of previous rounds. Therefore, the global state may be updated at each iteration. The state may only be partially known to individual parties and can be maintained by themselves in a secret sharing manner.

Some earlier protocols following this process use an unbounded number of iterations [32]. To achieve efficiency, some subsequent work obtained constant round protocols in some cases [5, 18, 27, 12]. Efficiency of the reactive computational protocoll can also be improved by using large sized packets exchanged during the execution of the protocol and optimizing the local computation time. In this project, we are not designing our own MPC protocol, but merely using such protocols as an aggregation tool. Hence, we will not provide any more discussion and refer to [20] for further reading.

2.3 Sharemind

Sharemind is a secure multi-party computation framework designed with a strong focus towards speed, scalability and ease of application development [9]. Sharemind uses secret sharing to split confidential information among several computing nodes denoted as miners. The data donors do not need to trust the miners provided that the number of corrupted miners colluding with each other is always less than a prescribed threshold, t, where the number of computing parties, $n > 3t$. The framework achieves provable security in semi honest model for information theoretically secure communication channels. In practice, models with three to five miner nodes in semi honest setting are common, where three miner models are comparatively communication-efficient. As Sharemind is strongly concentrated in improving processing efficiency in terms of speed, current implementation of the framework consists of three miner nodes.

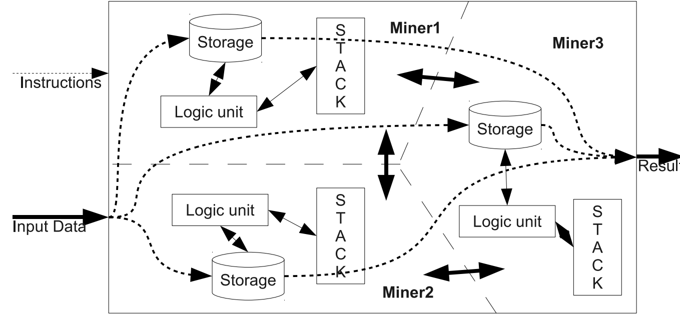

Figure 2.1. Deployment diagram of Sharemind (adapted from [9]).

Sharemind uses 32 bit integers as input to achieve efficiency in local computation time. Since Shamir secret sharing scheme does not work over 32 bit integers, an additive secret sharing scheme similar to the scheme presented in Section 2.1.1 is used by the framework. Figure 2.1 depicts the Sharemind framework. As said earlier, the framework consists of three miners. All secret values provided by the data donors are shared to the miners by the additive secret sharing given in Equation 2.3, where each miner, P_i, receives a share, s_i, of a secret s.

$$s_1 + s_2 + s_3 = s \, mod \, 2^{32} \tag{2.3}$$

The Sharemind protocols are implemented in a framework given in Figure 2.1. The framework consists of three computing parties defined as miners. The miners stores the shares of secret values in a secure storage. If the input values are not confidential then they are replicated to each miners in a publically accessible storage.

Sharemind implements its MPC functionality using constant round share computing protocols. The share computing functions provided by the framework are addition, multiplication and comparison operations. The miners are equipped with a runtime environment to implement the functions. Privacy preserving algorithms are designed using these share computing operations.

During data processing, the shares of the secret values are pushed to a secure stack and privacy preserving algorithm is executed on them. Intermediate and final result generated by each instruction of privacy preserving algorithm are also shares and stored in the same secure stack. When the execution is over, the miners collaborate with each other to reconstruct the final result. Sharemind provides a declassifying function for such reconstruction. Unless this function is explicitly invoked, the parties cannot reconstruct any secret value. The reconstructed result is publically accessible.

The share computation functions are complicated and hence kept hidden from the application developers. A controller library, provided by the framework, interfaces these operations to the application developers, so that they can be used without knowing their underlying details. For efficiency reasons, vectorized operations have been added to the framework, so that the same protocol can be executed in parallel with many inputs. This reduces the number of iteration significantly for larger datasets.

For privacy preserving application development, the framework provides a programming environment including an assembly and a high level language [21]. The framework also provides controller libraries for data distribution, program execution and performance analysis.

Several privacy preserving application have been designed using Sharemind. A Sharemind version of histogram computation and frequent itemset mining applications is presented in [8]. Other application examples have been suggested in [13, 35].

2.4 Sorting Networks

Our secure set intersection protocol requires sorting of data elements with the constraint that sorting has to be executed without looking into the comparison results. This constraint inhibits the use of optimal comparison based sorting. A sorting network is an alternate sorting model, that can solve the problem, while satisfying the given constraint.

A sorting network is a data-independent sorting technique, where the comparison sequence is generated in advance and executed regardless of the outcome of the past comparisons. A sorting network consists of two components: comparators and wires. A wire acts as a carrier of data element. The number of wires in a set is equal to the input size and each wire is initialized to one of the input values. The comparators act as operational units, each taking a pair of wires as an input, comparing their values and writing the outcome to the same wire pair.

We use the following definition to formally define a comparator. Let us assume a data sequence of size n is represented by $A^n = [a_0 \ldots a_n]$, whose indexes are represented by the set, $J = [0 \ldots n-1]$. A comparator is a mapping $(i, j) : A^n \rightarrow A^n, i, j \in J$ with

$$a_i = min(a_i, a_j)$$
$$a_j = max(a_i, a_j)$$
$$a_k = a_k \text{ for all } k \text{ with } k \neq i,\ k \neq j$$

A set of comparators is used to compose a stage. This composition, $S = [(i_1, j_1)...(i_k, j_k)]$, must be organized in such a way that each $i_r \neq j_s$, $i_r \neq i_s$ and $j_r \neq j_s$. A set of comparator stages is used to compose a comparator network. A sorting network is a comparator network that sorts a whole input sequence. Usually all comparators in a stage are independent. Therefore a sorting network may be parallelized.

Figure 2.2 illustrates three examples of sorting networks: bubble sort [24], odd-even transposition sort [24] and odd-even merge sort [4] for eight inputs. Figure 2.2(a) represents a bubble sort sorting network. It consists of $\{\frac{1}{2} \cdot n(n-1)\}$ comparators and $(2n-3)$ stages. Figure 2.2(b) shows odd-even transposition sort, which consists of the same number of comparators as the bubble sort but has fewer stages $(n-1)$. A odd-even transposition sort contains more comparators per stage, it is more parallelizable than bubble sort. The algorithms are easy to implement, but suffer from poor performance in case of large inputs.

The efficiency of a sorting network can be measured by its size, defined in terms of the total number of comparators in the network. Both bubble sort and odd-even transposition sort have the size $O(n^2/2)$. The best known sorting network, called an AKS Network [2], achieves the size, $O(n \log n)$ for n inputs, but has large linear constants hidden behind the O notation, which makes it impractical. Several practical sorting networks exist which bound the complexity to $O(n \log^2 n)$. Odd-even merge sort presented in Figure 2.2(c) is an example of such an algorithm. It achieves the size $n \log^2 n$ and $\log^2 n$ stages. For this reason, odd-even merge sort is both practical and highly parallelizable.

2.5 Outlier Detection

Anomaly detection is a crucial part of our project. We designed our anomaly detection system by a nearest neighbor (NN) based outlier detection scheme. In this section, we give

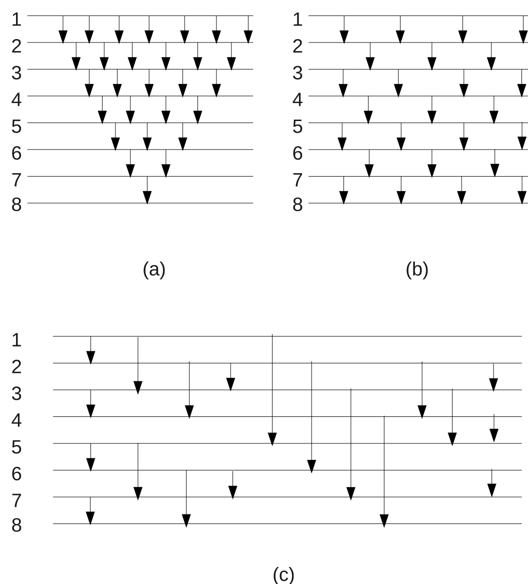

Figure 2.2. Sorting Networks

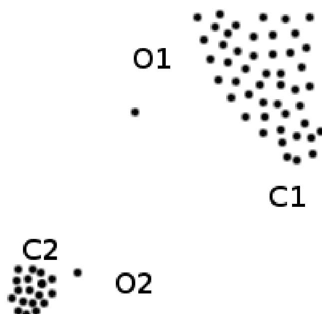

Figure 2.3. Outliers

a brief overview of nearest neighbor based outlier detection schemes. We start by giving a definition of outliers. In statistics, an outlier is an observation that is numerically distant from the rest of the data [3]. For example, let us consider the points in cluster $C1$ in Figure 2.3. Each point represents a statistical (bivariate) observation of some event. Point $O1$ is visibly distant from all other points in the cluster. Hence, point $O1$ is an outlier.

We can correlate some network anomalies (e.g. denial of service attack, IP sweep) with the outliers, as their traffic patterns are significantly different than that of the normal traffic. So, outlier detection systems can be used for anomaly detection. Anomaly events in networks are sporadic and it is difficult to train a machine to classify such events correctly. Therefore, unsupervised learning outlier detection systems such as nearest neighborhood schemes, are practical choice for detection of anomalies.

Traditional nearest neighborhood based techniques, such as k^{th} nearest neighborhood (KNN) scheme [23], detects outlier using the distance to the kth nearest neighbor from a

given point. According to the KNN scheme, a point, p, in a dataset, D, is an outlier, if at least T percentage of the points in D lie outside the distance (usually Euclidean) to the kth nearest neighbor of the point p (k-distance). However, if a given dataset has both sparse and dense regions, the scheme generates inaccurate results. This can be explained by Figure 2.3.

The dataset in Figure 2.3 has two regions of points marked as clusters $C1$ and $C2$. This points are statistical (bivariate) observations of some event. The density of $C1$ is sparse and $C2$ is dense. There are two points, $O1$ and $O2$, which are potential candidate outliers. The KNN scheme, defined above, uses the same threshold, T, to detect outliers for both of the clusters. It detects point $O1$ as an outlier for cluster $C1$. On the other hand, $C2$ is densely populated and there is a chance that number of points outside k-distance of $O2$ are below the threshold. Therefore, it may not detect $O2$ as an outlier, despite the fact that $O2$ is in fact an outlier for cluster $C2$.

To alleviate this problem, Breunig et al. [10] have come up with a more robust scheme: density based nearest neighbor technique. This class of outlier detection uses the concept of local neighborhood density. A local neighborhood is a circular area whose radius is determined by a local distance, usually a fraction of the k-distance. Local neighborhood density is the number neighbors (points) lying within the perimeter of the local neighborhood. Several methods exist in this class: LOF [10], COF [36] and LOCI [31]. These schemes define a degree of being an outlier of a point, rather than directly considering a point an outlier. Outliers are selected based on a cut-off threshold on the degree, which makes them more robust in scenarios, such as the one shown in Figure 2.3.

However for LOF and COF scheme, the cut-off threshold is manually selected, which is a weakness for automatic detection of outliers. This problem does not exist in LOCI, since it uses automatic data dictated cut-off threshold to determine whether a point is an outlier. We have chosen to use the density based KNN scheme LOCI as our anomaly detection system for the scheme's robustness.

2.6 Secure Set Operation

We have devised several privacy preserving subprotocols that can be assembled to design secure set operation protocol, specifically secure set union and intersection protocol. We use these set operation protocols to aggregate multi domain attack traffic. In this section, we discuss some earlier researches in this area. For our convenience, we divide the protocols into two classes:

1. Absolutely secure protocols that leak minimum amount of information. These protocols only disclose the result of the set operation and nothing else. A party can only learn information that can only be deduced from its input and the final output.

2. Less secure protocols that may reveal some more information other than the result of the set operation, e.g. input length of each party. This information may aid in revealing private inputs.

We discuss the following protocols based on our classification. In [1], Agrawal et el. proposed a two party secure set operation protocol using commutative encryption. The solution is easy to implement in databases and requires linear communication complexity. Another secure set operation protocol, based on additive homomorphic encryption and polynomial evaluation, is presented by Kissner et el. [22]. Both of the protocols employ

expensive cryptographic primitives, and hence suffer from weak performance in processing large datasets.

In [16], Emecki et el. a secure set operation protocol is defined using Shamir's secret sharing scheme [34]. The protocol replaces the trusted third party with a P2P network for query processing that works on the shared values of the private input. This protocol gives much better performance, compared to the first two protocols since it does not use the asymmetric cryptography. All of the above mentioned protocols leak the size of the input-set for each party, and hence belong to the second class defined above.

Naor et el. proposed a two party set operation protocol, using oblivious transfer and polynomial evaluation in [30]. The parties learn only the outcome of the operation and nothing else. Therefore, the protocol belongs to the first class. Unfortunately, the protocol is significantly slow compared to the previous protocols, which makes it impractical for large datasets. Our set operation protocol leaks minimum information (similar to first class), while achieving acceptable performance for large datasets.

Chapter 3

Algorithms

3.1 Problem Statement

Multi domain network attacks are increasing [26]. Such attacks include distributed denial of service (DDoS), spamming and scanning attacks to networks of multiple organizations. Often, traffic logs of a single victim contain only the partial evidence of the whole incident. When intrusiond detection system is applied on single domain traffic logs, they fail to grasp the complete visualization of the attack, which makes the detection result of the system less credible in case of multi domain attacks. Aggregating traffic logs of multiple victim organizations and applying intrusion detection system on them, can increase the strength of multidomain intrusion detection.

Anomaly detection is an intrusion detection technique, in which deviations from normal pattern in network traffic suggest malicious behavior. So, we define multi domain anomaly detection system as an intrusion detection system which detects multidomain network threats by detecting the deviation from normal pattern in aggregated traffic.

Aggregation of network traffic should be performed in privacy preserving manner. Traffic logs contain usage statistics of network users, and aggregating this kind of information openly may lead to legal problems. Secure multi party computation (MPC) is a cryptographic technique that allow information aggregation from multiple organizations with a high degree of privacy. In this project, we aim to design a privacy preserving collaborative anomaly detection system, that aggregates traffic logs from multiple organizations, using a MPC protocol, and detects multidomain network intrusions in the aggregated features, using an anomaly detection system. We formulate our solution with a use case. Then we expand our design to generalize the solution.

Let us assume we want to detect a multidomain IPsweep attack. An IPsweep is a surveillance sweep to determine the active hosts in a network. This information is useful for an attacker to orchestrate attacks and search for vulnerable machines. There are several methods to perform an IPsweep. The most common method is to send ICMP echo requests to every usable address in a subnet and wait to see which hosts respond. When a remote host performs an IPsweep in multiple subnets, then the attack may be classified as a multidomain sweep. The detection signature of an IPsweep attack is

If a remote host probes a high number of local hosts in a network by ICMP packets within a given period of time, then the host can be classified as an IPsweeper.

Usually, the time period is user defined but they can be as small as five to ten seconds. IPsweepers generally show recognizable abnormal patterns, and victim organizations can

hence easily detect them without any collaboration. Sometimes, attackers hide their actions by mounting small scale attacks. In this case, intrusion detection systems generally fail to detect the anomaly event [33]. However, if an attacker runs an undetectable small scale IPsweep over multiple networks, then counting the aggregate number of destination IP addresses in the ICMP traffic sent by the attacker to the networks may help to reveal the anomaly event.

Counting the frequency of an IPsweeping host over the domains reveals how many networks have been probed. If the frequency count is higher than the normal count (where boundary between normal and abnormal is set by a threshold), the host is classified as a multidomain IPsweeper. We revise the previously defined detection signature as follows:

If a remote host has a high number of destination IP addresses in the ICMP packets in a multidomain aggregated log within a defined interval and appears in multiple networks, then the host can be classified as a multi domain IP sweeper.

From now we will use the term frequency count to reflect the number appearence of a host in multiple networks. Implementing such signature to detect the attackers requires aggregation over multiple domains. We explain the aggregation scenario using Figure 3.1. Let us assume, there are five networks (A-E), that are victims of IPsweeping attack. Remote hosts sending ICMP packets to these networks can be divided into three classes:

- **Multi Domain Sweeper:** IP sweepers which have scanned multiple networks. The bold line from the multidomain sweeper to the networks represents a multidomain sweeping event.
- **Single Domain Sweeper:** IP sweepers which have scanned one or two networks only. The thin lines from the single domain attackers to the networks represent single domain sweeping events.
- **Benign Hosts:** Some remote normal hosts which sent random ICMP packets. The dotted line from the benign host to the networks reflects the random ICMP events.

The victim organizations want to detect the multidomain attackers. Therefore, they aggregate the destination and frequency count of each remote host, using a privacy preserving MPC system. Some or all of the victim parties can be the computing parties of the MPC system, which means they can run the MPC protocol by themselves. MPC system can also be implemented by some external third parties. In this scenerio, we assume that the data donors (the victims) and the computing parties are separate entity.

The dashed flow lines from the networks to the MPC framework show the input (containing the attack features) to the MPC protocol. The MPC protocol aggregates the features of the identical IP addresses in a privacy preserving manner. Such operation can be termed a secure set union operation. The bold dashed lines between the computing parties (P1-P3) in the MPC framework reflects the message flow in the set union computation. After the aggregation, the output is fed into an anomaly detection system. The thin line between the MPC framework and anomaly detection system shows such event. The anomaly detection system is executed in public. The reason is given in the following discussion:

A MPC protocol is suitable for simple privacy preserving data mining and counting operations. It is possible to implement simple anomaly detection systems (e.g k^{th} nearest neighborhood scheme [23]) in MPC frameworks. But this approach is not scalable. Implementing scalable anomaly detection systems, such as LOCI [31], in MPC framework is very difficult, and even if it is done, will give poor performance in terms of speed.

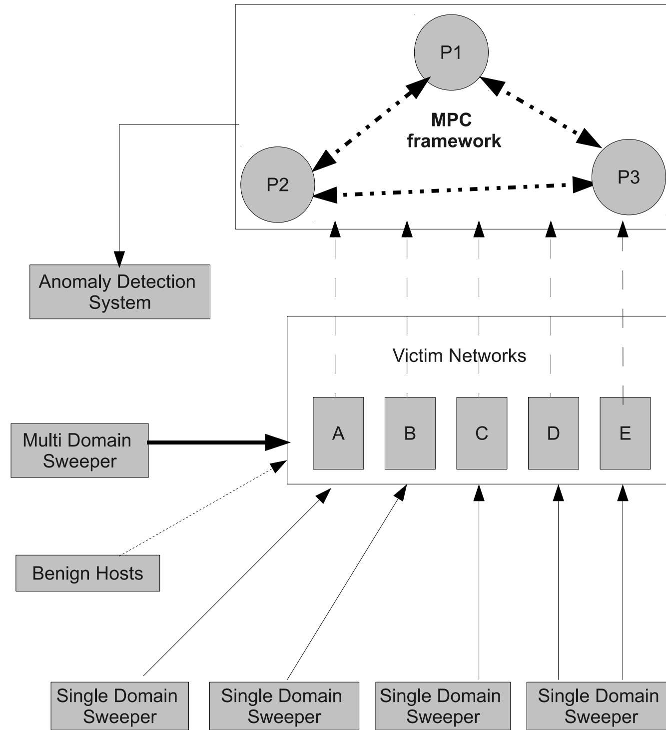

Figure 3.1. A scenerio for privacy preserving anomaly detection

We have designed our privacy preserving collaborative anomaly detection system using the two components: a MPC framework and an anomaly detection algorithm. We explain the design strategy in the next section.

3.2 Solution Formulation

As we said, our target is to build a secure collaborative intrusion detection system using a MPC framework and an anomaly detection algorithm. We skipped constructing our own MPC protocol. Rather, we design a generic algorithm, that can be executed using any secret sharing based MPC protocol. As for the anomaly detection system, we use a nearest neighbor based outlier detection algorithm, called LOCI. We build our system using the following scheme:

1. We perform a privacy preserving set union computation on the private multi-domain input, using a MPC based generic algorithm. The multi-domain input is a multiset formed by merging local inputs from each party, consisting of series of IP addresses, their destination, and frequency counts. This operation aggregates destination and frequency count of identical candidates in a privacy preserving manner.

2. We publish the aggregated destination counts of the union-set. We detect IPsweepers in a non-privacy preserving environment, by applying the LOCI scheme to the

published counts.

3. Using the list of detected sweepers from step 2 and a user defined threshold, we perform a set reduction operation (explained later) in privacy preserving manner on the private union-set (obtained from step 1) by the MPC protocol, to obtain a list of possible multidomain IP sweepers.

The threshold in step 3 reflects the minimum appearance count for a multi domain attacker. If the value of the threshold is equal to the number of victim organizations, the threshold based set reduction operation can be reduced to a set intersection operation. Thus, we are actually using a variant of a set intersection operation for network anomaly detection.

As outlined in the scheme, we need two generic MPC based algorithms: one for set union and one for set reduction. We define some secure set operations based on Kissner and Song [22] before describing the algorithms. We assume there are n parties, each having an input set denoted by S_i $(1 \geq i \geq N)$. Elements in the sets are private values of some common attributes.

Secure Set Union: *Secure set union is a multi party operation, which at the end allows all parties to learn the multiset union of the private sets* $(S_1 \cup S_2 \cup ... \cup S_N)$*, without gaining additional information.*

Secure Set Intersection: *Secure set intersection is a multi party operation, which at the end allows all parties to learn the multiset intersection of the private sets* $(S_1 \cap S_2 \cap ... \cap S_N)$*, without gaining additional information.*

Secure Over-threshold Set Union: *Secure over-threshold set union is a variant of the secure Set Intersection operation, which at the end allows all parties to learn elements that appear in the multiset union of the private sets* $(S_1 \cup S_2 \cup ... \cup S_N)$ *at least* t *(a threshold number) times, without gaining additional information.*

Therefore, the privacy preserving generic algorithms we need to develop are: secure set union and secure over-threshold set union. If the threshold in a privacy preserving over-threshold set union is equal to the number of parties, then the operation can be reduced to secure set intersection. Therefore, we use the term secure set intersection instead of secure over-threshold set union from now on.

We restate this scheme as Algorithm 3.2.1. To understand the notational convention and other related issues of this algorithm in detail, refer to the discussion in Section 3.4.

Algorithm 3.2.1: PRIVACY PRESERVING ANOMALY DETECTION($\|D\|$)

1. $\|A\| \leftarrow SecureSetUnion(\|D\|)$
2. $C \leftarrow PublishCount(\|A\|)$
3. $O \leftarrow LOCI(C)$
4. $\|F\| \leftarrow SecureSetIntersection(\|A\|, O, t)$
5. **output** $(\|F\|)$

Here, $\|D\|$ is a private multiset formed by merging the private local inputs from each party. $\|D\|$ is represented by a two dimensional array of 3 columns and N rows. Each row consists of an IP address, its destination and frequency count. $\|D\|$ is fed into the function defined in step 1 to compute privacy preserving a set union. The function aggregates the destination and frequency counts of the identical IP addresses, which are stored in the private vector $\|A\|$. In step 2, the column consisting of the aggregated destination counts is published in the public environment. In step 3, an anomaly detection scheme based on LOCI is applied to the published count. The function calculates openly and exposes the detected attackers (O).

In step 4, we performs privacy preserving set reduction operation on the union-set $\|A\|$ based on the outlier list O and predefined threshold T. The output is stored in private vector $\|F\|$, which is our desired multi-domain sweeper (outlier) list. If T is equal to the number of victim organizations, then the set reduction operation can also be termed a secure set intersection. This means, when T is equal to the number of victim parties, the detected attacker performed IPsweeping in all of the networks. The final step, declassifies the private result $\|F\|$ in a public environment. We will state our privacy goals for our project in the next section.

3.2.1 Privacy and Performance Goals

We will start by discussing our privacy goals. Our privacy preserving algorithms are generic and their privacy guarantees strongly depend on the MPC framework over which they are implemented. Therefore, unless the framework leaks information during computation, the privacy of the inputs are preserved. The only privacy concern for our scheme is whether the output of each step (steps of Algorithm 3.2.1) leaks sensitive information.

According to our previous discussion, we are aggregating a list 3-tuples: IP Address, Destination Count and Frequency Count. Our target is to aggregate the list and then apply anomaly detection system on the aggregated value. It would be ideal, if our scheme allows the computing parties to learn only the IP address and other attributes of the multidomain attackers and nothing else. Since our anomaly detection scheme is executed in non-privacy preserving manner, such privacy assurance is not possible, as a non-privacy preserving anomaly detection system needs to use aggregated counts (e.g. destination count) in public environment. But, even if the aggregated counts are published, they do not breach privacy of the clients or networks, unless the associated IP addresses are disclosed. In that perspective, we can state our following privacy goals of our scheme:

1. The scheme should not explicitly disclose any host IP address at the beginning or intermediate stages of the computation.

2. The scheme should not leak any information that may aid in disclosing the IP address of the hosts.

3. The scheme should only disclose the IP address of the detected multidomain attacker as the final result.

If our scheme satisfies the above mentioned goals then we can say that privacy goal is achieved. We give a detailed description of each step of Algorithm 3.2.1 in the following discussion. We start our discussion with the LOCI algorithm.

3.3 Outlier Detection by LOCI

LOCI is a density based nearest neighbor scheme for outlier detection, where it uses the concept of local neighborhood density for outlier detection. As said earlier, local neighborhood is a circular area. The radius of the circle is defined by a fraction of the distance (usually euclidean) to the k nearest neighbor. Local neighborhood density is defined by the number of neighbors within in the perimeter of the circle.

Like other density based schemes [10, 36], LOCI uses a term to define local neighborhood density - *Multi-granularity deviation factor (MDEF).* MDEF is defined as the relative deviation of local neighborhood density of a point from the average local neighborhood density in its k-neighborhood. Here, the k-neighborhood is a circle with radius k (distance to the k-nearest neighbor from the point). The local neighborhood is a smaller circle within the k-neighborhood having the radius αk (a fraction of the k-distance). Here, α is a user defined constant term that determines the fraction of k-distance.

If the local neighborhood of a point is closer to the average local neighborhood density, then its MDEF is closer to zero, and far from zero otherwise. Therefore, a small MDEF value reflects a smaller degree of outlierness, whereas a big MDEF value reflects a higher degree. We explain the formulation of the LOCI scheme with a scenerio (given in figure 3.1) in the following discussion. We start by formulating the MDEF.

$$MDEF = \frac{|\mu - n|}{\mu} = 1 - \frac{n}{\mu} \tag{3.1}$$

where, n = local neighborhood density of a point
μ = average local neighborhood density of the K-neighborhood of a point

The scheme also introduces a term called normalized standard deviation, denoted by σ_{MDEF}:

$$\sigma_{MDEF} = \frac{\sigma}{\mu} \tag{3.2}$$

where,σ = standard deviation of the local neighborhood densities in the k-neighborhood of a point
μ = average local neighborhood density of the K-neighborhood of a point

To automatically flag a point as an outlier, we use the inequality shown in equation 3.3.

$$MDEF > \lambda \cdot \sigma_{MDEF} \tag{3.3}$$

Here, λ is a constant to reflect the extent of deviation. Interestingly, equation 3.3 can be simplified easily. Using equation 3.1 on the left hand side and 3.2 on the right hand side of the equation 3.3, we get,

$$\frac{|\mu - n|}{\mu} > \lambda \cdot \frac{\sigma}{\mu}$$

$$\Rightarrow |\mu - n| > \lambda \cdot \sigma \tag{3.4}$$

Equation 3.4 reflects the classical definition of outliers presented in [14], which states, "an observation is an outlier if it is three standard deviation from its mean". Therefore, the LOCI scheme utilizes the classical definition of outliers in the context of the local

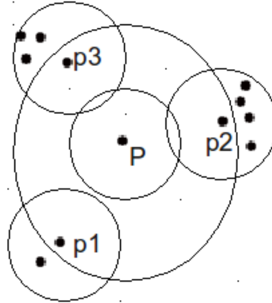

Figure 3.2. An outlier detection scenario

neighborhood density. Using the above definition and equation 3.4, we can set λ to three (which is same as [31] suggested).

We illustrate how the LOCI scheme flags a point as an outlier in the scenario explained by figure 3.2. We want to find out whether a point, denoted by P, is outlier. Points in the k-neighborhood of P are p1, p2, p3 and P itself. The smaller circles around the points denote local neighborhoods. Local neighborhood densities of the points are 1, 2, 5 and 4, respectively. Now we have, $n = 1$, $\mu = \frac{1+2+5+4}{4} = 3$ and $\sigma = \sqrt{\frac{(1-3)^2+(2-3)^2+(5-3)^2+(4-3)^2}{\mu}} = 1.12$. Applying equation 3.4 to these values, we find out that point P is not an outlier.

But the LOCI scheme presented in equation 3.4 has a problem. Mean and standard deviation are not robust statistics for outlier detection, as they are strongly influenced by the same outliers they are trying to detect. This is known as the masking affect, where the outliers hide their presence by changing the mean and standard deviation to such degree that they become undetectable [14].

One workaround to handle the masking problem is to choose a smaller value of λ (currently set to 3), the constant to reflect the extent of the deviation. Another way is choosing an alternate set of statistics that are less vulnerable to extreme values. The median and median absolute deviation (MAD) are such functions [37]. If we replace mean by median and standard deviation in equation 3.4 by MAD, we arrive at 3.5. However we cannot use the same value for λ (= 3). In such case, Davies and Gather mentioned in [14] to set $\lambda = 5.2$ as a general all purpose value.

$$median - n > \lambda \cdot MAD \tag{3.5}$$

Unfortunately, both of the solutions to minimize the masking problem have drawbacks. While they try to improve the success rate by increasing the number of true positives, they increase the number of false positives as well. In fact, choosing a right value for the configuration parameters for outlier detection is always a trade off between the true positive and false positive ratio. Therefore, different versions of LOCI is tried in this work, to pick the right scheme for our anomaly detector in chapter four.

3.4 Important Preliminaries for Privacy Preserving Algorithms

Let us begin by discussing some fundamental issues of privacy preserving algorithms.

3.4.1 Setup

We assume there is a MPC protocol and some computing parties. Each party contains some libraries to execute the MPC functionality (e.g. arithmetic and relational operation) and a secure data storage. Privacy of the input is preserved by secret sharing, and hence MPC functionalities are implemented by homomorphic share computing protocols. Data sources apply a common secret sharing algorithm on their input and distribute the shares to the computing parties. Data analysis algorithms are built using the MPC functionality and replicated to each party.

We assume the MPC protocol executes the algorithms in a constant number of rounds. In each round, the algorithm operates over some input shares and generates output shares that can be combined to form a result. The MPC protocol may exchange messages between the parties after each round, if the MPC functionality requires shares from other parties (e.g. multiplication operation [19, 15]). The exchange of the shares is performed in such a manner that no party can use the share to derive unauthorized knowledge. Without explicit disclosure, output shares are not recombined, and hence privacy is preserved. The algorithms use some public values, e.g. iteration limit and threshold value. Finally, we assume that the MPC framework supports vectorization of mathematical and relational operations (element wise operation over the whole array).

3.4.2 Notational Convention

Following conventions are used. All private values (secret shared inputs and the output shares generated from them) are enclosed within parallel lines, whereas the public counterparts are represented as it is. Variables are represented by lowercase and arrays by uppercase letters. For example, $\|b\|$ is a variable in secret shared form, $\|B\|$ is an array in secret shared form and c is a public variable.

The indexing operator ($'[..]'$) is used to select rows, columns or elements of an array. Array indexing starts from 1. For example, $\|B\|[1]$ means 1st element of the one dimensional private array B and C[1,2] represents an element from the 1st column and 2nd row of two dimensional public array C. A wildcard symbol is used to select all values in a row or a column. For example, $\|D\|[5,*]$ means the 5th column of private array D and $\|B\|[*,1]$ means the 1st row of private array B.

Let us define operations over all elements in an array as vector operations. Vector operations are done element wise, unless otherwise specified. For example, $\|B\| \cdot \|D\|$ is element wise multiplication of private vector, B and D. In some cases, a subset of private values are needed. This is represented as: $\|D\|[i:j]$, which is a subarray formed by the items from i to j of private vector D.

3.4.3 Oblivious Operations

Performing set union and intersection requires multiple numbers of comparison between the set elements. The secrecy requirements preclude us from using well known branching operation, such as:

```
If (predicate) Then
  (consequent)
Else
  (alternative)
```

There is an alternate and more efficient way to implement branching, using the boolean value generated by the comparison operation, which we call an oblivious operation. The following statement implements an oblivious way of determining the maximum of two numbers.

$$max = a \cdot (a \geq b) + b \cdot (1 - (a \geq b))$$

This statement evaluates $(a \geq b)$, and based on the evaluation, it either assigns a to max (if evaluation is TRUE) or assigns b to max (if evaluation is FALSE). If the evaluation result is privacy preserved (secret shared), then no party learns who is greater and what is assigned to max. We implement branching operations, using such oblivious operations, to design our privacy preserving algorithms.

3.4.4 Data Structure

Let us define the data structure of the input values. The input set from each data source contains a series of 3-tuples: $\{IP, Count, Freq\}$. The 1st tuple is the IP address of the remote host, the 2nd element is the feature count (e.g. destination count) of the remote host and the 3rd element is the frequency count of the remote host (potential attacker) in the set. Initially, the frequency count for each host is set to 1. The size of the input set may vary from one source to another. We set this length to a fixed size, so that none of the computing parties in the MPC protocol can learn about the input length from any data source. In case the number of remote hosts are less than the fixed length, empty slots is filled with dummy private IP addresses with zero count and frequency. The range of private IP addresses should be different for different domains.

When all of the sources have provided their input set to a data storage of the MPC protocol, the computing parties see a merged set of size N $(= n_1 + \ldots + n_m$, where the number of data sources are m). We call this merged set the multiset $\|D\|$, represented by a 2 dimensional array of N rows and 3 columns. The set operation algorithms use this multiset to generate the desired output.

3.5 Privacy Preserving Algorithms

We need two privacy preserving algorithms: secure set union and secure set intersection. To implement such schemes we need several subprotocols. In this section, we discuss the formulation of the algorithms using the subprotocols and schemes, to implement the subprotocols. We start by formulating the secure set union and intersection algorithms.

3.5.1 Secure Set Union Algorithm

In our problem statement, we discussed that the secure set union is the aggregation of attributes of common elements in a multiset, without compromising the privacy of the input values. As private inputs are secret (represented in secret shared form) and the computation results on input values are also secret, we cannot implement a straightforward compare and aggregate method. We resorted to a novel method of oblivious aggregation as presented in Algorithm 3.5.1.

(a) Party A		
IP	Cnt	Freq
IP1	10	1
IP2	20	1
IP3	30	1
IP4	40	1

(b) Party B		
IP	Cnt	Freq
IP3	10	1
IP4	20	1
IP5	30	1
IP6	40	1

Table 3.1. Input from party A and B

(a) Input Multiset		
IP	Cnt	Freq
IP1	10	1
IP2	20	1
IP3	30	1
IP4	40	1
IP3	10	1
IP4	20	1
IP5	30	1
IP6	40	1

(b) After step 1		
IP	Cnt	Freq
IP1	10	1
IP2	20	1
IP3	40	2
IP4	60	2
IP3	0	0
IP4	0	0
IP5	30	1
IP6	40	1

(c) After step 2		
IP	Cnt	Freq
IP3	0	0
IP4	0	0
IP1	10	1
IP2	20	1
IP5	50	1
IP6	60	1
IP3	40	2
IP4	60	2

(d) After step 4		
IP	Cnt	Freq
IP1	10	1
IP2	20	1
IP5	50	1
IP6	60	1
IP3	40	2
IP4	60	2

Table 3.2. An example of secure set union using Algorithm 3.5.1 and Table 3.1

Algorithm 3.5.1: SECURESETUNION($\|D\|$)

comment: $\|D\|$ is a private multiset input

1. $\|D\| \leftarrow \mathit{ObliviousAggregate}(\|D\|)$
2. $\|D\| \leftarrow \mathit{ObliviousSort}(\|D\|)$
3. $\|D\| \leftarrow \mathit{SecureSetReduction}(\|D\|, t)$

return $(\|D\|)$

We explain each step of the Algorithm 3.5.1 with a simple example. Let us assume there are two parties, A and B, who want to perform a secure set union on their traffic to detect anomalies. Input from each party contains 4 rows (see Table 3.1), which are merged by the MPC protocol as Table 3.2(a), assuming all of the values are actually stored in secret shared form. The secure set union function takes the merged table ($=\|D\|$) as an input.

In the first step, the algorithm obliviously aggregates attributes (count and frequency in this case) of identical IP addresses. The aggregated values are assigned to the first row from the rows with identical IP addresses, while the attributes of the rest of the rows are zeroed. The output of this step is presented in Table 3.2(b). In the second step, the rows are sorted obliviously in ascending order, according to the frequency count. After this step, IP addresses with zeroed attributes are pushed to the top of the list, while rest of the rows are placed after them in a sorted manner. This is shown in Table 3.2(c). Finally, rows with zeroed attributes are removed. This is done by counting the number of rows with zeroed attributes and then removing the corresponding number of times. The output of this step is given Table 3.2(d).

(a) Union-set		
IP	Cnt	Freq
IP1	10	1
IP2	20	1
IP5	50	1
IP6	60	1
IP3	40	2
IP4	60	2

(b) Outliers
Index
3
4
5
6

(c) After step 1		
IP	Cnt	Freq
IP5	50	1
IP6	60	1
IP3	40	2
IP4	60	2

(d) After step 2		
IP	Cnt	Freq
IP3	40	2
IP4	60	2

Table 3.3. An example of secure set intersection using Algorithm 3.5.2

3.5.2 Secure Set Intersection Algorithm

The secure set intersection algorithm detects the attackers that are found in some or all of the domains. This is achieved by performing a secure set reduction twice over the union-set obtained from the secure set union algorithm. The set reduction function in step 1, is an indexed set reduction operation, which is performed based on the output the anomaly detection system. This operation isolates the possible attackers. The function in step 2, is a secure threshold based set reduction operation, where the threshold value reflects the boundary of single domain and multidomain sweepers. This reduction rules out the single domain attackers. The scheme is given in Algorithm 3.5.2.

Algorithm 3.5.2: SECURESETINTERSECTION($\|D\|, O, t$)

comment: $\|D\|$ is a private union-set, O list of index of outliers, t is a threshold

1. $\|D\| \leftarrow IndexedSetReduction(\|D\|, O)$
2. $\|D\| \leftarrow SecureSetReduction(\|D\|, t)$

return ($\|D\|$)

We give a simple demonstration of this algorithm using the input from Table 3.1. Let us assume a union-set has been computed by Algorithm 3.5.1 and the result is given in Table 3.3(a) (which is same as 3.2(d)). Using the destination counts of this union-set, the outlier detector detected some outliers, the index of which is given in Table 3.3 (b). In the first step, the scheme deleted all the elements whose indexes are not in the outlier list. The result of this step is shown in Table 3.3(c). In the final step, the output from the previous step is reduced based on a threshold t, which is set to 2, in this example. The output is given in Table 3.3(d). This is our desired list of multidomain sweepers, which probed in networks of both Parties A and B. Now, we explain the subprotocols necessary to implement Algorithms 3.5.1 and 3.5.2.

3.5.3 Oblivious Aggregation

We start with the first subprotocol in the secure set union: Oblivious Aggregation. The objective of this algorithm is to aggregate features of identical elements in a privacy preserving manner. We designed a novel oblivious aggregation method, given in Algorithm 3.5.3.

Algorithm 3.5.3: OBLIVIOUSAGGREGATE($\|D\|$)

comment: $\|D\|$ is a seqeuence of 3-tuples: {IP, count, frequency}

1. $n \leftarrow length(\|D\|)$
2. **for** $i \leftarrow 1$ **to** n
3. **for** $j \leftarrow 1$ **to** n
4. $\|ip_1\|, \|ip_2\| \leftarrow \|D\|[1,j], \|D\|[1,j+1]$
5. $\|cnt_1\|, \|cnt_2\| \leftarrow \|D\|[2,j], \|D\|[2,j+1]$
6. $\|c\| \leftarrow (\|ip_1\| == \|ip_2\|)$
7. $\|D\|[2,i] \leftarrow (\|cnt_1\| + \|cnt_2\|) \cdot \|c\| + \|cnt_1\| \cdot (1 - \|c\|)$
8. $\|D\|[2,j] \leftarrow |cnt_2\| \cdot (1 - \|c\|)$

return ($\|D\|$)

The key lines in Algorithm 3.5.3 are steps 6 to 8, which implement the oblivious aggregation. Here, $\|D\|$ is the multiset input, where IPs are placed in column 1, counts are placed in column 2 and frequencies are placed in column 3. In step 6, we check equality of two IP addresses and store the result in the private boolean variable $\|c\|$. Step 7 and 8 implement oblivious aggregation based on this value.

Step 7 is formulated in such a manner, that based on the value of $\|c\|$ (1 or 0), it makes one part of the statement (either $\|cnt_1\| + \|cnt_2\|$ or $\|cnt_1\|$) zero. If two IP addresses are equal, then $\|c\|$ is 1, and the right part of step 8 ($\|cnt_1\|$) be zero, as it be multiplied with $(1 - \|c\| = 1 - 1 =)$ 0. Therefore, $\|D\|[2,i]$ is assigned the value: $\|cnt_1\| + \|cnt_2\|$, which is the aggregated destination count of two identical IPs. Similarly, in step 8, $\|D\|[2,i]$ is assigned the value: $\|cnt_2\| - \|cnt_2\|$, which is zero. On the other hand, if $\|c\| = 0$, then $\|D\|[2,i]$ is assigned $\|cnt_1\|$ and $\|D\|[2,j]$ is assigned $\|cnt_2\|$. This means, when two IP addresses are not equal, their respective destination count does not change.

After a complete iteration, we get an output as Table 3.2(b) for input as Table 3.2(a). This scheme is very slow since it contains N^2 comparisons and $8 \cdot N^2$ multiplications for N recordsets. Multiplication and comparison in MPC requires multiple rounds of local computation and message exchanges, and is expensive in terms of execution time. So, implementing such a high number of expensive operations definitely gives poor performance.

An easy solution to the problem is a complete vectorization of Algorithm [9]. Vectorization is a special case of parallelization, in which programs that by default perform one operation at a time on a single thread are modified to perform multiple operations simultaneously. This gives significant performance enhancement, even in current conventional computers. As we assumed our chosen MPC protocol to have vectorized versions of all mathematical and relational operations, we can design vectorized oblivious aggregation algorithms. Vectorized applications allow sending bigger chunks of messages during the protocol execution, which reduces the number of messages exchanged during the MPC protocol execution. Goldreich has pointed the issue of improving performance through bigger messages in [20]. That is why, we believe vectorization improves the execution speed of our algorithm significantly. Algorithm 3.5.4 presents a novel vectorized oblivious aggregation algorithm.

(a) Input Multiset		(b) Iteration 3 of Algorithm 3.5.4					
IP	Cnt	IP	I Step 4	C Step 5	Cnt Step 6	A Step 7-8	Cnt Step 9
IP1	10	IP1	IP3	0	10	0	10
IP2	20	IP2	IP3	0	20	0	20
IP3	30	IP3	IP3	1	40	0	40
IP4	40	IP4	IP3	0	40	0	40
IP3	10	IP3	IP3	1	10	10	0
IP4	20	IP4	IP3	0	20	0	20
IP5	30	IP5	IP3	0	30	0	30
IP6	40	IP6	IP3	0	40	0	40

Table 3.4. An iteration of vectorized oblivious aggregation by Algorithm 3.5.4

Algorithm 3.5.4: ObliviousAggregate($\|D\|$)

comment: $\|D\|$ is a seqeuence of 3-tuples: {IP, count, frequency}
1. $n \leftarrow length(\|D\|)$
2. $\|IP\|, \|CNT\| \leftarrow \|D\|[1,*], \|D\|[2,*]$
3. **for** $i \leftarrow 1$ **to** n
4. $\quad \|I\| \leftarrow \|IP\|[1]$
5. $\quad \|C\| \leftarrow (\|IP\| == \|I\|)$
6. $\quad \|CNT\|[i], \leftarrow colSum(\|CNTS\| \cdot \|C\|)$
7. $\quad \|A\| \leftarrow \|CNT\| \cdot \|C\|$
8. $\quad \|A\|[i] \leftarrow 0$
9. $\quad \|CNT\| \leftarrow \|CNT\| - \|A\|$
10. $\|D\|[2,*] \leftarrow \|CNT\|$
return $(\|D\|)$

The algorithm works with the same input as Algorithm 3.5.3, and the key concept is similar. If an IP address is identical to some other IP address in the list, then their feature counts are aggregated, othewise the feature counts are left as it is. We explain the steps of Algorithm 3.5.4 with a complete iteration. The input set is presented in Table 3.4(a). We observe the step by step execution of iteration number 3 in example.

In this iteration, the third IP address (IP3) is duplicated for element wise comparison (step 4). The output of this step is given in the 2nd column of Table 3.4(b). In step 5, we compare the IP vector and the duplicated vector, where the comparison generates a vector of zero (where elements are not equal) and one (where elements are equal). Observe the output of this step in the 3rd column of Table 3.4(b). By taking an element wise product of the resultant vector with the packet count and taking sum over the product vector, we obtain the aggregated packet count for IP3. The aggregated count is copied to the current index. This is shown in the 4th column of Table 3.4(b).

As the iteration covers all of the IP addresses, occurrence of the IP3 in further iterations is affected by this aggregated value. To keep only one correct instance of the aggregated value, we create a temporary vector $\|A\|$, where the counts are zeroed for the current index and indexes whose IP does not match with the current IP. This is shown in the 5th column of Table 3.4(b). By subtracting the temporary vector $\|A\|$ with the actual vectors $\|CNT\|$,

we obtain the desired vectors for each iteration. The output is given in the last column of Table 3.4(b).

Algorithms 3.5.3 and 3.5.4 only show how to aggregate the destination count. We can perform frequency count aggregation as well with some minor adjustments. Both algorithms use n^2 comparisons for aggregation, but the latter executes n comparisons in parallel, which in our opinion improves the speed of execution. We have verified our assumption about the performance improvement in chapter four.

3.5.4 Oblivious Sorting

From the oblivious aggregation, we obtain our desired aggregated values with some residues (nullified values). To obtain secure set union, we need to get rid of the residues. An easy way to eliminate null values is pushing the null values to the top, and then simply popping them. Pushing the null values to the top of a list can be achieved by sorting the values in ascending order.

Implementing sorting in MPC protocols is complicated because of the privacy constraints. Since we are not allowed to see the results of the comparisons, optimal comparison based sorting cannot be applied. An alternate way to sort a list is using a sorting network, as discussed in Section 2.4. Sorting networks have two important attributes:

1. A sorting network is data independent. It only depends on the input size. So, the comparison sequence can be generated in advance.
2. Comparisons can be divided into multiple rounds, where all of the comparisons in one round can be executed in parallel.

This first attribute makes sorting networks a perfect choice to implement sorting on MPC platforms. The second attribute allows them to be vectorized for performance speedup. To implement a sorting network in MPC, we need an oblivious swapping technique such as:

$$\begin{aligned} c &= (a \geq b) \\ a &= a \cdot (1 - c) + b \cdot c \\ b &= b \cdot (1 - c) + a \cdot c \end{aligned}$$

This is a novel approach to perform swapping operation between two elements. Algorithm 3.5.5 shows a simple implementation of Bubble sorting network, using the novel oblivious swapping. Each inner iteration moves the lowest value to the last position. The remaining $n - 1$ elements are sorted iteratively by applying the same procedure.

Now, this algorithm suffers from two problems. The algorithm uses $n^2/2$ comparisons and consists of $2n - 3$ stages. Hence, it is very inefficient for large datasets, and cannot be parallelized to improve the performance. To efficiently implement sorting operation, we need a sorting network that can be vectorized. Odd-even transposition sorting, discussed in chapter 2.4, is such a network. It uses a similar number of comparisons, but has fewer stages $(n - 1)$. We can implement a vectorized odd-even merge sort by executing the comparisons in a single stage in parallel.

Algorithm 3.5.6 gives a vectorized implementation of a odd-even transposition sorting network. It has $n/2$ iterations, where each iteration consists of two stages. In the first stage, $n/2$ comparisons are executed in parallel, while in the second stage, $(n - 2)/2$ comparisons are executed in parallel. We use a small example to demonstrate the algorithm. Let us assume we have a private list of four values, $\|L\| = \{40, 30, 20, 10\}$ and we want to sort it in ascending order, using Algorithm 3.5.6. According to the scheme, we need two iterations

with two stages for the given list. As we said earlier, we can generate the comparison sequence (per stage) in advance, as is given below.

Iteration 1

Stage 1: (1,2),(3,4)
Stage 2: (2,3)

Iteration 2

Stage 1: (1,2),(3,4)
Stage 2: (2,3)

Algorithm 3.5.5: OBLIVIOUSSORTING($\|L\|$)

comment: Bubble Sorting Network

comment: $\|L\|$ is a 1d array of private values

1. $n \leftarrow length(\|L\|)$
2. **for** $i \leftarrow n$ **to** 1
3. **for** $j \leftarrow 1$ **to** $i - 1$
4. $\|a\|, \|b\| \leftarrow \|L\|[j], \|L\|[j+1]$
5. $\|c\| \leftarrow (\|a\| \geq \|b\|)$
6. $\|L\|[1, j] \leftarrow \|a\| \cdot (1 - \|c\|) + \|b\| \cdot \|c\|$
7. $\|L\|[1, j+1] \leftarrow \|b\| \cdot (1 - \|c\|) + \|a\| \cdot \|c\|$

return $(\|L\|)$

Observe that iterations 1 and 2 are identical. In fact, each iteration generates identical comparison sequences for the input size. As shown in list mentioned above, each pair of indices, enclosed by brackets in each stage, is compared with each other in parallel. The parallel pairwise comparison is implemented with the help of a temporary vector, $\|O\|$ (Table 3.5(a)). $\|O\|$ is compared with vector $\|L\|$ and the comparison result is stored in a boolean vector, $\|C\|$ (step 5). The value of $\|C\|$ is modified a bit (step 6-7). Observe the value of $\|C\|$ in Table 3.5(a) after modification. Vectorized oblivious sorting is implemented in step 8, using a vectorized version of the oblivious swapping scheme described earlier. The result of this step is given in the 4th column of Table 3.5(a).

(a) Stage 1 Iteration 1			
$\|L\|$	$\|O\|$	$\|C\|$	$\|L\|$
	Step 4	Step 7	Step 8
40	30	1	30
30	40	1	40
20	10	1	10
10	20	1	20

(b) Stage 2 Iteration 1				
$\|M\|$	$\|P\|$	$\|D\|$	$\|M\|$	$\|L\|$
	Step 12	Step 15	Step 16	Step 17
-	-	-	-	30
40	10	1	10	10
10	40	1	40	40
-	-	-	-	20

(c) Stage 1 Iteration 2			
$\|L\|$	$\|O\|$	$\|C\|$	$\|L\|$
	Step 4	Step 7	Step 8
30	10	1	10
10	30	1	30
40	20	1	20
20	40	1	40

(d) Stage 2 Iteration 2				
$\|M\|$	$\|P\|$	$\|D\|$	$\|M\|$	$\|L\|$
	Step 12	Step 15	Step 16	Step 17
-	-	-	-	10
30	20	1	20	20
20	30	1	30	30
-	-	-	-	40

Table 3.5. Oblivious sorting by Algorithm 3.5.6

Algorithm 3.5.6: OBLIVIOUS SORTING($\|L\|$)

comment: Vectorized Odd-even transposition sorting network

comment: $\|L\|$ is a 1d array of private values

1. $n \leftarrow length(\|L\|)$
2. **for** $i \leftarrow 1$ **to** $n/2$
3.   **for** $i \leftarrow 1$ **to** $n/2$ *Step by* 2
4. $\|O\|[i], \|O\|[i+1] \leftarrow \|L\|[i+1], \|L\|[i]$
5. $\|C\| \leftarrow (\|L\| >= \|O\|)$
6.   **for** $i \leftarrow 1$ **to** n *Step by* 2
7. $\|C\|[i+1] \leftarrow \|C\|[i]$
8. $\|L\| \leftarrow \|L\| \cdot (1 - \|C\|) + \|C\| \cdot \|O\|$
9. $\|m\| \leftarrow n - 2$
10. $\|M\| \leftarrow \|L\|[i+1 : n-1]$
11.   **for** $i \leftarrow 1$ **to** $m/2$ *Step by* 2
12. $\|P\|[i], \|P\|[i+1] \leftarrow \|M\|[i+1], \|M\|[i]$
13. $\|D\| \leftarrow (\|M\| >= \|P\|)$
14.   **for** $i \leftarrow 1$ **to** m *Step by* 2
15. $\|D\|[i+1] \leftarrow \|D\|[i]$
16. $\|M\| \leftarrow \|M\| \cdot (1 - \|D\|) + \|D\| \cdot \|P\|$
17. $\|L\|[1 : N-1] \leftarrow \|M\|$

return $(\|L\|)$

Stage 2 operations are similar (Table 3.5(b)) and implemented on a subarray of $\|L\|$, which is formed by removing the top and bottom element. The subarray is denoted by $\|M\|$. The complete sorted list is obtained after all the iterations. Processing of each stage of each iteration is given in Table 3.5(a), (b), (c) and (d)).

Both of the algorithms mentioned above are derived from the schemes with $O(n^2)$ complexity [24]. This is a major drawback. We need a parallelizable network that has lower

complexity. Odd-even merge sort is such a network. Odd-even merge sort is a variant of the merge sort algorithm that merges two sorted equal length sequence into a completely sorted sequence. Its complexity is $O(n \log^2 n)$, which means we have to execute fewer comparisons. It uses $\log^2 n$ rounds which means it is parallelizable. The only problem with odd-even merge sort is that it is a recursive algorithm. It is very difficult to vectorize recursive functions. Converting a recursive algorithm to an iterative one is a sensible way to address this problem. We can perform this conversion in the following manner.

- Generate the comparison sequence for each round beforehand by executing the recursive algorithm.
- Sort iteratively by using the pre-generated comparisons in each sequence in parallel.

We implemented a recursive odd-even merge sort to generate a comparison sequence per stage for input size of 2^n (see Appendix A.1). An example of the comparison seuqence for a list of four elements is given as follows:

1. Iteration 1: (1,2),(3,4)
2. Iteration 2: (1,3),(2,4)
3. Iteration 3: (2,3)

Observe that the number of sequences is one less than for the odd-even transposition sort. This difference increases as the input size increases. The sequence and the private input is fed into the function defined in Algorithm 3.5.7, which is almost identical to Algorithm 3.5.6 with some minor changes. Since the number of comparison sequences of this scheme is smaller than the previous scheme, we expect improved performance. We test the performance of all of the sorting algorithm in chapter four.

Algorithm 3.5.7: OBLIVIOUSSORTING($\|L\|, SEQ$)

---

comment: Vectorized Odd-even merge sorting network

comment: $\|L\|$ is a 1d array of private values

comment: SEQ is a comparison sequence generated by odd-even mergesorting network

1. $n \leftarrow length(\|L\|)$
2. **for** $i \leftarrow 1$ **to** $len(SEQ)/n$
3.   **for** $j \leftarrow 1$ **to** n
4. $\|O\|[j] \leftarrow \|L\|[SEQ[i, j]]$
5. $\|C\| = (\|L\| >= \|O\|)$
6.   **for** $j \leftarrow 1$ **to** n
7. $\|C\|[SEQ[i, j]] = \|C\|[j]$
8. $\|L\| = \|L\| \cdot (1 - \|C\|) + \|O\| \cdot \|C\|$

return $(\|L\|)$

We can use any of these algorithms to perform oblivious sorting. The sorting algorithms are designed to operate on one dimensional arrays. Two dimensional array sorting can implemented by some simple modifications.

3.5.5 Secure Set Reduction

After oblivious aggregation, we obtain a list, where values in some rows are zero and some are non zero. The rows with zero values are residues of the aggregation and they must be removed in a privacy preserving manner. Since, we sort the list after aggregation, the rows with zero values are pushed to the top and the rest of the rows are placed at the bottom. This allows us to securely eliminate the rows, as we can remove from the top without having to access the values. We define this process as secure set reduction.

However, to implement secure set intersection (as given in Algorithm 3.5.2), we need to implement two more row elimination process, one of which is a threshold based set reduction. The threshold based set reduction operation is actually a variation of the secure set reduction operation defined above. The threshold based elimination operation removes rows, whose feature counts are less than or equal to a threshold. Since the list is sorted (due to oblivious sorting in the secure set union), rows with feature counts below or equal to the threshold, are in the top of the list. Therefore, we can eliminate these rows by removing from top. This is similar to the previous set reduction process.

Algorithm 3.5.8: SECURESETREDUCTION($\|D\|, t$)

---

comment: $\|D\|$ is a union-set obtained from Algorithm 3.5.1, t = [0, t_1]

1. $\|F\| \leftarrow \|D\|[3, *]$
2. $\|C\| \leftarrow (\|F\| == t)$
3. $n \leftarrow vecSum(\|C\|)$
4. **for** $i \leftarrow 1$ **to** n
5. $vecRemove(D, 1)$

return $(\|D\|)$

We can generalize the row elimination process for both of the cases. The generic algorithm is a threshold based set reduction scheme, where in the first case the threshold t, is set to zero and in the second case, is set to some user defined limit t_1. We present the generic scheme in Algorithm 3.5.8. Here, the scheme determines the number of rows, with values equal to or less than t, in step 3. The number is denoted by n. Then, the algorithm uses a pop function (vecRemove()) to remove n records from the top of the list.

The other row elimination process in secure set intersection, is a simple index based elimination. The indexes are generated by the outlier detection scheme, which indicates some IP address in the private list as anomalous. We eliminate the items, that are not represented in this outlier list. An index based row elimination process is presented in Algorithm 3.5.9. The algorithm is self explanatory.

Algorithm 3.5.9: INDEXEDSETREDUCTION($\|D\|, O$)

---

comment: $\|D\|$ is a union-set obtained from Algorithm 3.5.1, O is an index list

1. $\|n\| \leftarrow length(\|D\|)$
2. **for** $i \leftarrow 1$ **to** n
3. $if(i \neq O)$
4. $vecRemove(D, i)$

return $(\|D\|)$

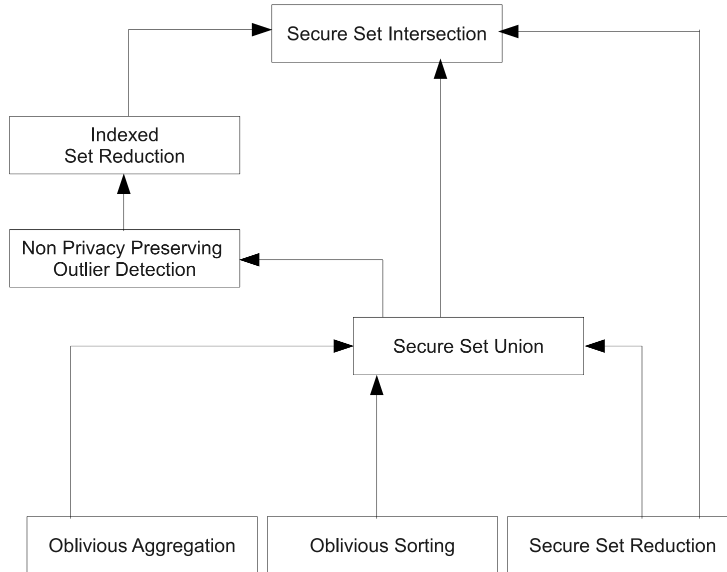

Figure 3.3. Merging the pieces

3.6 Merging the Pieces

A privacy preserving collaborative anomaly detection system can be implemented by merging the subprotocols defined in Sections 3.3 and 3.5. To demonstrate the merging process, we use Figure 3.3. We start from the bottom. We form the secure set union by merging the oblivious aggregation, oblivious sorting and secure set reduction algorithms, defined in Sections 3.5.3, 3.5.4 and 3.5.5 respectively. We can choose any of the aggregation algorithms from Algorithms 3.5.3 and 3.5.4 for the oblivious aggregation operation. Similarly, we can choose any of the sorting algorithms from Algorithms 3.5.5, 3.5.6 and 3.5.7, for the oblivious sorting operation. Finally, we have to choose algorithm 3.5.8 with threshold t, set to zero, for secure set reduction.

The anomaly detection mechanism use a part of the aggregated output of the secure set union. The outliers revealed by the scheme acts as an input for the indexed set reduction operation (given in Algorithm 3.5.9). The secure set intersection is implemented by merging this indexed set reduction algorithm with the secure set union and secure set reduction algorithms. The secure set reduction algorithm sets the threshold t, to some user defined limit t_1.

As there are several schemes for some of the subprotocols, we have several possible combinations for the privacy preserving anomaly detection scheme. The combination varies in performance in terms of execution time. We have to evaluate the performance of different combinations, to find the best performing scheme.

Our scheme detects multi domain IP sweepers. With some simple modifications, we can implement the scheme to detect any kind of signature based multi domain anomalies, i.e. distributed denial of service (DDoS), spamming, multidomain port sweeping etc. Observe the Table 3.2(a). We can extend Table 3.2(a) as Table 3.6, to detect a wide range of multi domain attacks. This table contains feature counts for n number of anomalies and m IP

Input Multiset						
IP	Feature 1	Feature 2	...	...	Feature n	Freq
IP1	10	15	...	...	40	1
IP2	23	20	...	...	80	1
IP3	30	50	...	...	10	1
...	...	...	...	...	...	...
IPm	55	35	...	...	37	1

Table 3.6. Input table for generalized anomaly detection system

addresses. The last column is reserved for the frequency counts and rest of the columns contain some anomaly event feature count, e.g. features for Neptune attack[1], Mail Bombing[2] and Port Sweeping[3].

To generalize the detection scheme, we compute the set union of Table 3.6, using secure set union algorithm (defined in Algorithm 3.5.1) and then implement the anomaly detection scheme (given in Section 3.3) to identify the possible attackers for each of the threats considered. We perform secure set intersection separately for each column as list of outliers and the user defined threshold may be different for different kind of anomalies.

In the next chapter, we present the performance evaluation of the algorithms.

3.7 Privacy Level of the Algorithms

In an ideal MPC setting, each party learns only the output of the computation and nothing else. Even though we wanted to achieve that level of privacy, some part our scheme given in Algorithm 3.2.1, failed to reach that. Following list shows the information leaked by the different parts of our scheme.

- Oblivious aggregation algorithms: These algorithms (Algorithms 3.5.3 and 3.5.4) do not leak any information unless the MPC framework on which the algorithms are implemented leaks information.
- Oblivious sorting algorithms: These algorithms (Algorithms 3.5.5, 3.5.6 and 3.5.7) do not leak any information unless the MPC framework on which the algorithms are implemented leaks information.
- Secure set reduction algorithms: These algorithms (Algorithms 3.5.8 and 3.5.9) leak information. Secure set reduction algorithm (Algorithms 3.5.8) leaks the number of the inputs which are lower and higher than the threshold value. Indexed set reduction algorithm (Algorithms 3.5.9) leaks the number of the outlier and non-outlier points and ofcourse the index of the outlier points, where each index represents an outlier point from a given set.

Therefore, we observe that, when we merge the pieces to design a privacy preserving anomaly detection system, we do not leak much information during the aggregation phase (the first two steps in secure set union Algorithm 3.5.1) but we leak information during the set reduction phase (the last step in secure set union Algorithm 3.5.1 and both steps

[1] DDoS attack by SYN flooding. See '*http://tools.ietf.org/html/rfc*4987' for details.
[2] DDoS attack to mail servers by high number of emails. See '*www.cert.org/tech_tips/*' for details.
[3] Scanning to find open ports in a host. See '*www.linuxjournal.com/article/*4234' for details.

in secure set intersection Algorithm 3.5.2). Moreover, as we are implementing the outlier detection scheme in public, we are disclosing the packet counts as well. So from an execution of the whole scheme, each party learns the following information:

- Before execution: Each parties' own input.
- After secure set union operation (after step 1): The number of residues (values that are equal to zero).
- Before outlier detection (after step 2): The aggregated packet counts.
- After outlier detection (after step 3): The index of the IP addresses that are potential outliers, the number of outliers and the number of non-outliers.
- After secure set intersection operation (after step 4): The number of single and multi domain attackers.
- When execution ends (after step 5): IP address and other information of the potential multidomain attackers.

Having this information, a dishonest party may have a leverage to declassify the private input from other parties. However none of the leaked information aid in leaking the IP addresses of the single domain attackers and non-attackers. Morever, as the list is shuffled by the sorting operation, it is no possible for an adversary to disclose classified values by correlation. Therefore, we can say that even if our scheme leaks some information and fails to achieve the desired level of privacy but it still protects privacy of the inputs to an acceptable level.

Chapter 4

Performance Evaluation

4.1 Introduction

In the last chapter, we designed schemes for our privacy preserving anomaly detection system. We evaluate the performance of these schemes in this chapter. The algorithms, can be divided into two classes.

1. Schemes for network anomaly detection.

2. Schemes for privacy preserving aggregation.

We explain the performance analysis of these schemes in the following discussion.

4.1.1 On Performance Analysis of Anomaly Detection System

We built our anomaly detection system using LOCI [31], which is an outlier detection algorithm. The performance of LOCI should be evaluated based on its costs and benefits. By benefit, we mean how many network anomalies are detected correctly from a given set of anomalies. By cost, we mean the number of normal hosts it wrongly detects as anomalies. The cost-benefit analysis of anomaly detection system can be done by the *receiver operating characteristic (ROC)* curve [17].

We have argued in Section 3.3 that changing configuration parameters influences the accuracy of the LOCI scheme. Therefore, we experiment with different configurations of LOCI on the same set of test cases to evaluate the performance of each one. The evaluation helps us to choose the right configuration parameters for LOCI.

4.1.2 On Performance Analysis of Privacy Preserving Algorithms

We implemented several privacy preserving algorithms, as given in following list:

- Secure Set Union
  - Oblivious Aggregation (Algorithm 3.5.3 and 3.5.4)
  - Oblivious Sorting (Algorithm 3.5.5, 3.5.6 and 3.5.7)
  - Secure Set Reduction (Algorithm 3.5.8)
- Secure Set Intersection

We built these algorithms in such a way, that they can be ported to any secure multi party computation (MPC) framework, provided that the framework satisfies some requirements (see Section 3.4.1 for the list of requirements). MPC protocols have been known for being slow in performing data analysis on large datasets. We therefore designed our algorithms with efficiency in mind. Performance of MPC based algorithms is influenced by the following classes of operations [9]:

1. **Local Operations:** Operations that have one round of local computations, e.g. addition operation.
2. **Multi party operations:** Operations that have multiple rounds of local computations with messages exchanged among the parties between the rounds, e.g. comparison operations.

Execution times of the local operations are significantly lower compared to that of the multi party operations. So, if a scheme is built using both kinds of operations, the multi party operations have significantly more impact than the local operations on the overall performance. The Subprotocols for our privacy preserving algorithms use both kinds of operations to some extent. For example, the oblivious sorting algorithm uses more multi party operations than the set reduction algorithm. Out of the three algorithms used by the secure set union protocol, two of them (oblivious aggregation and sorting algorithm) rely heavily on the multi party operations. Therefore, the performance of the secure set union protocol is strongly influenced by these algorithms.

On the other hand, the set intersection protocol performs the set reduction operation over the output of the set union protocol. The set reduction algorithm has a few multi party operations, and hence execute faster compared to the algorithms used in the secure set union protocol. We believe performance evaluation of secure set union gives an idea of the performance of secure set intersection. So we skipped the performance evaluation of secure set intersection.

4.2 Outlier Detection by LOCI

In this section, we compare the performance of different configurations of the LOCI scheme. We have limited our experiments scope to IP Sweepers. Before discussing the test evaluation, we give a brief background on the test data and performance analysis tools.

4.2.1 Feature Construction

We applied our anomaly detection system to the 1998 and 1999 Darpa intrusion detection evaluation data [29, 28]. Both of the sets contain several weeks of simulated training and testing data to measure the probability of detection and false alarm rate of intrusion detection systems. The training set contains normal traffic samples, as well as several weeks of (simulated) network based attacks. These network based attacks are various kinds of scanning (probing) attacks[1], denial of service attacks[2], user to root attacks[3] and remote to local attacks[4].

[1] Automatically scan a network of computers to gather information or find known vulnerabilities

[2] An attack in which the attacker makes some computing or memory resource too busy or too full to handle legitimate requests, or denies legitimate users access to a machine

[3] A class of exploit in which the attacker starts out with access to a normal user account on the system and is able to exploit some vulnerability to gain root access to the system

[4] An attack in which an attacker who can send packets to a machine over a network but does not have an account on that machine exploits some vulnerability to gain local access as a user of that machine

LOCI is an unsupervised machine learning scheme for outlier detection. So, we avoided training LOCI using the normal data samples. We applied our system on the labeled training set, containing attack traces. As we have limited our detection to IPsweeps, we have used only those samples that contained traces of such attacks. For our tests, we picked five sets of IPsweep data.

We have stated the detection signature of IPsweep attack in the Section 3.1. We have written a small python script to extract the features of the attack from the traffic logs (tcpdump files[5]). The feature set generated by the script is a series of 2-tuples: {IP address, Destination Count}. The first element represents the remote hosts which send the ICMP packets, and the second element represents the number of destinations probed by the remote hosts.

We obtained five feature sets from the five attack logs. As the samples are labeled, the attackers are known. This helps us to evaluate the probability of detection and false alarm by the anomaly detection system. Out of these five sets, only one set contains three IP sweepers and rest of them contain one sweeper each. Having a small number of attackers is troublesome for anomaly detection system evaluation, so we merged the five sets. The merged set consists of 7 unique IPsweepers and 97 unique normal hosts. We test different configurations of LOCI on this merged set and evaluate their performance.

4.2.2 Receiver Operating Characteristic Curve

We analyze the performance of LOCI in terms of true and false positives. We represent these terms as benefit and cost of the anomaly detection system. Following definitions are used to define these terms.

- ***Detected Attacker Set:*** *A set of observations which are flagged as anomalies by an outlier detection scheme.*
- ***Actual Attacker Set:*** *A set of observations which are actual anomalies.*
- ***Actual Non-attacker Set:*** *A set of benign observations.*

We introduce some more terms using these sets.

- ***Actual True Positives:*** *The size of the actual attacker set.*
- ***Actual False Positives:*** *The size of the actual non-attacker set.*
- ***Calculated True Positives:*** *The size of the set intersection of the detected attacker set and the actual attacker set.*
- ***Calculated False Positives:*** *The size of the set intersection of the detected attacker set and the actual non-attacker set.*

Now, the benefit of an anomaly detection system is defined in terms of true positive ratio (TPR), which is a ratio between the calculated and the actual true positives. The cost is defined in terms of false positives ratio (FPR), which is a ratio between the calculated and the actual false positives.

We can evaluate the performance of the LOCI based anomaly detection system in terms of benefit and cost using a ROC curve. A ROC curve is a two dimensional curve, where the

[5] http://www.tcpdump.org/

X axis represents FPR and Y axis represents TPR. Thus, it depicts the trade-off between cost and benefit. The space enclosed by the X and Y axis, is called a ROC space. A pair (FPR, TPR) represents a point in the ROC space. For multiple executions of the system, we have a set of value-pair represented as a set of points in the ROC space.

A diagonal line is drawn from the bottom left corner to the top right corner. This line is defined as the line of no discrimination. Points above the line are considered good and points below the lines are considered bad. An anomaly detection system is considered good if most of the points fall above the line. Similarly, the system is considered bad if a significant amount of points fall below the line.

4.3 Evaluation of LOCI

In Section 3.3, we have simplified the LOCI scheme to the formula given in Equation 3.4. The equation used $\lambda = 3$, according to [31]. We have argued that this equation is affected by the masking effect and we have suggested using a smaller value for λ to minimize the problem. We have also suggested using an alternate form of the formula, as given in Equation 3.5, using $\lambda = 5.2$.

In this section, we evaluate the performance of the LOCI scheme for both formulas, for different value of λ, to find out the best parameter configuration for LOCI. Following list shows these cases:

- Configuration 1: LOCI scheme using the mean function, the standard deviation function (Eq. 3.4) and $\lambda = 3$.
- Configuration 2: LOCI scheme using the mean function, the standard deviation function (Eq. 3.4) and $\lambda = 2$.
- Configuration 3: LOCI scheme using the median function, the median absolute deviation function (Eq. 3.5) and $\lambda = 5.2$.

There are other parameters to consider: the value of k that determines the k-neighborhood and the fraction α that determines the local neighborhood. The parameters are explained in Section 3.3. The best choice of k for k^{th} nearest neighborhood schemes (kNN) is data dependent, but the general consensus is that larger values of k reduce the effect of noise on the classification. On the other hand, a high value for k makes boundaries between classes less distinct. We picked five random, high odd values for k: 69, 75, 79, 85 and 89. We execute the three configurations of LOCI scheme, using $\alpha = 0.125$, for all five values of k.

We have written a python script that implements the LOCI scheme for the configurations mentioned above (see Appendix B). The script is designed to detect IPsweeping hosts from a given sample of features. The method of feature extraction is described in Section 4.2.1. The script generates the following outputs for the values of k, defined above, for all three configurations:

1. Detected outliers
2. FPR
3. TPR

The first output is a list of IP addresses marked as outliers. This list may contain false positives. The second and third outputs are calculated based on the definitions given in

	Configuration 1		Configuration 2		Configuration 3	
k	TPR(%)	FPR(%)	TPR(%)	FPR(%)	TPR(%)	FPR(%)
69	57.1	3.1	71.4	7.2	85.7	13.4
75	71.4	8.3	71.4	12.4	100.0	16.5
79	71.4	11.3	85.7	11.3	100.0	17.5
85	71.4	15.5	85.7	14.4	100.0	19.6
89	85.7	17.5	100.0	18.6	100.0	22.7

Table 4.1. Performance of the LOCI scheme for different configurations and k values

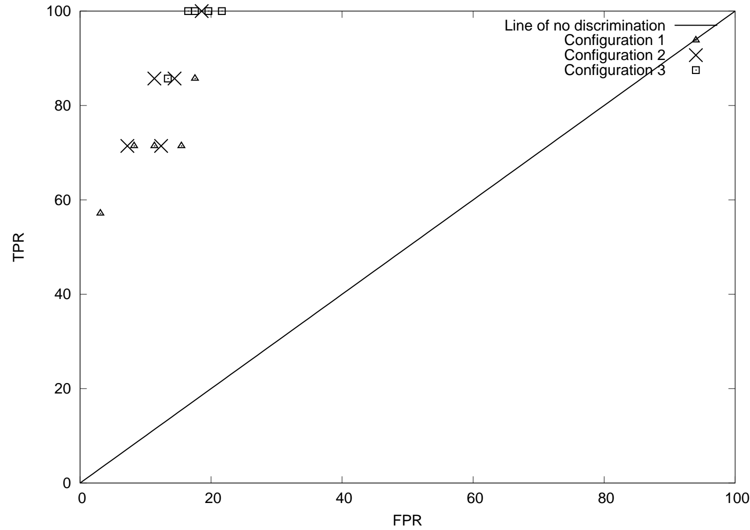

Figure 4.1. ROC curve for different configuration of the LOCI scheme, for different k values

Section 4.2.2, using the detected outlier list. The results are presented in Table 4.1. We observe that the, TPR and FPR increase with the increment of k for all three configuration, with some minor exceptions. Configuration 1 achieves lower success rate with the benefit of lower amount of false positives. On the other hand, configuration 3 achieves higher success rate with the cost of high number of false positives. The performance of configuration 2, falls between configurations 1 and 2. Configuration 3 achieved 100 percent success in most of the cases, whereas, configuration 2 achieved that only once. Configuration 1 failed to achieve 100 percent success rate.

The values are plotted on a ROC curve, given in Figure 4.1. All of the points in the ROC space lie above the line of no discrimination. Most of the points generated by configuration 3 are placed in the absolute top position of the space, which is a very desirable outcome for an anomaly detection system. Based on the analysis of Table 4.1 and Figure 4.1, we can be say that for anomaly detection, configuration 3 is the best choice.

Now, we analyze the detected outliers. For this particular test, we have chosen configuration 3, with k set to 45. The test outcome is given in Figure 4.2. We create an imaginary line called line of outlierness. Points above the line are the detected outliers and those below the line are the hosts detected as benign. The scheme detected eleven points as outliers. But the total number of actual IPsweepers are seven. The list of the outliers revealed,

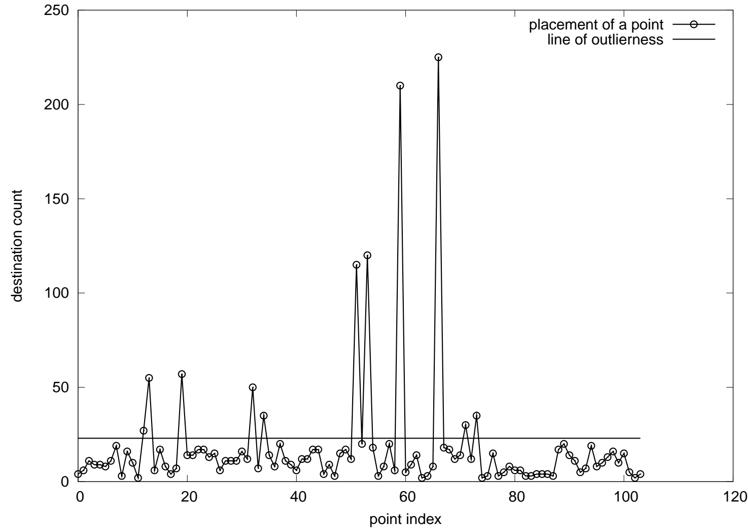

Figure 4.2. Outliers detected by the LOCI scheme for configuration 3, $k = 45$

that all seven of the IPsweepers were detected successfully, which means the rest are false positives. Actually, the four points, that are very close to the line of outlierness, are the false positives. It is possible to eliminate these false positives by tweaking the configuration parameters, but at the risk of lower success rate.

4.4 Evaluation of the Privacy Preserving Algorithms

We designed our privacy preserving algorithms with a strong focus towards minimizing the processing delay. In this section, we test the performance of the privacy preserving algorithms in terms of execution time.

4.4.1 Test Setup

We designed several generic privacy preserving algorithms that can be executed using any MPC protocol, that supports several requirements, described in section 3.4.1. Sharemind is a MPC platform that supports all of the requirements. Although we are supposed to evaluate the performance of the schemes in a networked environment of three computing nodes, we skip that for initial testing. We run each of our algorithms in a setting where all three computing parties are emulated in a local machine. In this setting, the computing parties are emulated by a separate process in a single computer, and MPC protocol is implemented by message exchange between the processes. This approach has two drawbacks.

- It violates the privacy requirement.
- Processing time evaluation becomes flawed.

Since, privacy of the algorithms strongly depends on the security of the MPC platform, we assume that in an actual implementation, the privacy is preserved by the MPC platform used, so we can ignore the first drawback at the moment. For the second problem, only

testing in a networked environment gives accurate statistics to evaluate the performance of the schemes, but due to lack of time, we had to resort to more simplified testing. Nevertheless, we believe the tests performed give sufficient indication of the relative performance of the algorithms.

For the implementation, we used a Dell Vostro 1320 model notebook, with a 2.66 GHz Core 2 Duo CPU and a 4 GB RAM. Sharemind runs in several operating systems but we have chosen to run it on Ubuntu 9.10. We have configured[6] the localized version of Sharemind with the 'SecreC' [21] compiler, where all three computing parties are emulated on a single machine.

We have written our privacy preserving algorithms in *SecreC*. We have also written a controller program, using the Sharemind controller library that stores private values in the secure storage of the computing parties in the additive secret shared format. We have written another controller program that executes the *SecreC* codes and returns execution result. Any party, who wants to obtain the output of the privacy preserving algorithms, have to use this program. This prevents the modification of privacy preserving algorithms by the dishonest parties.

The algorithms are data independent. Therefore, it is not necessary to execute the algorithms with real data for the purpose of performance evaluation. Therefore, we use fabricated data for our tests. The fabricated data is a list of two tuple: $\{$ ID, CNT$\}$. Both of the elements are some randomly generated integers from a given range. The first element represents an IP address and the second element represents a feature count of that IP address.

4.4.2 Performance Evaluations

We have argued in Section 4.1.2, why we test only some of the algorithms. We execute the following tests:

1. Testing performance of the different schemes for oblivious aggregation (Algorithms 3.5.3 and 3.5.4).

2. Testing performance of the different schemes for oblivious sorting (Algorithms 3.5.5, 3.5.6, 3.5.7).

3. Testing performance of the following assembly configurations for the secure set union protocol:

   a) Vectorized oblivious aggregation algorithm (Algorithm 3.5.4), vectorized odd-even merge sorting networks (Algorithm 3.5.7) and secure set reduction algorithm (Algorithm 3.5.8).

   b) Vectorized oblivious aggregation algorithm (Algorithm 3.5.4), vectorized odd-even transposition sorting network (Algorithm 3.5.6) and secure set reduction algorithm (Algorithm 3.5.8).

For the first test, we executed the SecreC code of both versions of the oblivious aggregation algorithms given in Algorithm 3.5.3 and 3.5.4 (see Appendix C.1 and C.2). The latter algorithm is the vectorization of the former one. For performance analysis, we measured the processing time of each execution. The time is calculated by taking the difference between the time when the execution starts and the time when the execution ends. Table 4.2 shows the processing time for different sized input for each algorithm given in minutes.

[6] A configuration tutorial is given in http://research.cyber.ee/sharemind/docs/sharemind-1.9/

	Execution Time (minutes)	
Input size	Oblivious Aggregation (Algorithm 3.5.3)	Vectorized Oblivious Aggregation (Algorithm 3.5.4)
32	12.2	0.2
64	39.6	0.4
128	142.3	1.2
256	321.8	3.4
512	-	12.4
1024	-	39.4
2048	-	85.2

Table 4.2. Performance of oblivious aggregation algorithms

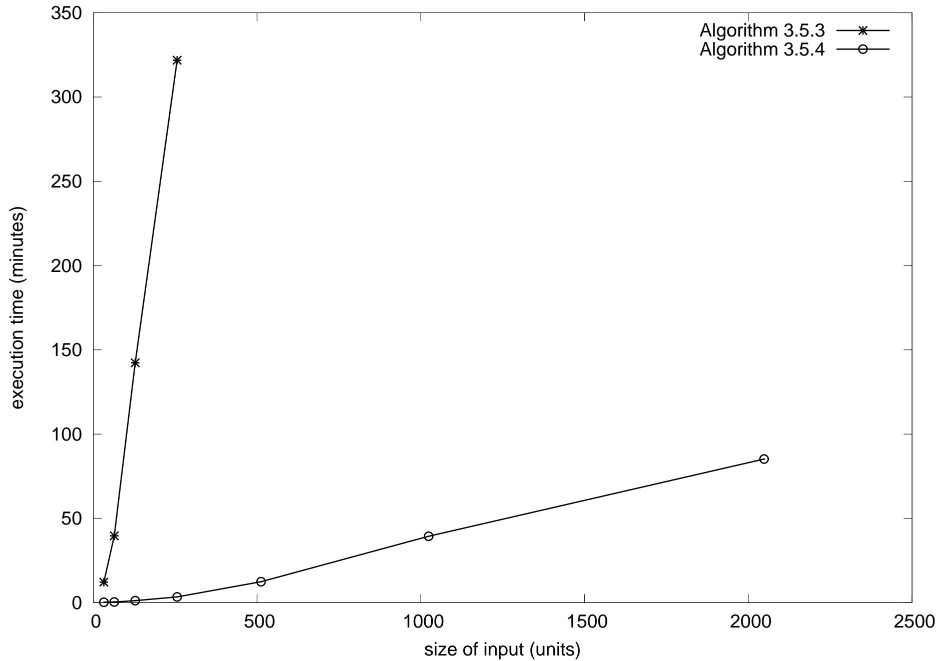

Figure 4.3. Performance of oblivious aggregation algorithms

From Table 4.2, we observe that unvectorized oblivious aggregation is extremely slow, while the vectorized version is significantly faster. We did not execute the unvectorized aggregation for input size larger than 256 because of its high processing time. We plot these values in Figure 4.3. We observe that the execution times for the vectorized algorithm scales much better with the input size. Therefore, it is clear that the vectorized algorithm should be chosen for oblivious aggregation.

For the second test, we executed the SecreC code of all three versions of the oblivious sorting Algorithms 3.5.5, 3.5.6 and 3.5.7 (see Appendix C.3, C.4 and C.5). The first algorithm is a bubble sorting network, the second is a vectorized odd-even transposition sorting network and the last one is a vectorized odd-even merge sorting network. Like the performance analysis of the oblivious aggregation, we calculated the processing time for different

Input size	Execution Time (minutes)		
	Bubble Sorting Network (Algorithm 3.5.5)	Vectorized Odd-even Transposition Sorting Network (Algorithm 3.5.6)	Vectorized Vectorized Odd-even Merge Sorting Network (Algorithm 3.5.7)
32	14.5	0.2	0.1
64	42.5	0.5	0.2
128	157.2	1.3	0.5
256	341.1	3.7	1.2
512	-	13.1	4.6
1024	-	41.5	18.9
2048	-	91.2	37.3

Table 4.3. Performance of oblivious sorting algorithms

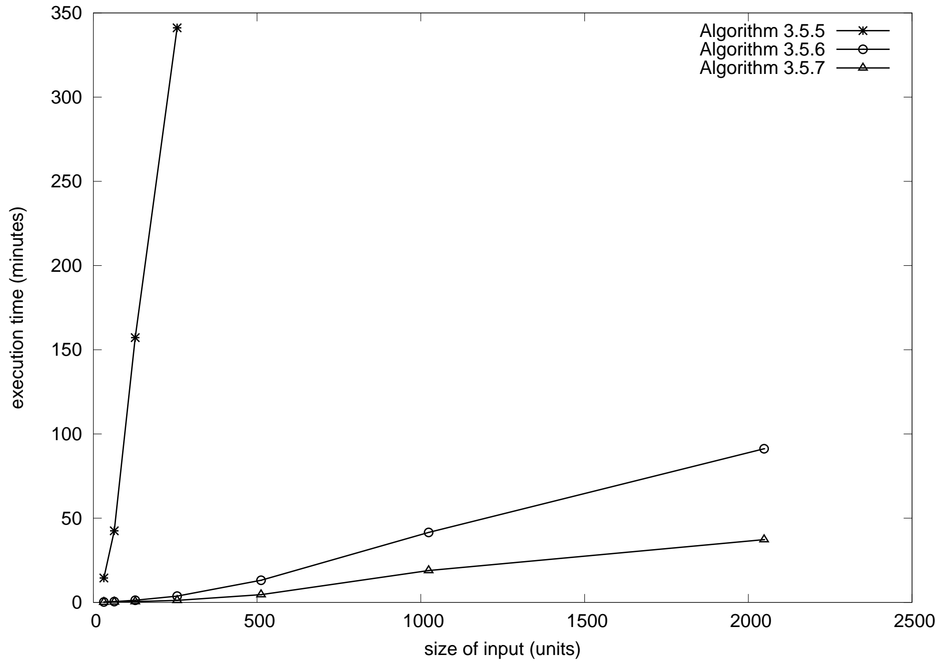

Figure 4.4. Performance of oblivious sorting algorithms

sizes of inputs. Table 4.3 presents these times in minutes.

We observe that, like the unvectorized aggregation algorithm, the unvectorized oblivious sorting network (Bubble sorting network) is significantly slower than the vectorized ones. Due to its high processing time, we did not execute the oblivious bubble sorting network for size of input data larger than 256 units. Between the two vectorized algorithms, odd-even merge sort (which has relatively lower complexity) gives better performance. To analyze this clearly, we plotted the values in Figure 4.4.

Again, we observe more favorable scaling properties for the vectorized algorithms. Of the two, we observe that odd-even merge sort performs noticably better.

	Execution Time (minutes)	
Input size	Configuration 1	Configuration 2
32	0.27	0.48
64	0.68	1.08
128	1.55	3.08
256	5.67	9.83
512	21.62	35.98
1024	70.37	132.00
2048	230.2	347.57

Table 4.4. Performance of the secure set union algorithm

For the third test, we executed the secure set union protocol, using both of the assembly configurations mentioned in Section 4.3.2. An implementation of the building blocks for both of these configurations is given in Appendix C. We define these assembly configurations as 'Configuration 1' and 'Configuration 2'. We measured the processing time for these configurations for different sizes of input and present them in Table 4.4. The times are given in minutes.

The configurations used the same aggregation and set reduction algorithms (Algorithms 3.5.4 and 3.5.8), but different versions of the oblivious sorting algorithms (Algorithms 3.5.6 and 3.5.7). Configuration 1 uses vectorized oblivious odd-even merge sort, whereas configuration 2 uses vectorized oblivious odd-even transposition sort. We have already shown in Section 4.3.4 that the former algorithm gives better performance, compared to the latter. So, it can be easily predicted that configuration 1 gives better performance. As seen in Table 4.4, the configuration 1 attains lower processing delay than configuration 2, as predicted. This is shown in Figure 4.5.

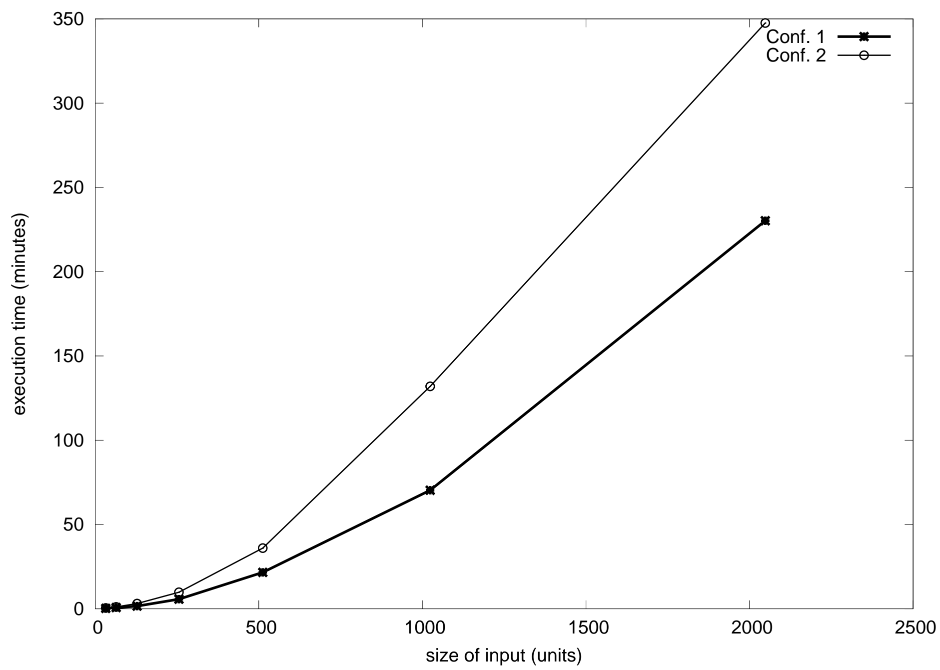

Figure 4.5. Performance of the secure set union algorithm

Chapter 5

Discussions

In this thesis, we have designed and implemented a novel method to compute privacy preserving set union and intersection using secure multiparty computation (MPC). We have designed several subprotocols to implement these set operations. We used these solutions in conjunction with an outlier detection system to design a privacy preserving collaborative anomaly detection system. We implemented the privacy preserving algorithms in the Sharemind framework [9] and the anomaly detection system using the LOCI scheme [31].

Our main achievement in this project, is a novel approach to implement privacy preserving sorting operations in secure multiparty computation based frameworks. We have designed a novel oblivious swapping technique that allows us to implement the sorting operations without comprimising the privacy of the inputs. We have also designed vectorized versions of oblivious swapping, that improve the performance of the sorting operations. We have also designed a novel vectorized oblivious algorithm for privacy preserving aggregation.

These two subprotocols (oblivious aggregation, oblivious sorting) played an integral role in designing the privacy preserving set union and intersection operations. In our experiments, we showed that it is possible to compose a fast privacy preserving set union protocol using the vectorized version of these algorithms. Finally, we have presented an alternate approach to formulate the LOCI scheme, using more robust statistics. We showed in our experiments that the modified LOCI scheme utilizing the median based statistics detected the network anomalies more successfully than the LOCI scheme utilizing the mean based statistics.

We have several limitations as well. The main objective of our project was to design a highly secure solution, that allows the parties to learn only the outcome and nothing else. But we failed to achieve that level of security, as our privacy preserving anomaly detection system leaks some information. Most of this information is leaked in executing the outlier detection system (LOCI) in a public environment. We can potentially eliminate this problem by implementing the LOCI scheme in a privacy preserving manner. As we stated earlier, implementing the LOCI scheme in MPC frameworks (e.g. Sharemind) can be very complicated and can significantly deteriorate the performance of the system.

Another limitation of our system is that our aggregation algorithms uses n^2 comparisons. We can reduce the number of comparisons to $n \log^2 n$ by modifying the odd-even merge sorting network to generate the comparison sequence. See Appendix A.2 for such an implementation. Due to lack of time, we could not implement the optimized privacy preserving aggregation algorithm. We leave this implementation for future work.

We tested our anomaly detection system in an isolated setting as we could not obtain multidomain attack traces. Finally, we tested our privacy preserving algorithms in an

localized setting by emulating the computing parties on a single computer. Since, there is no real network delay involved, the performance given by the algorithms at best an indication of their relative performance. We leave testing in a real networked environment for future work.

Although we tested our privacy preserving schemes are tested in fabricated data, but they give similar performance for real data set. The fabricated data that we used for testing are all 32 bit integers. In case of real data, the packet and the frequency counts are represented by 32 bit integers, and IP addresses can be easily converted to 32 bit integers. Since all of the values in real data can be represented by 32 bit integers, our scheme give the same performance.

We showed in our performance evaluation that the fastest version of our privacy preserving algorithm gives computation result for 1024 and 2048 units of input within an acceptable time. However, it is clear from the test results that our algorithm produces very slow output for higher number of inputs. Therefore, we can say that our current implementation of the privacy preserving algorithms are not practical in processing input set of large number of values e.g. 10000 items.

Bibliography

[1] Rakesh Agrawal, Alexandre Evfimievski, and Ramakrishnan Srikant. Information sharing across private databases. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 86–97, New York, NY, USA, 2003. ACM.

[2] M. Ajtai, J. Komlós, and E. Szemerédi. An 0(n log n) sorting network. In *STOC '83: Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 1–9, New York, NY, USA, 1983. ACM.

[3] V. Barnett and T Lewis. *Outliers in Statistical Data.* John Wiley & Sons, 3rd edition edition, 1994.

[4] K. E. Batcher. Sorting networks and their applications. In *AFIPS '68 (Spring): Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314, New York, NY, USA, 1968. ACM.

[5] D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols. In *STOC '90: Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 503–513, New York, NY, USA, 1990. ACM.

[6] Jean-Paul Berrut and Lloyd N. Trefethen. Barycentric lagrange interpolation. *SIAM Review*, 46(3):501–517, 2004.

[7] G. R. Blakley. Safeguarding cryptographic keys. *Managing Requirements Knowledge, International Workshop on*, 0:313, 1979.

[8] Dan Bogdanov, Roman Jagomägis, and Sven Laur. Privacy-preserving histogram computation and frequent itemset mining with sharemind. Technical Report Cybernetica research report T-4-8, Cybernetica AS, 2009.

[9] Dan Bogdanov, Sven Laur, and Jan Willemson. *Sharemind: A Framework for Fast Privacy-Preserving Computations*, volume 5283/2008 of *Computer Security - ESORICS 2008.* Springer Berlin / Heidelberg, October 2008.

[10] Markus M. Breunig, Hans-Peter Kriegel, Raymond T. Ng, and Jörg Sander. Lof: identifying density-based local outliers. *SIGMOD Rec.*, 29(2):93–104, 2000.

[11] Josh Cohen Benaloh. Secret sharing homomorphisms: keeping shares of a secret secret. In *Proceedings on Advances in cryptology—CRYPTO '86*, pages 251–260, London, UK, 1987. Springer-Verlag.

[12] Ronald Cramer and Ivan Damgård. Secure distributed linear algebra in a constant number of rounds. In *CRYPTO '01: Proceedings of the 21st Annual International*

Cryptology Conference on Advances in Cryptology, pages 119–136, London, UK, 2001. Springer-Verlag.

[13] Bogdanov Dan and Richard Sassoon. Privacy-preserving collaborative filtering with sharemind. Technical Report Cybernetica research report T-4-2, Cybernetica AS, 2008.

[14] Laurie Davies and Ursula Gather. Robust statistics. *Handbook of Computational Statistics: Concepts and Methods*, pages 670–672, 2004.

[15] Wenliang Du and Mikhail J. Atallah. Protocols for secure remote database access with approximate matching. Technical report, CERIAS, Purdue University, 2000.

[16] Fatih Emekci, Divyakant Agrawal, Amr El Abbadi, and Aziz Gulbeden. Privacy preserving query processing using third parties. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering*, page 27, Washington, DC, USA, 2006. IEEE Computer Society.

[17] Arian R. Van Erkel and Peter M. Th. Pattynama. Receiver operating characteristic (roc) analysis: Basic principles and applications in radiology. *European Journal of Radiology*, 27(2):88 – 94, 1998.

[18] Rosario Gennaro, Yuval Ishai, Eyal Kushilevitz, and Tal Rabin. The round complexity of verifiable secret sharing and secure multicast. In *STOC '01: Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 580–589, New York, NY, USA, 2001. ACM.

[19] Rosario Gennaro, Michael O. Rabin, and Tal Rabin. Simplified vss and fast-track multiparty computations with applications to threshold cryptography. In *PODC '98: Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, pages 101–111, New York, NY, USA, 1998. ACM.

[20] Oded Goldreich. Foundations of cryptography: a primer. *Found. Trends Theor. Comput. Sci.*, 1(1):1–116, 2005.

[21] Roman Jagomägis. Secrec: a privacy-aware programming language with applications in data mining. Master's thesis, University of Tartu, 2010.

[22] Lea Kissner and Dawn Song. Privacy-preserving set operations. In *in Advances in Cryptology - CRYPTO 2005, LNCS*, pages 241–257. Springer, 2005.

[23] Edwin M. Knorr and Raymond T. Ng. Algorithms for mining distance-based outliers in large datasets. In *Algorithms for Mining Distance-Based Outliers in Large Datasets*, pages 392–403, 1998.

[24] Donald E. Knuth. *The Art of Computer Programming: Sorting and Searching (Volume 3)*. Addison-Wesley, 1998.

[25] George Kurtz. Google attack is tip of iceberg, January 2010. Available at: http://siblog.mcafee.com/cto/google-attack-is-tip-of-iceberg/, Last accessed in May 2010.

[26] Adam J. Lee, Parisa Tabriz, and Nikita Borisov. A privacy-preserving interdomain audit framework. In *WPES '06: Proceedings of the 5th ACM workshop on Privacy in electronic society*, pages 99–108, New York, NY, USA, 2006. ACM.

[27] Yehuda Lindell. Parallel coin-tossing and constant-round secure two-party computation. In *CRYPTO '01: Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, pages 171–189, London, UK, 2001. Springer-Verlag.

[28] Richard Lippmann, Joshua W. Haines, David J. Fried, Jonathan Korba, and Kumar Das. The 1999 darpa off-line intrusion detection evaluation. *Comput. Netw.*, 34(4):579–595, 2000.

[29] Richard P. Lippmann, David J. Fried, Isaac Graf, Joshua W. Haines, Kristopher R. Kendall, David McClung, Dan Weber, Seth E. Webster, Dan Wyschogrod, Robert K. Cunningham, and Marc A. Zissman. Evaluating intrusion detection systems: The 1998 darpa off-line intrusion detection evaluation. *DARPA Information Survivability Conference and Exposition*, 2:1012, 2000.

[30] Moni Naor and Benny Pinkas. Oblivious transfer and polynomial evaluation. In *STOC '99: Proceedings of the thirty-first annual ACM symposium on Theory of computing*, pages 245–254, New York, NY, USA, 1999. ACM.

[31] S. Papadimitriou, H. Kitagawa, P.B. Gibbons, and C. Faloutsos. Loci: fast outlier detection using the local correlation integral. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 315 – 326, 5-8 2003.

[32] P. Rogaway. *The Round Complexity of Secure Protocols*. PhD thesis, MIT, June 1991. Available from www.cs.ucdavis.edu/ rogaway/papers/.

[33] Stuart E. Schechter, Jaeyeon Jung, and Arthur W. Berger. Fast detection of scanning worm infections. In *RAID*, pages 59–81, 2004.

[34] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[35] Riivo Talviste and Dan Bogdanov. An improved method for privacy-preserving web-based data collection. Technical Report Cybernetica research report T-4-5, Cybernetica As, 2009.

[36] Jian Tang, Zhixiang Chen, Ada Wai-chee Fu, and David Cheung. A robust outlier detection scheme for large data sets. In *In 6th Pacific-Asia Conf. on Knowledge Discovery and Data Mining*, pages 6–8, 2001.

[37] David Wagner. Resilient aggregation in sensor networks. In *SASN '04: Proceedings of the 2nd ACM workshop on Security of ad hoc and sensor networks*, pages 78–87, New York, NY, USA, 2004. ACM.

[38] Andrew C. Yao. Protocols for secure computations. In *SFCS '82: Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, pages 160–164, Washington, DC, USA, 1982. IEEE Computer Society.

Appendix A

Comparison Sequence Generators

A.1 Python script for odd-even merge sort sequence generator

```
import sys
import math
def mergesort(idxs, depth):
    logsize = int(math.log(len(idxs)) / math.log(2))
    assert 2**logsize == len(idxs)
    if depth > logsize and len(idxs) > 2:
       return mergesort(idxs[:len(idxs)/2], depth - logsize) +
       mergesort(idxs[len(idxs)/2:], depth - logsize)
    else:
       return oddevenmerge(idxs, depth)
def oddevenmerge(idxs, depth):
    if len(idxs) <= 1:
       return []
    if depth == 1:
       if len(idxs) == 2:
           return [(idxs[0], idxs[1])]
       return [ (idxs[i], idxs[i+1]) for i in xrange(1, len(idxs) - 2, 2)]
    else:
        return oddevenmerge(idxs[::2], depth - 1) +
        oddevenmerge(idxs[1::2], depth - 1)
def main():
    tosort = range(2**int(sys.argv[1]))
    maxdepth = 1
    while mergesort(tosort, maxdepth):
       maxdepth += 1
    maxdepth -= 1
    for i in xrange(maxdepth, 0, -1):
       print mergesort(tosort, i)
if __name__ == '__main__':
   main()
```

A.2 Python script for modified odd-even merge sorting network sequence generator

```
import sys
import math
def oddevenmerge(idxs, depth):
    if len(idxs) <= 1:
        return []
    if depth <= 2:
        return [ (idxs[i], idxs[i+1]) for i in xrange(depth - 1, len(idxs) - 1, 2)]
    else:
        return oddevenmerge(idxs[::2], depth - 2) +
        oddevenmerge(idxs[1::2], depth - 2)
def main():
    tosort = range(2**int(sys.argv[1]))
    maxdepth = 1
    while oddevenmerge(tosort, maxdepth):
        maxdepth += 1
    maxdepth -= 1
    for i in xrange(maxdepth, 0, -1):
        print oddevenmerge(tosort, i)
if __name__ == '__main__':
    main()
```

Appendix B

Python Script for LOCI scheme

```
from numpy import *
def outmean(n,np,l):
    lh = abs(np-mean(n))
    rh = l*std(n)
    s = 0
    if  lh>rh: s = 1
    return s
def mad(n):
    m = zeros(len(n))
    m[0:len(n)] = median(n)
    m = abs(m - n)
    return median(m)
def outmedian(n,np,l):
    lh = abs(np-median(n))
    rh = l*mad(n)
    s = 0
    if  lh>rh: s = 1
    return s
def distance(v):
    d = zeros((len(v),len(v)),dtype=int)
    for i in range(len(v)):
        d[i][0:len(v)] = abs(v[i]-v)
    return d
def flag_out(d,ind,k,a):
    n = []
    temp = sort(d[ind])
    kd = temp[k-1]
    kad = kd*a
    np = float(len(array([where(d[ind]<=kad)]).ravel()))
    knn = array(where(d[ind]<=kd)).ravel()
    for i in range(len(knn)):
        n.append(len(array(where(d[knn[i]]<=kad)).ravel()))
    n = array(n)
    lmean1=sqrt(2*log10(len(temp)))
    lmean2 = 3
```

```
    lmedian = 5.2
    s1=outmean(n,np,lmean1)
    s2=outmedian(n,np,lmedian)
    return s1,s2
f = open("data.txt","r")
ip=[]
origval=[]
for line in f:
    line = line.strip()
    line = line.split(",")
    ip.append(line[0])
    origval.append(float(line[1]))
val = sort(array(origval))
N = len(val)
O = 7
a=0.25
d=distance(val)
s_mean = zeros(N)
s_median = zeros(N)
k = int(ceil(N*0.85))
outval = []
for k in range(27):
    for i in range(N):
        s_mean[i],s_median[i] = flag_out(d,i,k,a)
        if(s_median[i]==1):
            outval.append(val[i])
    outval = unique(outval)
    outliers=[]
    for i in range(len(outval)):
        ind=array(where(origval==outval[i])).ravel()
        for i in range(len(ind)):
            outliers.append(ip[ind[i]])
    for i in outliers:
        mean_fpr = sum(s_mean[0:N-(O+1)])/(N-O)*100
        mean_tpr = sum(s_mean[N-O:N])/O*100
        median_fpr = sum(s_median[0:N-(O+1)])/(N-O)*100
        median_tpr = sum(s_median[N-O:N])/O*100
    print "K: ",k," MEAN TPR: ",mean_tpr,", MEAN FPR:",mean_fpr
    print "K: ",k," MED TPR: ",median_tpr,", MED FPR: ",median_fpr
```

Appendix C

Privacy Preserving Algorithms

C.1 SecreC code for 'oblivious aggregation algorithm'

```
private int[][] aggregate (private int[][] d){
    public int len; len = vecLength(d); len = len/3;
    private int IP1; private int IP2;
    private int cnt1; private int cnt2;
    private int f1; private int f2;
    private bool comp; private int c; private int invc;
    public int i;
    public int j;
    public int pub;
    for(j=0;j<len;j=j+1){
        for(i=0;i<len-1;i=i+1){
        IP1 = d[0][i];
        IP2 = d[0][i+1];
        cnt1 = d[1][i];
        cnt2 = d[1][i+1];
        f1 = d[2][i];
        f2 = d[2][i+1];
        comp = (IP1==IP2);
        c = boolToInt(comp);
        invc = 1 - c;
        d[1][i] = (cnt1+cnt2)*c + cnt1*invc;
        d[1][i+1] = (cnt2-cnt2)*c + cnt2*invc;
    d[2][i] = (f1+f2)*c + f1*invc;
        d[2][i+1] =(f2-f2)*c + f2*(1-c);
        }
    }
    return d;
}
```

C.2 SecreC code for 'vectorized oblivious aggregation algorithm'

```
// Shaping the aggregation by eliminating duplicate entries created in aggregate function
```

```
private int[] shaping (private int[] d, private int[] c, public int ind){
    public int len; len = vecLength(d);
    private int[len] tmp;
    tmp = d*c;
    d[ind] = vecSum(tmp);
    tmp[ind] = 0;
    d = d - tmp;
    return d;
}


// Aggregate the counts and frequencies whose IPs are equal
private int[][] aggregate (private int[][] d){
    public int len; len = vecLength(d); len = len/3;
    private int[len] d0; d0 = d[0][*]; private int[len] d1; d1 = d[1][*];private int[len] d2; d
    private int[len] seed; private bool[len] comp; private int[len] c; public int i;
    for(i=0;i<len;i=i+1) {
        seed=d0[i];
        comp = (d0==seed);
        c = boolToInt(comp);
        d1 = shaping(d1, c, i);
d2 = shaping(d2, c, i);
    }
    d[1][*] = d1;
    d[2][*] = d2;
    return d;
}
```

C.3 SecreC code for 'oblivious bubble sorting network'

```
private int[][] sorting (private int[][] d){
    public int len; len = vecLength(d); len = len/3;
    private int IP1; private int IP2;
    private int cnt1; private int cnt2;
    private int f1; private int f2;
    private bool comp; private int c; private int invc;
    public int i; public int j;
for(i=len-1;i>=1;i=i-1) {
        for(j=0;j<i;j=j+1) {
IP1 = d[0][j]; IP2 = d[0][j+1];
cnt1 = d[1][j]; cnt2 = d[1][j+1];
f1 = d[2][j]; f2 = d[2][j+1];
        comp = (f1>=f2);
        c = boolToInt(comp);
        invc = 1 - c;
d[0][j] = IP2*c + IP1*invc;
        d[0][j+1] = IP1*c + IP2*invc;
        d[1][j] = cnt2*c + cnt1*invc;
```

```
        d[1][j+1] = cnt1*c + cnt2*invc;
    d[2][j] = f2*c + f1*invc;
        d[2][j+1] = f1*c + f2*invc;
     }
  }
    return d;
}private int[][] sorting (private int[][] d){
    public int len; len = vecLength(d); len = len/3;
    private int IP1; private int IP2;
    private int cnt1; private int cnt2;
    private int f1; private int f2;
    private bool comp; private int c; private int invc;
    public int i; public int j;
for(i=len-1;i>=1;i=i-1) {
        for(j=0;j<i;j=j+1) {
IP1 = d[0][j]; IP2 = d[0][j+1];
cnt1 = d[1][j]; cnt2 = d[1][j+1];
f1 = d[2][j]; f2 = d[2][j+1];
        comp = (f1>=f2);
        c = boolToInt(comp);
        invc = 1 - c;
d[0][j] = IP2*c + IP1*invc;
        d[0][j+1] = IP1*c + IP2*invc;
        d[1][j] = cnt2*c + cnt1*invc;
        d[1][j+1] = cnt1*c + cnt2*invc;
    d[2][j] = f2*c + f1*invc;
        d[2][j+1] = f1*c + f2*invc;
     }
  }
    return d;
}
```

C.4 SecreC code for 'vectorized oblivious odd-even transposition sorting network'

```
//One sorting round according to the frequency
private int[][] sort_round(private int[][] d){
    public int len; len = vecLength(d); len = len/3;
    private int[len] IP; IP = d[0][*];
    private int[len] cnt; cnt = d[1][*];
    private int[len] freq; freq = d[2][*];
    private int[len] otherIP; private int[len] othercnt; private int[len] otherfreq;
    private bool[len] comp; private int[len] c;
    private int[len] temp; temp = 1; public int i;
    for(i=0;i<len;i=i+2){
            otherIP[i] = IP[i+1]; otherIP[i+1] = IP[i];
        othercnt[i] = cnt[i+1]; othercnt[i+1] = cnt[i];
            otherfreq[i] = freq[i+1]; otherfreq[i+1] = freq[i];
```

```
        }
        comp = (freq>=otherfreq);
        for(i=0;i<len;i=i+2){
            comp[i+1] = comp[i];
        }
        c = boolToInt(comp);
        IP = (temp-c)*IP+c*otherIP;
    cnt = (temp-c)*cnt+c*othercnt;
        freq = (temp-c)*freq+c*otherfreq;
    d[0][*] = IP; d[1][*] = cnt; d[2][*] = freq;
    return d;
 }

 //sorting a vector
private int[][] sorting (private int[][] d){
    public int len; len = vecLength(d); len = len/3;
    public int i; public int j; public int k; k = len-2;
    private int[3][len-2] d_next; d_next = 0;
    for(i=0;i<len/3;i=i+1){
        d = sort_round(d);
        for(j=0;j<k;j=j+1){
           d_next[0][j] = d[0][j+1]; d_next[1][j] = d[1][j+1]; d_next[2][j] = d[2][j+1];
        }
        d_next = sort_round(d_next);
        for(j=0;j<k;j=j+1){
            d[0][j+1] = d_next[0][j]; d[1][j+1] = d_next[1][j];d[2][j+1] = d_next[2][j];
        }
    }
    return d;
}
```

C.5 SecreC code for 'vectorized oblivious odd-even merge sorting network'

Comments: This is a python based code generator that generates the SecreC code for oblivious sorting, based on the comparison sequence generated by the odd-even merge sorting network sequence geneator (Appendix A.1).

```
from numpy import *
import sys

f = open("newfile",'r')
rounds=0
num = pow(2,int(sys.argv[1]))
for s in f: rounds = rounds+1
f.close()
sn = zeros((rounds, num),'i')
```

```
for i in range(rounds):
    for j in range(num):
        sn[i][j] = 32768

f = open("newfile",'r')
i=0
for s in f:
    s = s.strip()
    s = s.strip('[')
    s = s.strip(']')
    s = s.replace('(','')
    s = s.replace(')','')
    s = s.strip(' ')
    s = s.split(',')
    n = len(s)
    for j in range(0,n,2):
       sn[i][int(s[j])]=int(s[j+1])
       sn[i][int(s[j+1])]=int(s[j])
    i = i+1

# start sharemind conversion
print "void main(){"
print "public int num;"
print "num = "+str(rounds)+";"
print "dbLoad(\"IPTables\");"
print "public int rows;"
print "rows = dbRowCount(\"IPTables\");"
print "private int[rows] IP;"
print "IP= dbGetColumn (\"IP\", \"IPTables\");"
print "private int[rows] cnt;"
print "cnt= dbGetColumn (\"CNT\", \"IPTables\");"
print "private int[rows] freq;"
print "freq= dbGetColumn (\"FREQ\", \"IPTables\");"
print "private bool[rows] comp;"
print "private int[rows] c;"
print "private int[rows] temp;"
print "temp = 1;"
print "private int[rows] otherIP;"
print "private int[rows] otherCnt;"
print "private int[rows] otherFreq;"
print "private int[rows] tmpCnt;"
print "private int[rows] tmpFreq;"
print "private int[rows] seed;"
print "private int[rows] th;"
print "private int z;"
print "th = 0;"
print "public int i;"
print "public int k;"
print "public int[num][rows] sn;"
print "public int[rows] pub;"
```

```
print "public int tmp;"
for i in range(i):
    for j in range(num):
        print 'sn['+str(i)+']['+str(j)+']='+str(sn[i][j])+';'

print "for(k=0;k<num;k=k+1){"
print "for(i=0;i<rows;i=i+1){"
print "tmp = sn[k][i];"
print "if(tmp!=32768){"
print "otherIP[i] = IP[tmp];"
print "otherFreq[i] = freq[tmp];"
print "}"
print "else{"
print "otherIP[i] = IP[i];"
print "otherCnt[i] = cnt[i];"
print "otherFreq[i] = freq[i];"
print "}"
print "}"
print "comp = (freq>=otherFreq);"
print "for(i=0;i<rows;i=i+1){"
print "tmp = sn[k][i];"
print "if(tmp!=32768){"
print "comp[tmp] = comp[i];"
print "}"
print "}"
print "c = boolToInt(comp);"
print "IP = c*IP+(temp-c)*otherIP;"
print "freq = c*freq+(temp-c)*otherCnt;"
print "freq = c*freq+(temp-c)*otherFreq;"
print "}"
print "pub = declassify(cnt);"
print "publish(\"Result\",pub);"
print "}"
```

C.6 Secure Set Reduction Operation

```
//reduce the vector if frequency is zero (important for set union and intersection)
private int[][] reduce(private int[][] d){
    public int len; len = vecLength(d); len = len/3;
    private int[len] IP; IP = d[0][*];
    private int[len] cnt; cnt = d[1][*];
    private int[len] f; f = d[2][*];
    private int[len] zeros; zeros = 0;
    private bool[len] compzero;
    private int s; public int z; public int i; public int ind; public int[len] pub;
    compzero = (f==zeros); s = vecSum(compzero); z = declassify(s);
    for (i=0;i<z;i=i+1) {
        vecRemove(IP,0);
```

```
        vecRemove(cnt,0);
        vecRemove(f,0);
    }
    public int rows; rows = len - z;
    private int[3][rows] e;
    e[0][*] = IP;
    e[1][*] = cnt;
    e[2][*] = f;
    return e;
}
```