ABSTRACT

XU, DINGBANG. Correlation Analysis of Intrusion Alerts. (Under the direction of Dr. Peng Ning).

Security systems such as intrusion detection systems (IDSs) are widely deployed into networks to better protect digital assets. However, there are several problems related to current IDSs. (1) IDSs may flag a large number of alerts everyday, thus overwhelming the security officers. (2) Among the alerts flagged by IDSs, false alerts (i.e., false positives) are mixed with true ones, and usually it is difficult to differentiate between them. (3) Existing IDSs may not detect all attacks launched by adversaries. These problems make it very challenging for human users or intrusion response systems to understand the alerts and take appropriate actions. Thus, it is necessary to perform alert correlation. My dissertation focuses on correlation analysis of intrusion alerts. In particular, I have worked on the following issues.

The first issue is the efficiency of alert correlation. This work is extended from our previous correlation method [83]. The initial implementation of [83] is a Database Management System based toolkit. To improve its performance, we propose to adapt main memory index structures and database query optimization techniques to facilitate timely correlation of intensive alerts. We present three techniques named *hyper-alert container, two-level index,* and *sort correlation,* and study the performance of these techniques.

The second issue is to learn attack strategies. We notice that understanding the strategies of attacks is crucial for security applications such as network forensics and intrusion response. We propose techniques to automatically learn attack strategies from intrusion alerts, where attack strategies are modeled as directed graphs with nodes representing attacks and edges representing constraints between corresponding nodes. We further present techniques to measure the similarity between attack strategies using the techniques in error tolerant graph/subgraph isomorphism.

The third issue is how to hypothesize and reason about attacks missed by IDSs. We notice that current alert correlation methods depend heavily on the underlying IDSs for providing alerts, and cannot deal with attacks missed by IDSs. We present techniques to hypothesize attacks possibly missed by the IDSs, to infer attribute values for hypothesized attacks, to validate and prune hypothesized attacks through examining raw audit data, and to consolidate hypothesized attacks to get concise attack scenarios.

The fourth issue is to correlate alerts from different security systems. We notice that complementary security systems such as IDSs and firewalls are widely deployed in networks. We

propose a correlation approach based on triggering events and common resources. Our approach first performs alert clustering such that the alerts in each cluster share "similar" triggering events. We further propose techniques to build attack scenarios through identifying "common" resources between different attacks.

The fifth issue is privacy-privacy alert correlation. We notice that there are privacy concerns when intrusion alerts are shared and correlated among different organizations. We propose one generalization based scheme and three perturbation based schemes to anonymize alerts to protect data privacy. To evaluate privacy protection, we use *entropy* to guide alert anonymization. In addition, to learn the utility of anonymized alerts, we further perform correlation analysis for anonymized data sets. We focus on estimating similarity values between anonymized attributes and building attack scenarios from anonymized data sets.

Finally, the conclusion of my dissertation is provided and future work is pointed out.

Correlation Analysis of Intrusion Alerts

by

Dingbang Xu

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh

2006

Approved By:

Dr. Douglas S. Reeves	Dr. Ting Yu
Dr. Peng Ning Chair of Advisory Committee	Dr. S. Purushothaman Iyer

To my parents Yuxin Xu and Juying Xue.

Biography

Dingbang Xu received a Bachelor's degree from Huazhong University of Science and Technology, and a Master's degree from Tsinghua University. He is a Ph.D. student in Computer Science Department at North Carolina State University from 2001 to 2006. His research interests are information and network security. In particular, he is interested in intrusion detection techniques, security data management, and privacy-preserving techniques.

Acknowledgements

This dissertation would not be possible without the support and help from many professors, friends and my parents over many years.

I would like to thank my Ph.D. advisor Dr. Peng Ning. His guidance, encouragement, and support is so valuable to my Ph.D. research. I feel very fortunate to work with him. Sincere thanks are also extended to my committee members Dr. S. Purushothaman Iyer, Dr. Douglas S. Reeves and Dr. Ting Yu for their valuable comments, suggestions, and help. I also would like to thank Dr. Annie I. Antón for her valuable feedback and comments on my research.

The work in this dissertation is supported by the National Science Foundation (NSF) under grants ITR-0219315 and CCR-0207297, and by the U.S. Army Research Office (ARO) under grant DAAD19-02-1-0219.

I would like to thank Director of Graduate Program (DGP) Dr. David J. Thuente for his help during my Ph.D. study. I am also grateful to Ms. Margery Page for her help.

I would like to thank many friends in Cyber Defense Laboratory Srinath Anantharaju, Yun Cui, Yiquan Hu, Qinglin Jiang, Hua Li, An Liu, Donggang Liu, Jaideep Mahalati, Pai Peng, Alfredo Serrano, Pratik Shah, Kun Sun, Pan Wang, Yan Zhai, Qing Zhang, Qinghua Zhang, Yi Zhang, and Yuzheng Zhou for their help.

Finally, I am deeply grateful to my parents Yuxin Xu and Juying Xue for their continued support and encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The focus of my dissertation is correlation analysis of intrusion alerts. To defend against various attacks, various security systems such as intrusion detection systems (IDSs) are widely deployed into hosts and networks. These systems may flag alerts when suspicious events are observed. Correlating the alerts from these security systems can help us understand the security threats and take appropriate response. This chapter gives an introduction and also motivates my research.

1.1 Intrusion Detection and Intrusion Alert Correlation

With the development of the Internet, more and more organizations manage their data in networked information systems. Due to the open nature of the Internet, network intrusions have become an increasingly serious problem in recent years. For example, Code Red worm infected more than 250,000 machines in about 9 hours on July 19, 2001, and *Computer Economics* estimated the financial loss of Code Red was $2.6 billion [20]. Intrusion detection, which is aimed at detecting activities violating the security policies of the networked information systems, has been considered a necessary component to protect these systems along with other prevention-based security mechanisms such as access control.

Generally speaking, intrusion detection techniques can be classified into two categories: misuse detection and anomaly detection [10, 85]. Misuse detection builds signatures (patterns) for known attacks or vulnerabilities, and raises alerts when it monitors the activities that match

the signatures. Anomaly detection builds models (e.g., statistical profiles) for normal activities, and raises alerts when the monitored activities (significantly) deviate from the normal operations. Intrusion detection systems (IDSs) are widely deployed into hosts and networks to protect digital assets.

Despite more than 20 years' efforts on intrusion detection, current intrusion detection systems still have several well-known problems. First, existing IDSs cannot detect all intrusions. While a misuse detection system cannot detect an unknown attack (or an unknown variation of a known attack), an anomaly detection system may fail to recognize stealthy malicious activities, too. Second, current IDSs cannot ensure that all alerts reflect actual attacks; *true positives* (attacks detected as intrusive) are usually mixed with *false positives* (benign activities detected as intrusive). Third, an IDS usually produces a large number of alerts [11, 59, 60, 61]. As indicated in [59], five IDS sensors reported 40MB of alert data within ten days, and a large fraction of these alerts are false positives. In our experience with IDSs in campus networks, we observed more than 320,000 alerts on a small subnet in less than five days. The high volumes and low quality (i.e., missed attacks and false positives) of the intrusion alerts make it very challenging for human users or intrusion response systems to understand the alerts and take appropriate actions. Thus, it is necessary to develop techniques to deal with the large volumes and low quality of intrusion alerts.

Besides the aforementioned problems, current IDSs are not sufficiently prepared for several trends in attacks. According to a 2002 CERT report [20], there are increasingly more automated attack tools, which typically consist of several (evolving) phases such as scanning for potential victims, compromising vulnerable systems, propagating the attacks, and coordinated management of attack tools. Moreover, attack tools are increasingly more sophisticated. In particular, "today's automated attack tools can vary their patterns and behaviors based on random selection, predefined decision paths, or through direct intruder management" [20]. These attack trends require more capable systems than the current IDSs to handle large volumes of alerts that potentially belong to different complex attack scenarios.

Intrusion alert correlation, focusing on discovering the relationships between the alerts raised by security systems, is necessary and crucial to understand the security threats from inside and outside sources and take appropriate actions. In recent years, several alert correlation techniques have been proposed to facilitate the analysis of intrusion alerts. These techniques can be roughly divided into four categories: (1) the methods based on similarity between alerts (e.g, [109, 98, 60, 61, 33, 91, 28]), which essentially perform alert clustering through computing similarity between alert attributes, (2) methods based on predefined attack scenarios (e.g., [36, 34, 78]), which build

attack scenarios through matching alerts to predefined scenario templates, (3) techniques based on prerequisites (pre-conditions) and consequences (post-conditions) of attacks (e.g., [102, 29, 83]), which build attack scenarios through (partially) matching consequences of earlier attacks to the prerequisites of earlier attacks, and (4) approaches using multiple information sources (e.g., [90, 79, 115, 116], which correlate alerts from multiple security systems such as firewalls and IDSs.

We are interested in the prerequisites and consequences based correlation method such as [83] (We give an overview of [83] in Chapter 2). These methods model each attack through specifying its prerequisite and consequence. Intuitively, the prerequisite of an attack is the necessary condition to launch an attack successfully (e.g., a vulnerable *FTP* service running on a victim host is the prerequisite to launch an *FTP* buffer overflow attack), and the consequence of an attack is the possible outcome if the attack succeeds (e.g., the consequence of an *FTP* buffer overflow attack may be gaining the root privilege on the victim host). Through matching the consequences of earlier attacks with the prerequisites of later ones (e.g., an port scanning attack may discover vulnerable *FTP* ports, which is the prerequisite for later *FTP* buffer overflow attacks), causal relations between attacks are identified, and we can build attack scenarios by connecting different attacks through a sequence of causal relations.

In my dissertation, I have addressed some important issues in intrusion alert correlation (especially for prerequisites and consequences based methods). We motivate these problems in the following subsections.

1.2 Efficiency of Intrusion Alert Correlation

Our previous correlation method [83] has been implemented as an offline intrusion alert correlator (the details of [83] are given in Chapter 2). Our initial experiments with 2000 DARPA intrusion detection scenario specific data sets [77] indicate that our approach is promising in constructing attack scenarios and differentiating true and false alerts [83]. However, our solution still faces some challenges. In particular, we implemented the previous intrusion alert correlator as a DBMS-based application [83]. Involving a DBMS in the alert correlation process provided enormous convenience and support in our initial implementation; however, relying entirely on the DBMS also introduced performance penalty. For example, to correlate about 65,000 alerts generated from the DEF CON 8 CTF dataset [37], it took the DBMS-based intrusion alert correlator around 45 minutes with the JDBC-ODBC driver included in Java 2 SDK, Standard Edition

(http://java.sun.com/j2se/), and more than 4 minutes with the Microsoft SQL Server 2000 Driver for JDBC (http://www.microsoft.com/sql/). Such performance is clearly not sufficient to make alert correlation a practical tool, especially for interactive analysis of intensive alerts. Our timing analysis indicates that the performance bottleneck lies in the interaction between the intrusion alert correlator and the DBMS. In particular, the processing of each single alert entails interaction with the DBMS, which introduces significant performance overhead.

Our solution to this problem is to adapt main memory index structures and query optimization techniques to perform alert correlation efficiently, which is discussed in Chapter 3.

1.3 Learning Attack Strategies

It is often desirable, and sometimes necessary, to understand attack strategies in security applications such as computer and network forensics and intrusion responses. For example, attack strategies may be used to profile hackers or hacking tools in computer and network forensics. As another example, it is easier to predict attacker's next move, and reduce the damage caused by intrusions, if the attack strategy is known during intrusion response. However, in practice, it usually requires that human users manually analyze the data collected during intrusions to understand the corresponding attack strategies. This process is not only time-consuming, but also error-prone. An alternative to manual analysis is to enumerate and reason about attack strategies through static vulnerability analysis (e.g., [97, 6]). However, these techniques usually require predefined security properties so that they can identify possible attack sequences that lead to the violation of these properties. Although it is easy to specify certain security properties such as the compromise of root privileges, it is non-trivial to enumerate all possible ones. Moreover, analyzing intrusion alerts allows inspecting actual execution of attack strategies with different levels of details. Thus it is desirable to have complementary techniques that can profile attack strategies from intrusion alerts.

In Chapter 4, we present techniques to automatically learn attack strategies from intrusion alerts reported by IDSs.

1.4 Hypothesizing and Reasoning about Attacks Missed by Intrusion Detection Systems

As we mentioned earlier, several alert correlation techniques have been proposed in recent years to facilitate the analysis of intrusion alerts. These techniques include similarity based approaches, predefined attack scenarios based methods, prerequisites and consequences based approaches, and multiple information sources based techniques. We observe that a common requirement of these approaches is that they all heavily depend on the underlying IDSs for providing alerts. As a result, the performance of alert correlation is strictly limited by the performance of IDSs. In particular, if the IDSs miss critical attacks, the correlated alerts cannot reflect the actual attack scenarios due to the lack of the corresponding alerts, and thus may provide misleading information.

In Chapter 5, we develop a series of techniques to hypothesize and reason about attacks possibly missed by IDSs, aiming at constructing high-level attack scenarios even if the underlying IDSs miss critical attacks.

1.5 Alert Correlation through Triggering Events and Common Resources

Current approaches on intrusion alert correlation are effective at addressing some challenges, however, it is also clear that none of them dominates the others. Similarity based approaches group alerts based on the similarity between alert attributes; however, they are not good at discovering steps in a sequence of attacks. Predefined attack scenario based approaches work well for known scenarios; however, they cannot discover novel attack scenarios. Prerequisites and consequences based approaches can discover novel attack scenarios; however, the procedure of specifying prerequisites and consequences are time-consuming and error-prone. Multiple information sources based approaches correlate alerts from multiple information sources such as firewalls and IDSs; however, they are not good at discovering novel attack scenarios.

To address some limitations of the current correlation techniques, we propose an alert correlation approach based on triggering events and common resources. In particular, we propose a novel similarity measure based on triggering events, which helps us group alerts into clusters such that the alerts in the same cluster share “similar” triggering events. We enhance the prerequisites and

consequences based approaches through using input and output resources to facilitate the specification of prerequisites and consequences. Intuitively, the *input resources* of an attack are the necessary resources for the attack to succeed, and the *output resources* of the attack are the resources that the attack can supply if successful.

Compared with the approaches in [29, 83] which use predicates to describe prerequisites and consequences, our input/output resources based approach has several advantages. (1) When using predicates to specify prerequisites and consequences for each type of attacks, it may introduce too many predicates. Whereas input and output resource types are rather limited compared with the types of predicates and are easy to specify. (2) Since different experts may use different predicates to represent the same condition, or use the same predicate to represent different conditions, it is usually not easy to discover implication relationships between predicates and match consequences with prerequisites. Whereas input and output resource types are rather stable, straightforward to match and easy to accommodate new attacks. This correlation method is presented in Chapter 6.

1.6 Privacy-Preserving Alert Correlation: A Generalization Based Approach

In recent years, the security threats from infrastructure attacks such as worms and distributed denial of service attacks are increasing [19]. They affect large numbers of hosts and services on the Internet, and may bring serious financial loss. To defend against these attacks, the cooperation among different organizations is necessary. Several organizations such as CERT Coordination Center [17] and DShield [106] collect data (including security incident data) over the Internet, perform correlation analysis, and disseminate information to users and vendors. The security incident data are usually collected from different companies, organizations or individuals, and their privacy concerns have to be considered. To prevent the misuse of incident data, appropriate data sanitization through which the sensitive information is obfuscated is highly preferable. For example, DShield [106] lets audit log submitters perform partial or complete obfuscation to destination IP addresses in the datasets, where partial obfuscation changes the first octet of an IP address to decimal 10, and complete obfuscation changes any IP address to a fixed value 10.0.0.1.

As we mentioned earlier, to protect networks and hosts on the Internet, many security systems such as IDSs and firewalls are widely deployed. To better understand the security threats,

it is necessary to perform alert correlation. Current alert correlation approaches generally assume all alert data (e.g., the source and destination IP addresses) are available for analysis, which is true when there are no privacy concerns. However, when multiple organizations provide sanitized alert and incident data (because of privacy concerns) for intrusion analysis, alert correlation will be affected due to the lack of precise data. It is desirable to have techniques to perform privacy-preserving alert correlation such that the privacy of participating organizations is preserved, and at the same time, alert correlation can provide useful results. To our best knowledge, [69] is the only paper addressing privacy issues in alert correlation, which uses hash functions (e.g., MD5) and keyed hash functions (e.g., HMAC-MD5) to sanitize sensitive data. This approach is effective in detecting some high-volume events (e.g., worms). However, since hash functions destroy the semantics of alert attributes (e.g., the loss of topological information due to hashed IP addresses), the interpretation of correlation results is non-trivial. In addition, hash functions may be vulnerable to brute-force attacks due to limited possible values of alert attributes, and keyed hash functions may introduce difficulties in correlation analysis due to the different keys used by different organizations.

In Chapter 7, we propose a privacy-preserving alert correlation approach through generalization based on concept hierarchies.

1.7 Privacy-Preserving Alert Correlation: A Perturbation Based Approach

As we mentioned in Section 1.6, to defend against large-scale distributed attacks such as worms and distributed denial of service (DDoS) attacks, it is usually desirable to deploy security systems such as intrusion detection systems (IDSs) over the Internet, monitor different networks, collect security related data, and perform analysis to the collected data to extract useful information. In addition, different organizations, institutions and users may also have the willingness to share their data for security research as long as their privacy concerns about the data can be fully satisfied. For example, Department of Homeland Security sponsors PREDICT [51] project to create a repository collecting network operational data for cyber security research.

Data generated by security systems may include sensitive information (e.g., IP addresses of compromised servers) that data owners do not want to disclose or share with other parties. It is always desirable and sometimes mandatory to anonymize sensitive data before they are shared and

correlated. To address this problem, In Chapter 8, we propose three perturbation based schemes to flexibly perform alert anonymization. These schemes are closely related but can also be applied independently. In Scheme I, we generate artificial alerts and mix them with original alerts. Attribute values related to any alert in the mixed set may or may not be real, which helps hide original attribute values. In Scheme II, we map sensitive attributes to random values based on concept hierarchies. In Scheme III, we propose to partition an alert set into multiple subsets and apply Scheme II in each subset independently. To evaluate privacy protection and guide alert anonymization, we define *local privacy* and *global privacy*, and use *entropy* to compute their values. Though we emphasize alert anonymization techniques in Chapter 8, to examine data usability, we further perform correlation analysis for data sets anonymized by our three schemes. We focus on computing similarity values between anonymized attributes and building attack scenarios from anonymized data sets.

1.8 Dissertation Organization

The remainder of this dissertation is organized as follows. Chapter 2 gives an overview of approaches in intrusion detection and intrusion alert correlation. Chapter 3 discusses the techniques that efficiently correlate intrusion alerts through adapting main memory index structures and query optimization techniques. Chapter 4 provides the methods that learn attack strategies from intrusion alerts. Chapter 5 presents approaches to hypothesizing and reasoning about attacks missed by IDSs. Chapter 6 presents an alert correlation method based on triggering events and common resources. Chapter 7 provides an approach for privacy-preserving alert correlation through generalization based on concept hierarchies. Chapter 8 discusses privacy-preserving alert correlation based on perturbation based techniques. Chapter 9 concludes this dissertation and points out some future research directions.

Chapter 2

Related Work

In 1980, James Anderson published his seminal work *Computer Security Threat Monitoring and Surveillance* [8], which introduced the concept of *intrusion detection*. Intrusion detection focuses on detecting activities that violate the system's security policy [10, 85]. Amoroso [7] defines *intrusion detection* as "the process of identifying and responding to malicious activity targeted at computing and networking resources." With the development of the Internet and the wide usage of networked systems, network intrusions have become a serious problem. Intrusion detection is necessary to detect the intrusions and take appropriate actions.

As we mentioned in Introduction, intrusion detection techniques can be roughly classified into two categories: misuse detection and anomaly detection [10, 85]. Misuse detection builds signatures (patterns) for known attacks, and raises alerts when it monitors the activities that match the signatures. Anomaly detection builds models (e.g., statistical profiles) for normal activities, and raises alerts when the monitored activities (significantly) deviate from the models. These misuse and anomaly detection systems are widely deployed to protect the security of hosts and networks. Correlation analysis of the alerts from these intrusion detection systems and other security systems, is crucial for security officers to understand the security threats from inside and outside sources and take appropriate actions.

In this chapter, we first discuss intrusion detection techniques, then review the methods for intrusion alert correlation and privacy-preserving techniques, and finally give an overview of our previous correlation method [83] (it is an approach based on prerequisites and consequences of attacks).

2.1 Intrusion Detection

In this subsection, we first review misuse detection systems such as *NetSTAT* [112, 111], USTAT [50], IDIOT [66] and ASAX [80]. For anomaly detection techniques, we further classify them into two classes: user activity based approaches and program behavior based approaches. We first discuss user activity based anomaly detection systems such as *SRI IDES* [56], ADAM [12] and W&S [108], and then we review program behavior based anomaly detection techniques such as [43].

2.1.1 Misuse Detection

Misuse detection systems detect intrusions through matching observed events with predefined attack signatures. Specifying attack signatures is crucial for misuse detection systems. For each known attack, NetSTAT [112, 111] creates the attack signature through state transition diagrams. State transition diagrams are directed graphs. In a state transition diagram, each state is represented by a set of assertions, which describes the state of the systems (e.g., the service name in a host), and the transitions between states are triggered by signature actions, which represent the events that are necessary for the attack to be successful (e.g., a message delivery between a source and a destination host). NetSTAT has four components: (1) a network fact base, (2) a state transition scenario database, (3) a set of probes, and (4) an analyzer. Through the interaction between these four components, NetSTAT monitors the events occurring in the network and raises alerts when suspicious events are observed. Likewise, USTAT [50] also uses state transition diagrams to specify attack signatures. IDIOT [66] applies Colored Petri Nets to specify attack signatures. ASAX [80] uses a rule-based language (RUSSEL) to describe attack signatures. The IDSs NetSTAT, USTAT, IDIOT and ASAX can detect the attacks with the corresponding attack signatures defined in the systems. Their limitation is that if novel attacks are created with no corresponding signatures available in the systems, these IDSs are not able to detect them.

2.1.2 Anomaly Detection

Based on the subjects being monitored, we divide anomaly detection techniques into two classes: user activity based techniques and program behavior based techniques. User activity based approaches create statistical models based on the user's historical data, and raise alerts when the

user's activity is significantly deviate from the statistical model. Program behavior based approaches create system call models for the normal execution of the programs, and raise alerts when monitored system call sequences do not satisfy the model.

SRI IDES [56], ADAM [12] and W&S [108] are anomaly detection systems based on user activities, which flag alerts based on audit record data. Security auditing systems can generate audit records for each user's activities. SRI IDES examines these audit records and tries to determine whether the corresponding user's activities are abnormal or not. Notice that each audit record may include several aspects about the activities, for example, the files being accessed, and the CPU processing time. Based on the statistics such as the users' access frequencies, means and covariances (these are called the users' profiles), for each audit record, SRI IDES can calculate the test statistic value to measure the abnormality of the corresponding user's activities. If this value is large, the corresponding activities would be considered as abnormal. And a value close to zero would suggest the activities are normal. ADAM [12] detects intrusions through applying association analysis and clustering analysis [49] to network connections, where association analysis is used to identify the suspicious connections, and clustering analysis is used to classify the suspicious connections into known attacks, unknow attacks, or false alerts. W&S [108] automatically creates detection rules based on the statistics of historical audit records, and flags anomalies when a user's activities are "unusual." The advantage of SRI IDES, ADAM and W&S is that they may detect the attacks that are significantly deviate from the normal activities. Their limitation is that they may not detect stealthy attacks.

Program behavior based techniques target on monitoring the behaviors of programs. Once a program's behavior does not satisfy a predefined model, an alert is flagged. A critical question to these techniques is how to build normal behavior models for programs. To our knowledge, all proposed techniques monitor program behaviors through monitoring system calls that the program invokes. Current approaches to building system call models can be roughly classified into two categories: program training based approaches, and static analysis based approaches.

The seminal work [43] proposed by Forrest et al. is one of the approaches based on program training. During the (attack-free) training phase, different sequences of system calls have been observed. These long system call sequences are chopped up into short sequences (the lengths of sequences are 5, 6, and 11) and are put into a database representing the program's normal behaviors. In the detection phase, the system calls that a program invokes are monitored and matched with the system call sequences in the database. Once there exists a mis-match, an alert is triggered. Given system call sequences, Warrander et al. [114] further studied different data modeling methods to

specify the program's normal behaviors. They tested four methods: simple enumeration of observed sequences, comparison of relative frequencies of different sequences, a rule induction technique, and Hidden Markov Models (HMMs). Based on the experimental results, HMMs produce the most accurate models. Warrander et al. also pointed out that even simple modeling methods can perform well given enough system call sequences. [96] proposes to use a compact finite state automaton (FSA) to learn the program's normal behaviors. This automaton based modeling is able to capture both long and short term behaviors (in term of system calls), and may reduce false positives. Feng et al. [42] further incorporated call stack information into the model. Their approach is to retrieve return addresses from call stacks, and construct virtual paths between system calls. Their approach can detect some intrusions that may be missed by other approaches.

Static analysis based approaches such as [113, 41, 47] analyze the program source code or executable to formalize the program's normal behaviors. [113] proposes to specify the program's normal behaviors through analyzing program source code. Through source code analysis, [113] proposes four models to capture the normal behaviors of programs: (1) the trivial model, (2) the callgraph model, (3) the abstract stack model, and (4) the digraph model. As an example, the callgraph model describes the program behaviors through a non-deterministic finite automaton. In this automaton, each node is a state, and each edge (labeled with a system call) between nodes represents a transition triggered by the corresponding system call. The benefit of this approach is that it is free of false positives. [41] formally analyzes the abstract stack model (it is also referred as pushdown automaton (PDA) model). Due to the non-determinism of stack activities, the operation of PDA model is inefficient. [41] proposes two techniques to determinize PDA model: the observational technique and the instrumentation technique. The first technique implemented by VPStatic model extracts the stack activity information. The second technique implemented by Dyck's model transforms the program to add more code to expose the program states. Giffin, Jha, and Miller [47] propose to statically analyze the program binary to create the model of normal behaviors. During detection, the system monitors the system calls that the program invokes, and the system calls are allowed to make only if they satisfy the model. [47] further introduces two techniques for program transforms: renaming and null call insertion, which may affect the model's precision and efficiency.

2.2 Alert Correlation

Current intrusion detection systems suffer from several limitations. First, an intrusion detection system may generate thousands of alerts a day [61], thus overwhelming security officers. Second, among the alerts reported by intrusion detection systems, false alerts may be combined with true alerts, and it is challenging to differentiate between them. Third, intrusion detection systems cannot detect all attacks (i.e., they may miss some attacks). Alert correlation, focusing on discovering the relationships between individual alerts raised by intrusion detection systems and other security systems, is necessary to address these challenges.

Several alert correlation techniques have been proposed over the past a few years. These techniques can be roughly classified into four categories: (1) the approaches based on the similarity between alerts [33, 91, 109, 98, 60, 61], (2) the approaches based on the predefined attack scenarios [78, 36, 34], (3) the approaches based on prerequisites and consequences [29, 83, 82, 84, 102], and (4) the approaches based on multiple information sources [90, 79, 115].

Similarity based approaches [33, 91, 109, 98, 60, 61] perform alert correlation through measuring the similarity between alerts. Each alert usually has several attributes associated with them, for example, source and destination IP addresses. A natural way to discover the relationships between alerts is to measure the similarity between alert attributes. For example, if two alerts have the same source and destination IP addresses, it may be possible that the corresponding attacks are launched by the same attacker. If the alerts are similar through calculating the similarity between their attributes, they can be put into the same group to facilitate the future analysis. One critical issue in these approaches is how to define similarity measure. Traditional similarity measures used in data mining [49, 63] may not be appropriate for alert correlation, because many alert attributes are categorical (e.g., TCP/UDP Port numbers) rather than numerical. Several techniques have been proposed to solve this problem. In particular, Julisch et al. [62, 60] use conceptual clustering and generalization hierarchy to aggregate alerts into clusters.

The predefined attack scenarios based approaches correlate alerts based on known scenario templates, which are patterns of known sequences of attacks consisting of individual attack steps. Such methods then match IDS alerts to attack steps in the scenario templates (in a similar way to misuse detection). Examples in this category include [36, 34, 78]. Some approaches in this category specify attack scenarios through attack languages such as STATL [40] and Chronicles [78]. For example, [78] models attack scenarios through chronicle language (a chronicle is a set of events that are connected by temporal constraints). [36] proposes to correlate alerts based on the explicit or

derived rules. [34] builds attack scenarios through comparing probabilities that an alert may be in a set of scenarios, and always choosing the most possible scenario to add a new alert. The probability measure is derived based on training data. Though effective at recognizing known attack scenarios, a limitation of these techniques is that they cannot discover novel attack scenarios.

The prerequisites and consequences based approaches [29, 83, 82, 84, 102] model each attack through describing its prerequisite (the necessary condition to launch an attack successfully) and its consequence (the outcome if an attack succeeds). Through matching the consequences of earlier attacks with the prerequisites of later ones, these approaches link different attacks together to build attack scenarios. These techniques have the potential to discover novel attack scenarios. However, specifying prerequisites and consequences of attacks requires knowledge of individual attacks, and is time-consuming and error-prone.

The multiple information sources based approaches [90, 79, 115] process alerts from several security systems such as IDS sensors, firewalls, vulnerability scanners and anti-virus tools.

- [90] uses M-Correlator to process alerts, where alert processing can be divided into four stages. In the first stage, M-Correlator applies dynamically controllable filters to remove low-interest alerts for various subscribers. In the second stage, the alerts are examined based on the known network topology, and a relevance score is assigned through comparing the alert's related network topology with the know vulnerabilities. In the third stage, M-Correlator calculates the priority for each alert, which denoting the severity of the alert. In the last stage, each alert is assigned an incident rank, which denotes the overall degree that the alert's corresponding incident affects the network's mission. The purpose of M-Correlator is to reduce the number of alerts and to evaluate the severity of alerts for different analysts. Thus it usually does not provide high-level attack scenarios.

- [79] proposes a formal model M2D2 for alert correlation. M2D2 performs alert correlation using four types of information: (1) the characteristics of the information system (e.g., network topologies), (2) vulnerability information (e.g., an SNMP vulnerability CAN-2002-0012), (3) security tool information (e.g., an IDS sensor or a vulnerability scanning tool), and (4) monitored events (e.g., an IDS sensor flags an *FTP_Glob_Expansion* alert). These four types of information are formally defined. Based on these formally defined concepts, M2D2 examines the relationship between them and performs alert correlation. In [79], M2D2 focuses on alert aggregation, and how to use M2D2 for other purposes such as building attack scenarios is still to be explored.

- [115] proposes an architecture DOMINO (Distributed Overlay for Monitoring InterNet Outbreaks) for distributed intrusion detection. Multiple IDS sensors in DOMINO are deployed in various locations. Each IDS is responsible for monitoring the corresponding local network and hosts, and different IDSs also share their intrusion data and collaborate with each other to detect global coordinated attacks (e.g., Internet worms). There are several challenges involved in DOMINO such as how to effectively sharing intrusion data.

Notice different security systems may put different emphases on protecting the network components and applications. Combining them can potentially obtain more comprehensive understanding about the security of the protected systems. However, we notice that the information provided by different sources may be syntactically or semantically different, or even conflict with each other. How to reconcile them remains challenging.

2.3 Privacy-Preserving Techniques

Privacy-preserving techniques need to balance the requirements of data privacy as well as data usability. There are several privacy-preserving techniques have been proposed in the field of statistical databases and data mining. However, to our best knowledge, the approach proposed by Lincoln et al. [69] is the only paper to address privacy issues in the field of alert correlation. Here we give an overview of these privacy-preserving techniques.

To protect the privacy of alert data, privacy-preserving techniques need to hide or obfuscate sensitive attribute values for individual alerts. There are several techniques that can possibly achieve this goal, for example, cryptographic techniques such as data encryption and hash functions [72], data perturbation techniques [92, 68] used in statistical databases [1], or privacy-preserving data mining techniques [3, 2, 110].

Cryptographic techniques usually do not maintain the semantics of original values. They transform sensitive attribute values to an unintelligible form, and based on transformed values, it is difficult to know original values. Possible candidate techniques in this category are (secret-key) data encryption techniques such as DES and Triple DES [107], cryptographic hash functions such MD5 [93] and SHA-1 [39] [1], or keyed hash functions such HMAC-MD5 and HMAC-SHA1 [14, 65]. Comparing these different cryptographic techniques, we argue that hash functions (and keyed hash

[1] In Crypto 2004, researches announced that collisions have been found in MD4, MD5, HAVAL-128 and RIPEMD, however, no collisions have been found for SHA-1 [81].

functions) are more appropriate than data encryption techniques: (1) from performance perspective, hash functions generally are faster than encryption algorithms. For example, on a Pentium 4 CPU (2.1 GHz) with Windows XP Service Pack 1, MD5 performs 216.674MB/second, SHA-1 performs 67.977MB/second, while DES performs 21.340MB/second [32], and (2) from storage perspective, the length of ciphertext usually is longer than the corresponding hash output especially when the length of plaintext is long. Hash functions usually have fixed length of output. For example, the output of MD5 is 16 bytes, and the output of SHA-1 is 20 bytes, while the output of encryption could be arbitrarily large depending on plaintext (original values). We also argue that keyed hash functions may be more appropriate than hash functions, especially when an attribute only has a small number of possible values. For example, the possible port numbers are from 2^0 to $2^{16} - 1$, and the possible IP (IPv4) addresses are within $000/8$ to $255/8$. Hence hash functions may be vulnerable to brute-force (exhaustion) attacks. To defend from possible brute-force attacks, keyed-hash functions can be applied to attribute values. Since a secret key is introduced in keyed hash operations, original attribute values cannot be disclosed without the knowledge of the secret key. However, this technique may have difficulty in correlation analysis due to different keys introduced by different organizations. In addition, correlation result interpretation is non-trivial. The privacy-preserving alert correlation technique proposed by Lincoln et al. [69] uses both hash functions and keyed hash functions to sanitize sensitive alert attributes. Their approach is effective on detecting high-volume events such as worms, but may have the limitations we mentioned above.

The techniques proposed in statistical databases and privacy-preserving data mining may potentially be adapted to privacy-preserving alert correlation. These techniques may (asymptotically) preserve statistical properties (e.g., frequency count and mean) about original attribute values, while individual alert's attribute values are perturbed through randomness or other techniques. Attackers cannot estimate original attribute values with arbitrary precision.

Data distortion technique proposed by Liew et al. [68] is applicable to both categorical and continuous data. It first estimates the distribution of original dataset, then generates a new dataset with the estimated distribution, and finally replaces the original data with the new data.

Data swapping technique proposed by Reiss [92] is applicable to categorical attributes. This technique is based on a key concept: t-order frequency counts. Given q sensitive attributes, a t-order frequency count ($t = 0, 1, 2, \cdots, q$) is the number of alerts satisfying $a_{m1} = v_1 \wedge a_{m2} = v_2 \wedge \cdots \wedge a_{mt} = v_t$, where $a_{m1}, a_{m2}, \cdots, a_{mt}$ are attribute names, and $v_1, v_2, \cdots, v_t$ are attribute values. The basic idea of data swapping is to generate a new dataset that preserves the t-order frequency counts of the original dataset.

Data perturbation techniques such as [103, 3] are applicable to continuous attributes. The basic idea is to transform attribute value X to a new value X', where $X' = X + \delta$ and δ is an independent random variable. To control the perturbation, it is necessary to give the mean and the variance of δ, for example, $E(\delta) = 0$ and $Var(\delta) = 20^2$.

DShield [106] lets audit log submitters perform partial or complete obfuscation to destination IP addresses in data sets, where partial obfuscation changes the first octet of an IP address to decimal 10, and complete obfuscation changes any IP address to a fixed value $10.0.0.1$.

Our work in this thesis is also closely related to the k-*Anonymity* approaches [95, 100, 99], where an entity's information may be released only if there exist at least $k - 1$ other entities in the released data that are indistinguishable from this entity. These approaches also apply generalization hierarchies to help obfuscate attributes, where k is the pre-defined parameter to control the generalization process. Our approach in Chapter 7 differs in that we use entropy to control the attribute sanitization as well as to help design satisfactory concept hierarchies. Our approach in Chapter 8 uses concept hierarchies to facilitate artificial alert generation and attribute randomization. Moreover, we also study methods to correlate sanitized alerts in this thesis. Our work in this thesis is also related to packet trace anonymization techniques [88, 89]. For example, Pang and Paxson [88] propose a high-level language based approach to anonymize packet headers and payloads. These approaches are complementary to our work.

2.4 Previous Work: Alert Correlation Using Prerequisites and Consequences of Attacks

My thesis work is related to the alert correlation method proposed in [83]. In this section, we briefly describe this method with a slight modification, which simplify our discussion without losing the essence of the method.

2.4.1 An Overview of Correlation Method [83]

[83] proposes to correlate intrusion alerts based on the prerequisites and consequences of attacks. Intuitively, the *prerequisite* of an attack is the necessary condition to launch the attack successfully, and the consequence of an attack is the possible outcome if the attack does succeed.

For example, consider an attack *FTP_Glob_Expansion*. The prerequisite of this attack is a vulnerable *FTP* service running on the victim machine, and the consequence of this attack is the root privilege the attacker may possibly gain. Attackers usually launch a sequence of attacks to achieve their goals, where the earlier attacks usually prepare for the later ones. The connections between different attacks may be discovered through investigating the consequences of the earlier attacks and the prerequisites of the later ones. Based on these observations, given a set of attacks, we first identify the prerequisites and consequences for each attack, then we correlate the attacks through (partially) matching the consequences of earlier attacks with the prerequisites of later ones.

The correlation method [83] uses logical formulas (logical combinations of predicates) to represent the prerequisites and consequences of attacks. For example, aforementioned attack *FTP_Glob_Expansion* may gain root privilege on the victim host if it succeeds, thus we can use predicate *GainRootAccess(VictimIP)* to represent its consequence. For simplicity, [83] limits logical operations to AND ("$\wedge$") and OR ("$\vee$") in logical formulas.

Prerequisites, consequences and attributes of attacks are formalized as hyper-alert types (or alert types). A *hyper-alert type* (or *alert type*) is a triple (*fact, prerequisite, consequence*), where (1) *fact* is a set of alert attribute names associated with the corresponding domains, (2) *prerequisite* is a logical formula, and (3) *consequence* is a set of logical formulas. Note all the variables in *prerequisite* and *consequence* are in *fact*.

We give an example for hyper-alert types here. (For simplicity, we do not list the corresponding domain for each attribute.) A *FTP_Glob_Expansion* hyper-alert type can be defined as *FTP_Glob_Expansion* = ({*SrcIP, SrcPort, DestIP, DestPort*}, *ExistService* (*DestIP, DestPort*) $\wedge$ *VulnerableFTPRequest(DestIP)*), {*GainRootAccess*(*DestIP*)}), where four attributes *SrcIP, SrcPort, DestIP, DestPort* are used to describe the attack, the prerequisite of the attack is that a *FTP* service runs on host *DestIP* at port *DestPort* and this service is vulnerable to certain request, and the consequence is attackers may gain root privilege on host *DestIP*.

We notice that manually specifying hyper-alert types is time-consuming and error-prone. To facilitate the hyper-alert type specification, a practical way is to predefine a set of predicates (which can be extended if necessary), classify attacks into different categories based on certain criteria (e.g., the resources related to attacks), and then look for desirable predicates in prerequisites and consequences for each category of attacks.

Given a hyper-alert type *T* = (*fact, prerequisite, consequence*), a *type T alert t* is a tuple on *fact*, and this tuple is associated with an interval-based timestamp [*begin_time, end_time*]. A *type T hyper-alert h* is a finite set of type *T* alerts. The notion of hyper-alerts provide us the flexibility of

treating multiple alerts with the same type collectively.

The correlation method in [83] is aimed at discovering attack scenarios among intrusion alerts, where an attack scenario is a sequence of attacks that the adversaries launch to achieve their goal. [83] discovers attack scenarios through identifying the *prepare-for* relations between hyper-alerts (alerts).

Intuitively, a *prepare-for* relation exists if an earlier alert *contributes* to the prerequisite of a later one. In the formal model, alert correlations are performed via prerequisite and consequence sets. Given a hyper-alert type *T* = (*fact, prerequisite, consequence*), the *prerequisite set* (or *consequence set*, resp.) of T, denoted *Prereq*(T) (or *Conseq*(T), resp.), is the set of all predicates that appear in *prerequisite* (or *consequence*). The *expanded consequence set* of T, denoted *ExpConseq*(T), is the set of all predicates implied by *Conseq*(T) (e.g., *OSSolaris(DestIP)* implies *OSUnix(DestIP)*). Thus we have *Conseq*$(T) \subseteq$ *ExpConseq*(T). Given a type T alert t, the *prerequisite set*, *consequence set*, and *expanded consequence set* of t, denoted *Prereq*(t), *Conseq*(t), and *ExpConseq*(t), respectively, are the instantiated predicates in *Prereq*(T), *Conseq*(T), and *ExpConseq*(T) with arguments replaced by the corresponding attribute values of t. Alert t_1 *prepares for* alert t_2 if $t_1.end\_time < t_2.begin\_time$ and there exist instantiated predicates $c \in$ *ExpConseq*(t_1) and $p \in$ *Prereq*(t_2) such that $c = p$. Similarly, consider two hyper-alerts h_1 and h_2. h_1 prepares for h_2 if there exist alerts $t_1 \in h_1$ and $t_2 \in h_2$ such that t_1 prepares for t_2. For convenience, we may also refer to prepare-for relations as *causal relations* in this thesis.

An alert (or hyper-alert) correlation graph is used to represent a sequence of correlated alerts (or hyper-alerts). Formally, an *alert (or hyper-alert) correlation graph* CG = (N, E) is a connected directed acyclic graph, where any $n \in N$ is an alert (or a hyper-alert), and each directed edge $(n_1, n_2) \in E$ denotes that n_1 *prepares for* n_2. Note that a hyper-alert correlation graph is acyclic, since if one attack *prepares for* the other, then the former must occurs before the latter. As an example, Figure 2.1 shows an alert correlation graph adapted from [83]. The numbers inside the nodes represent the alert IDs, and the types of alerts are marked below the corresponding nodes. For brevity, we refer to an alert correlation graph (or a hyper-alert correlation graph) as a *correlation graph* in this report. For brevity, we also refer to this correlation method as the *causal correlation method*, since its goal is to discover the causal relations between alerts.

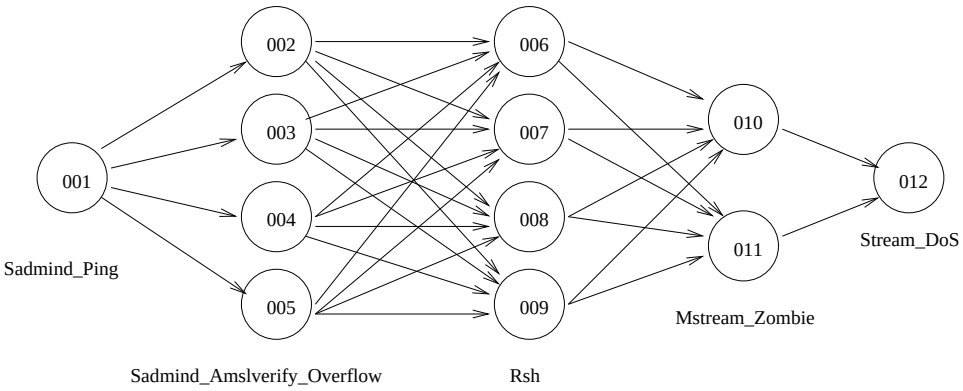


Figure 2.1: An example of alert correlation graphs

2.4.2 Implementation of [83]

We have implemented the correlation method based on prerequisites and consequences in [83]. To save our development efforts, our initial implementation takes advantage of the database systems. We store all hyper-alert types and alert data into a Microsoft SQL Server 2000 database. We use Java as a programming language, and use JDBC (JDBC is a component in Java to access databases) to interact with the database to perform correlation. For each alert, the predicates in the prerequisite and expanded consequence sets are instantiated as strings (e.g., *GainRootAccess(DestIP)* may be instantiated as *GainRootAccess(10.10.1.1)*) using alert attributes and saved into the tables *PrereqSet* and *ExpandedConseqSet*, respectively. To facilitate the correlation, we also store the corresponding alert (or hyper-alert) IDs and timestamps in the aforementioned tables. Thus each table has four columns *AlertID* (or *HyperAlertID*), *InstantiatedPredicate*, *begin_time*, and *end_time*. As a result, the matching of the consequences of the earlier alerts with the prerequisites of the later alerts can be performed through the following SQL statement [83].

```
SELECT DISTINCT c.AlertID, p.AlertID
FROM PrereqSet p, ExpandedConseqSet c
WHERE p.InstantiatedPredicate = c.InstantiatedPredicate
        AND c.end_time < p.begin_time
```

Based on the output of the above SQL statement, we can identify all *prepare-for* relations

among the given alerts. Connecting these *prepare-for* relations can provide us the attack scenarios hidden in the alert datasets.

This correlation method is effective according to the experiments with 2000 DARPA intrusion detection scenario specific data sets [77] and DEF CON 8 Capture The Flag (CTF) event datasets [37]. For details of these experimental results, please refer to [82, 83].

Chapter 3

Adapting Query Optimization Techniques for Efficient Correlation

As we mentioned in Chapter 2, to assist the analysis of intrusion alerts, several alert correlation methods (e.g., [34, 36, 109]) have been proposed recently to process the alerts reported by IDS. As one of these methods, we have been developing intrusion alert correlation and analysis techniques based on prerequisites and consequences of attacks [82, 83]. Intuitively, the prerequisite of an intrusion is the necessary condition for the intrusion to be successful, while the consequence of an intrusion is the possible outcome of the intrusion. Based on the prerequisites and consequences of different types of attacks, our method correlates alerts by (partially) matching the consequence of some previous alerts and the prerequisite of some later ones.

We have implemented an offline intrusion alert correlator using our previous correlation method [83]. Notice that to save our development efforts, our initial implementation is a DBMS-based application [83]. Involving a DBMS in the alert correlation process provided enormous convenience and support in our initial implementation; however, relying entirely on the DBMS also introduced performance penalty. For example, to correlate about 65,000 alerts generated from the DEF CON 8 CTF dataset, it took the DBMS-based intrusion alert correlator around 45 minutes with the JDBC-ODBC driver included in Java 2 SDK, Standard Edition, and more than 4 minutes with the Microsoft SQL Server 2000 Driver for JDBC. Such performance is clearly not sufficient

to make alert correlation a practical tool, especially for interactive analysis of intensive alerts. Our timing analysis indicates that the performance bottleneck lies in the interaction between the intrusion alert correlator and the DBMS. Since our current intrusion alert correlator completely relies on the DBMS, processing of each single alert entails interaction with the DBMS, which introduces significant performance overhead.

To address this problem, we propose to perform alert correlation entirely in main memory, while only using the DBMS as the storage of intrusion alerts. We study several main memory index structures, including Array Binary Search [5], AVL Trees [4], B Trees [21], Chained Bucket Hashing [64], Linear Hashing [70], and T Trees [67], as well as some database query optimization techniques such as nested loop join and sort join [105] to facilitate timely correlation of intrusion alerts. By taking advantage of the characteristics of the alert correlation process, we develop three techniques named *hyper-alert container, two-level index*, and *sort correlation*, which further reduce the execution time required by alert correlation.

We performed a series of experiments to evaluate these techniques with the DEF CON 8 CTF data set [37]. The experimental results demonstrate that (1) hyper-alert containers improve the efficiency of index structures with which an insertion operation involves search (e.g., B Trees, T Trees), (2) two-level index improves the efficiency of all index structures, (3) a two-level index structure combining Chained Bucket Hashing and Linear Hashing is most efficient for correlating streamed alerts, and (4) sort correlation with heap sort algorithm is most efficient for alert correlation in batch. With the most efficient method, the execution time for correlating the alerts generated from the DEF CON 8 CTF data set is reduced from over four minutes to less than one second.

3.1 Adapting Query Optimization Techniques

The essential problem in our approach is how to perform the SQL query (see "Implementation of [83]" in Chapter 2 for the query) efficiently. One option is to use database query optimization techniques, which have been studied extensively for both disk based and main memory based databases. However, alert correlation has a different access pattern than typical database applications; this may lead to different performance than traditional database applications. In addition, the unique characteristics in alert correlation may give us the opportunity for further improvement. Thus, in this and the next sections, we seek the possibilities to improve alert correlation by adapting existing query optimization techniques, evaluate various techniques and their adaptations, and

identify the most suitable ones for intrusion alert correlation.

In the following, we first go over some main memory index structures, and then present our adaptations for correlating streamed as well as batch alerts. In Section 3.2, we report our experimental results.

3.1.1 Main Memory Index Structures

Main memory index structures have been studied extensively in the context of search algorithms and main memory databases. Many different kinds of index structures have been proposed in the literature. In our study, we focus on the following ones: Array Binary Search [5], AVL Trees [4], B Trees [21], Chained Bucket Hashing [64], Linear Hashing [70], and T Trees [67].

In the following, we briefly describe these index structures. Detailed information can be found in the corresponding references. For comparison purpose, we also implement a naive, sequential scan method, which simply scans in an (unordered) array for the desired data item. We only care about insertion and search operations due to the need for alert correlation.

Sequential Scan is only implemented for reference purposes. In our study, Sequential Scan stores data items in an array. Search is performed by sequentially scanning the data items in the array, and insertion is simply to append to the end of the array.

Array Binary Search [64, 5] stores sorted data items in an array and locates the desired item via binary search. Array Binary Search is pretty efficient when searching in a static array. However, it has certain drawbacks in a dynamic environment. First, the array has to have enough space to accommodate new data items; otherwise, memory reallocation and copy of the entire array will have to be performed. In addition, even if there is enough space, insertion into the array involves $O(N)$ data movements.

AVL Trees [4] are balanced binary search trees. Each node in an AVL Tree contains a data item, control information, a left pointer which points to the subtree that contains the smaller data items (than the current data item), and a right pointer which points to the subtree that contains the bigger items (than the current data item). Search in an AVL Tree is very fast, since the binary search is intrinsic to the tree structure [67]. Insertion into an AVL Tree always involves a leaf node, and may lead to a rotation operation if it results in an unbalanced tree.

B Trees [21] are also balanced search trees. Unlike an AVL Tree, a node in a B Tree may have multiple data items and pointers. Data items in a B Tree node are ordered, and each pointer points to a subtree that consists of the data items that fall into the range identified by the adjacent data

items. B trees are shallower than AVL Trees, and thus involve less node accesses for a search operation. Insertion into a B Tree is fast, which usually involves only one node.

T Trees [67] are binary trees with many elements in a node, which evolved from AVL Trees and B Trees. The T Tree retains the intrinsic binary search nature of the AVL Tree, but it also has the good update and storage characteristics of the B Tree, since a T Tree node contains many elements. Search in a T Tree consists of a search in the binary tree followed by a search within a node. Insertion into a T Tree involves data movements within a single node, and possible rotations to rebalance the tree structure.

Chained Bucket Hashing [64] uses a static hash table and a chain of buckets for each hash entry. It is efficient in a static environment where the number of data items can be predetermined. However, in a dynamic environment in which the number of data items is not known (e.g., alert correlation), Chained Bucket Hashing may have poor performance. If the size of the hash table is too small, too many buckets may be chained for each hash entry; if the size of the hash table is too large, space may be wasted due to the empty entries.

Linear Hashing [70] uses a dynamic hash table, which splits hash buckets in predefined linear oder. Each time when the candidate bucket (i.e., the next bucket to split according to the linear order) overflows, Linear Hashing splits the candidate bucket into two, and the size of the hash table grows by one. The overflowed data items in the non-candidate buckets are placed in the overflow buckets for the same hash entries. The buckets are ordered sequentially, allowing the bucket address to be computed from a base address.

3.1.2 Correlating Streamed Intrusion Alerts

We first study alert correlation methods that deal with intrusion alert streams continuously generated by IDS. With such methods, an alert correlation system can be pipelined with IDS and produce correlation result in a timely manner.

Figure 3.1 presents a nested loop method that can accommodate streamed alerts. (As the name suggests, nested loop correlation is adapted from nested loop join [45].) It assumes that the input hyper-alerts are ordered ascendingly in terms of their beginning time. The nested loop method takes advantage of main memory index structures such as Linear Hashing and T Trees. While processing the hyper-alerts, the nested loop method maintains an index structure $\mathcal{I}$ for the instantiated predicates in the expanded consequence sets along with the corresponding hyper-alerts. Each time when a hyper-alert h is processed, the algorithm searches in $\mathcal{I}$ for each instantiated

Outline of Nested Loop Correlation
Input: A list H of hyper-alerts ordered ascendingly in their beginning times.
Output: All pairs of (h', h) such that both h and h' are in H and h' prepares for h.
Method:
Maintain an index structure $\mathcal{I}$ for instantiated predicates in the expanded consequence sets of hyper-alerts. Each instantiated predicate is associated with the corresponding hyper-alert. Initially, $\mathcal{I}$ is empty.
1. **for** each hyper-alert h in H (accessed in the given order)
2. **for** each instantiated predicate p in the prerequisite set of h
3. Search the set of hyper-alerts with index key p in $\mathcal{I}$. Let H' be the result.
4. **for** each h' in H'
5. **if** (h'.EndTime$<$ h.BeginTime) **then** output (h', h).
6. **for** each p in the expanded consequence set of h
7. Insert p along with h into $\mathcal{I}$.
end

Figure 3.1: Outline of the nested loop alert correlation methods

predicate p that appears in h's prerequisite set. A match of a hyper-alert h' implies that h' has the same instantiated predicate p in its expanded consequent set. If h'.EndTime is before h.BeginTime, then h' prepares for h according to the definition of *prepare-for* relation. If the method processes all the hyper-alerts in the ascending order of their beginning time, it is not difficult to see that the nested loop method can find all and only the prepare-for relations between the input hyper-alerts.

The nested loop correlation method has different performance if different index structures are used. Thus, one of our tasks is to identify the index structure most suitable for this method. In addition, we further develop two adaptations to improve the performance of these index structures. Our first adaptation is based on the following observation.

Observation 1 *Multiple hyper-alerts may share the same instantiated predicate in their expanded consequence sets. Almost all of them prepare for a later hyper-alert that has the same instantiated predicate in its prerequisite set.*

Observation 1 implies that we can associate hyper-alerts with an instantiated predicate p if p appears in the expanded consequence sets of all these hyper-alerts. As a result, locating an instantiated predicate directly leads to the locations of all the hyper-alerts that share the instantiated

predicate in their expanded consequence sets. We call the set of hyper-alerts associated with an instantiated predicate a *hyper-alert container*.

However, using hyper-alert containers does not always result in better performance. There are two types of accesses to the index structure in the nested loop correlation method (Figure 3.1): insertion and search. For the index structures that preserve the order of data items in them, insertion implies search, since each time when an element is inserted into the index structure, we have to place it in the "right" place. Using hyper-alert container does not increase the insertion cost significantly in this case, while at the same time reduces the search cost. However, for the non-order preserving index structures such as Linear Hashing, insertion does not involve search. Using hyper-alert containers would force to perform a search, since the hyper-alerts have to be put into the right container. In this case, hyper-alert container decreases the search cost but increases the insertion cost, and it is not straightforward to determine whether the overall cost is decreased or not. We study this through experiments in Section 3.2.

Our second adaptation is based on the following observation.

Observation 2 *There is a small, static, and finite set of predicates. Two instantiated predicates are the same only if they are instantiated from the same predicate.*

Observation 2 leads to a *two-level index structure*. Each instantiated predicate can be split into two parts, the predicate name and the arguments. The top-level index is built on the predicate names. Since we usually have a static and small set of predicate names, we use Chained Bucket Hashing for this purpose. Each element in the top-level index further points to a second-level index structure. The second-level index is built on the arguments of the instantiated predicates. When an instantiated predicate is inserted into a two-level index structure, we first locate the right hash bucket based on the predicate name, then locate the second-level index structure within the hash bucket (by scanning the bucket elements), and finally insert it into the second-level index structure using the arguments.

We expect the two-level index structure to improve the performance due to the following reasons. First, since the number of predicates is small and static, using Chained Bucket Hashing on predicate names is very efficient. In our experiments, the size of the hash table is set to the number of predicates, and it usually takes one or two accesses to locate the second-level index structure for a given predicate name. Second, the two-level index structure decomposes the entire index structure

Outline of Sort Correlation
Input: A set H of hyper-alerts.
Output: All pairs of (h', h) such that both h and h' are in H and h' prepares for h.
Method:

Prepare two arrays A_{pre} and A_{con}, each entry of which is a hyper-alert associated with a key field. Each array is initialized with a reasonable size, and reallocated with doubled sizes if out of space. Existing content is copied to the new buffer if reallocation happens.

1. **for** each h in H
2. **for** each p in the prerequisite set of h
3. Append h to A_{pre} with $key = p$.
4. **for** each p in the expanded consequence set of h
5. Append h to A_{con} with $key = p$.
6. Sort A_{pre} and A_{con} ascendingly in terms of the key field (with, e.g., heap sort).
7. Partition the entries in A_{pre} and A_{con} into maximal blocks that share the same instantiated predicate. Assume A_{pre} and A_{con} have B_{pre} and B_{con} blocks, respectively.
8. $i = 0$, $j = 0$.
9. **while** ($i < B_{pre}$ and $j < B_{con}$) **do**
10. **if** (A_{pre}.Block(i).InstantiatedPredicate $< A_{con}$.Block(j).InstantiatedPredicate) **then**
11. $i = i + 1$.
12. **else if** (A_{pre}.Block(i).InstantiatedPredicate $> A_{con}$.Block(j).InstantiatedPredicate) **then**
13. $j = j + 1$.
14. **else for** each h in A_{pre}.Block(i) and each h' in A_{con}.Block(j)
15. **if** $h'.EndTime < h.BeginTime$ **then** output (h', h).
16. $i = i + 1$, $j = j + 1$.

end

Figure 3.2: The sort correlation method

into smaller ones, and thus reduces the search time in the second-level index. We verify our analysis through extensive experiments in Section 3.2.

3.1.3 Correlating Intrusion Alerts in Batch

Some applications allow alerts to be processed in batch (e.g., forensic analysis with an alert database). Though the nested loop method discussed earlier is still applicable, there are more efficient ways for alert correlation in batch.

Figure 3.2 presents a sort correlation method, which is adapted from sort join [105]. The sort correlation method achieves good performance by taking advantage of efficient main memory

sorting algorithms. Specifically, it uses two arrays, A_{pre} and A_{con}. A_{pre} stores the instantiated predicates in the prerequisite sets of the hyper-alerts (along with the corresponding hyper-alerts), and A_{con} stores the instantiated predicates in the expanded consequence sets (along with the corresponding hyper-alerts). This method then sorts both arrays in terms of the instantiated predicate with an efficient sorting algorithm (e.g., heap sort).

Assume both arrays are sorted ascendingly in terms of instantiated predicate. The sort correlation method partitions both arrays into blocks that share the same instantiated predicate, and scans both arrays simultaneously. The sort correlation method maintains two indices, i and j, that references to the current blocks in A_{pre} and A_{con}, respectively. The method compares the instantiated predicates in the two current blocks. If the instantiated predicate in the current block of A_{pre} is smaller, it advances the index i; if the instantiated predicate in the current block A_{con} is smaller, it advances the index j; otherwise, the current blocks of A_{pre} and A_{con} share the same instantiated predicate. The method then examines each pair of hyper-alerts h' and h, where h' and h are in the current block of A_{con} and A_{pre}, respectively. If the end time of h' is before the beginning time of h, then h' prepares for h according to the definition of *prepare-for* relation.

It is easy to see that the sort correlation method can find all pairs of hyper-alerts such that the first prepares for the second. Consider two hyper-alerts h and h' where h' prepares for h. There must exist an instantiated predicate p in both the expanded consequence set of h' and the prerequisite set of h. Thus, p along with h' must be placed in the array A_{con}, and p along with h must be placed in the array A_{pre}. The scanning method in Figure 3.2 (lines 9 - 16) will eventually point i to p's block in A_{pre} and j to p's block in A_{con} at the same time, and thus output h' prepares for h. Therefore, the sort correlation can discover all and only pairs of hyper-alerts such that the first prepares for the second.

We also study the possibility of adapting two-index join and hash join methods [105] to improve the performance of batch alert correlation. However, our analysis indicates they cannot outperform nested loop correlation due to the fact that alert correlation is performed entirely in main memory.

A naive adaptation of two-index join leads to the following two-index correlation method: Build two index structures for the instantiated predicates in the prerequisite sets and the expanded consequence sets, respectively. For each instantiated predicate p, locate the hyper-alerts related to p in both index structures, and compare the corresponding timestamps. However, this method cannot perform better than the nested loop method. The nested loop method only involves insertion of instantiated predicates in the expanded consequence sets and search of those in the prerequisite sets.

In contrast, the above adaptation requires insertion of instantiated predicates in both prerequisite and expanded consequence sets, and search of instantiated predicates in at least one of the index structures.

A possible improvement over the naive adaptation is to merge the two index structures. We can associate two sets of hyper-alerts with each instantiated predicate p, denoted $H_{pre}(p)$ and $H_{con}(p)$, and build one index structure for the instantiated predicates. $H_{pre}(p)$ and $H_{con}(p)$ consist of the hyper-alerts that have p in their prerequisite sets and expanded consequence sets, respectively. After all the instantiated predicates in the prerequisite or the consequence set of the hyper-alerts are inserted into the index structure, we can simply scan all the instantiated predicates, and compare the corresponding timestamps of the hyper-alerts in $H_{pre}(p)$ and $H_{con}(p)$ for each instantiated predicate p. However, each insertion of an instantiated predicate entails a search operation, since the corresponding hyper-alert has to be inserted into either $H_{pre}(p)$ or $H_{con}(p)$. Thus, this method cannot outperform the nested loop method, which involves one insertion for each instantiated predicate in the expanded consequence sets, and one search for each instantiated predicate in the prerequisite sets. A similar conclusion can be drawn for hash join.

Another possibility to have a faster batch correlation is to use Chained Bucket Hashing. Since the number of alerts is known beforehand, we may be able to decide a relatively accurate hash table size, and thus have a better performance than its counter part for streamed alerts. We study this through experiments in Section 3.2.

3.1.4 Correlating Intrusion Alerts with Limited Memory

The previous approachs to in-memory alert correlation have assumed that all index structures fit in memory during the alert correlation process. This may be true for analyzing intrusion alerts collected during several days or weeks; however, in typical operational scenarios, the IDSs produce intrusion alerts continuously and the memory of the alert correlation system will eventually be exhausted. A typical solution is to use a "sliding window" to focus on alerts that are close to each other; at any given point in time, only alerts after a previous time point are considered for correlation. Such a method has been adopted by many IDSs such as ADAM [13].

We adopt a sliding window which can accommodate up to t intrusion alerts. The parameter t is determined by the amount of memory available to the intrusion alert correlation system. Each time when a new intrusion alert is coming, we check if inserting this new alert will result in more than t alerts in the index structure. If yes, we remove the oldest alert from the index structure.

In either case, we will perform the same correlation process as in Section 3.1.2. It is also possible to add multiple intrusion alerts in batch. In this case, multiple old alerts may be removed from the index structure. Note that though choosing a sliding *time* window is another option, it doesn't reflect the memory constraint we have to face in this application.

Using a sliding window in our application essentially implies deleting old intrusion alerts when there are more than t alerts in the memory. This problem appeared to be trivial at the first glance, since all the data structures have known deletion algorithms. However, we soon realized that we had to go through a little trouble to make the deletion efficient. The challenge is that the index structures we build in all the previous approaches are in terms of instantiated predicates to facilitate correlation. However, to remove the oldest intrusion alerts, we need to locate and remove alerts in terms of their timestamps. Thus, the previous index structures cannot be used to perform the deletion operation efficiently. Indeed, each deletion implies a scan of all the alerts in the index structures.

To address this problem, we add a *secondary data structure* to facilitate locating the oldest intrusion alerts. Since the intrusion alerts are inserted as well as removed in terms of their time order, we use a queue (simulated with a circular buffer) for this purpose. Each newly inserted intrusion alert also has an entry added into this queue, which points to its location in the *primary index structure* in terms of the instantiated predicates. Thus, when we need to remove the oldest intrusion alert, we can simply dequeue an alert, find its location in the primary index structure, and delete it directly. Indeed, this is more efficient than the generic deletion method of the order preserving index structures (e.g., AVL Trees), since deletion usually implies search in those index structures.

3.2 Implementation and Experiments

We have implemented all the techniques discussed in Section 3.1. All the programs are written in Java, with JDBC to connect to the DBMS. However, unlike our previous prototype system, the current implementation only uses the DBMS as the storage of hyper-alert types and hyper-alerts. All the processing of alerts is handled in main memory by the program. To make the execution time comparable, we reuse the code as much as possible, and make sure we use the most efficient way in coding.

Some index structures need array to store the data, which may need memory reallocation in dynamic environments. We implemented a simple memory reallocation strategy to handle all

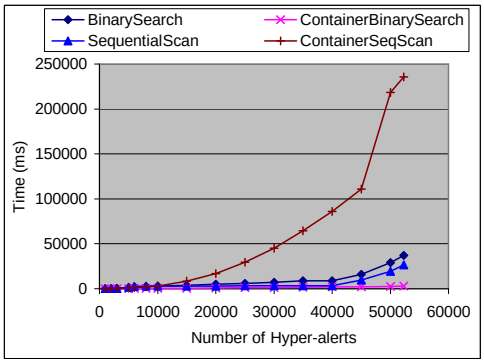
the array reallocation. Each array is initialized with a certain size. When the array is not enough, the program reallocates another array with a doubled size and copy over all the data items in the previous array.

Several index structures require some other parameters. For B Trees, we need to specify node size (i.e., how many data items to store in one B Tree node); for T Trees, we need minimum and maximum node sizes; for Chained Bucket Hashing and Linear Hashing, we need the bucket size (i.e., how many elements in each bucket). Different parameters may result in different performance. A common feature of these parameters is that both too large and too small values will result in poor performance. We found the experimentally optimal values for these parameters in the corresponding references, performed a series of experiments to compare the execution time, and picked the best values. As a result, the node size of B Trees is 7, the minimum and the maximum node sizes of a T Tree node are 8 and 10, respectively, the bucket size of Linear Hashing is 20, and the bucket size of Chained Bucket Hashing is 5.

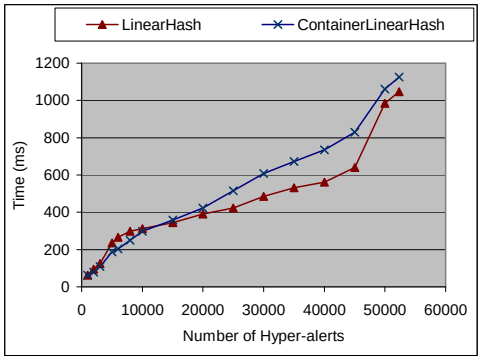
3.2.1 Experimental Results

We performed a series of experiments to compare the techniques discussed in Section 3.1. All the experiments were run on a DELL Precision Workstation with 1.8GHz Pentium 4 CPU and 512M memory. The alerts used in our experiments were generated by a RealSecure Network Sensor 6.0 [52], which monitors an isolated network in which we replayed the network traffic collected at the DEF CON 8 CTF event [37]. The Network Sensor was configured to use the *Maximum_Coverage* policy with a slight change, which forced the Network Sensor to save all the reported alerts.

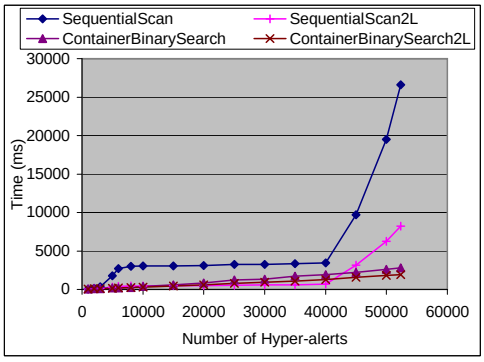
In these experiments, we mapped each alert type reported by the RealSecure Network Sensor to a hyper-alert type (with the same name), and generated one hyper-alert from each alert. The prerequisite and consequence of each hyper-alert type were specified according to the descriptions of the attack signatures provided by RealSecure. There are totally 65,058 hyper-alerts generated by the RealSecure Network Sensor, among which 52,318 hyper-alerts have prerequisite or consequence. The remaining hyper-alerts are mainly *Windows_Access_Error* and *IPDuplicate*, which we decided to ignore due to their overly general semantics. These hyper-alerts cannot be correlated with any other ones, and do not contribute to the time required by alert correlation. In order to precisely evaluate the relationship between the execution time and the number of hyper-alerts, we did not include them in our experiments.



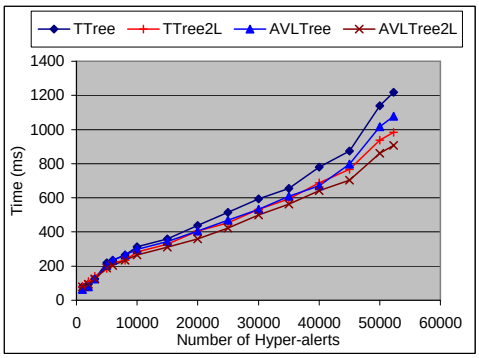
(a) Hyper-alert containers (1)



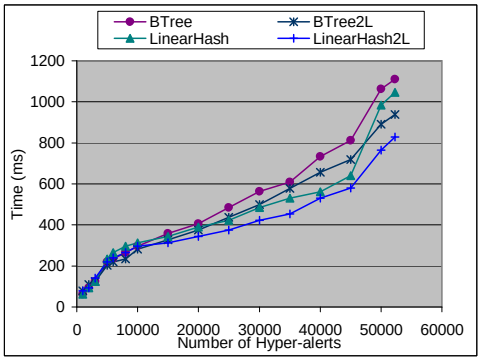
(b) Hyper-alert containers (2)



(c) Two-level index structures (1)



(d) Two-level index structures (2)

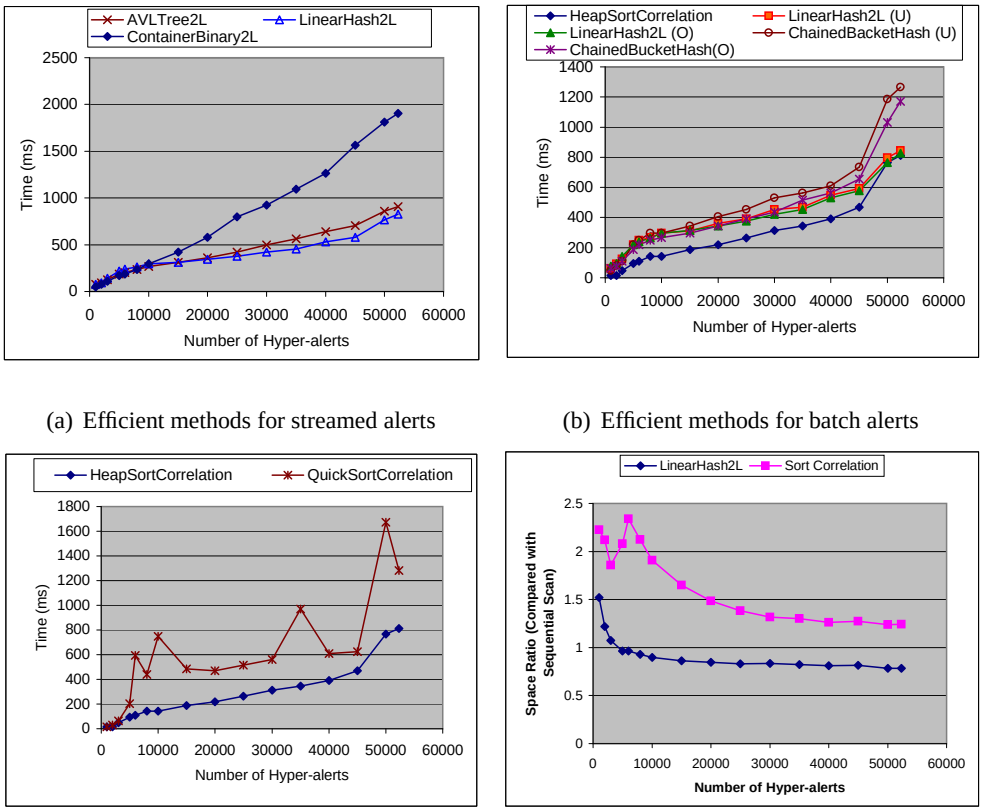


(e) Two-level index structures (3)

Figure 3.3: Experimental results (1)

Nested-Loop Correlation without Memory Constraint

Our first set of experiments was intended to evaluate the effectiveness of hyper-alert container in the nested loop correlation method. According to our analysis, hyper-alert container may



(a) Efficient methods for streamed alerts

(b) Efficient methods for batch alerts

(c) Impact of sorting algorithms

(d) Space overhead

Figure 3.4: Experimental results (2)

reduce the execution time if we use the order-preserving index structures. We compared the execution time for Sequential Scan, Array Binary Search, and Linear Hashing, with or without hyper-alert container. We did not perform a similar comparison for the tree index structures (i.e., T Tree, B Tree, and AVL Tree), since not having hyper-alert container not only increases both insertion and search cost, but also the complexity of the programs. As shown in Figures 3.3(a) and 3.3(b), hyper-alert container reduces the execution time for Array Binary Search, but increases the execution time for Sequential Scan significantly, and Linear Hashing slightly.

Our second set of experiments was intended to evaluate the effectiveness of two-level index structure in the nested loop correlation method. According to our analysis and the earlier experimental results, we used hyper-alert container in Array Binary Search and tree index structures, but not in Sequential Scan and Linear Hashing. As indicated by Figures 3.3(c) to 3.3(e), two-level

index reduces execution time for all index structures.

In Figure 3.3(c), the lines for Sequential Scan and two-level Sequential Scan have an interesting flat area when the number of input hyper-alerts is between 8,000 and 40,000. Our investigation revealed that the majority of hyper-alerts in this range do not have any prerequisite. Thus, processing of these hyper-alerts does not involve search (i.e., sequential scan) in a large array, and there is no big increase in execution time. In other words, the difference between insertion and search cost and the fact that there is not many searches for the hyper-alerts between 8,000 and 40,000 resulted in the flat area in Figure 3.3(c). In the other index structures, there is no significant difference between insertion and search costs. Thus, there is no dramatic change in execution time for the hyper-alerts between 8,000 and 40,000, though we can observe the slow down in the increase of execution time.

Our next goal is to find out which index structure (with or without the two adaptations) has the best performance for nested loop correlation. We take the fastest methods from Figure 3.3(c), 3.3(d), and 3.3(e), which are two-level Array Binary Search with hyper-alert container, two-level AVL Tree, and two-level Linear Hashing, and put them in Figure 3.4(a). The resulting figure shows both two-level AVL Tree and two-level Linear Hashing are significantly faster than two-level Array Binary Search with hyper-alert container, and two-level Linear Hashing outperforms two-level AVL Tree by up to 20%. Thus, nested loop correlation achieves the best performance with two-level Linear Hashing.

Batch Correlation (without Memory Constraint)

Our next set of experiments is focused on methods for correlating alerts in batch. Certainly, all the previously evaluated methods can be used for batch processing of intrusion alerts. Our evaluation here is to determine whether any method can achieve better performance than nested loop correlation with two-level Linear Hashing, the best method for correlating streamed alerts. For the index structures other than Chained Bucket Hashing, knowing the hyper-alerts before alert correlation will not change anything in the index structures. Thus, we believe their relative performance will not change for batch alert correlation. However, knowing how many hyper-alerts gives more information for Chained Bucket Hashing, since we can estimate the number of elements to be inserted into the hash table and thus have a good guess about the desired size of the hash table. In our experiments, we chose to set the hash table size the same as the number of input hyper-alerts. Moreover, the sort correlation method can potentially outperform nested loop correlation with two-level

Linear Hashing, since it adopts a different way to correlate the hyper-alerts. Thus, we decided to compare the execution time of nested loop correlation with two-level Linear Hashing, nested loop with Chained Bucket Hashing, and sort correlation. To further examine the impact of the time order of input hyper-alerts, we examined the timing results with ordered and unordered input. With input hyper-alerts not ordered in their beginning time, the algorithm must insert all of the instantiated predicates in the expanded consequence sets before it processes any instantiated predicate in the prerequisite sets. The time order of input does not have any impact on sort correlation.

Figure 3.4(b) shows the timing results of these methods. Surprisingly, Chained Bucket Hashing has the worst performance. Our further investigation explains this result: The average number of data items per hash entry is between 1.0 and 1.52; however, the maximum number of data items per hash entry is between 162 and 518. That is, the distribution of the instantiated predicates resulted in uneven distribution of hyper-alerts in the buckets. Having input hyper-alerts ordered by beginning time only reduced the execution time slightly differences for nested loop correlation with both two-level Linear Hashing and Chained Bucket Hashing. Finally, sort correlation with heap sort achieves the best performance among these four methods.

We also studied the impact of different sorting algorithms on the execution time of sort correlation. We compared two sorting algorithms, heap sort and quick sort. Heap sort has the least complexity in the worst case scenarios, while quick sort is considered the best practical choice among all the sorting algorithms [22]. Figure 3.4(c) shows the timing results of both algorithms: Sort correlation with quick sort performs significantly worse than the heap sort case. In addition, the execution time is not very stable in terms of the number of input hyper-alerts. This is because quick sort is sensitive to the input. In contrast, heap sort has stably increasing execution time as the number of hyper-alerts increases. Thus, we believe heap sort is a good choice for sort correlation.

Space Utilization

We examined the space overhead of these methods by comparing their space requirements with the sequential scan method. We use a quantitative measure *space ratio* = $\frac{\#bytes\ to\ use\ the\ method}{\#bytes\ to\ use\ sequential\ scan}$ for this purpose. As shown in Figure 3.4(d), the highest space ratios of sort correlation and nested loop correlation with two-level Linear Hashing are 2.34 and 1.52, respectively. Sort correlation requires about twice space as nested loop with two-level Linear Hashing, since it has to store instantiated predicates in both prerequisite and expanded consequence sets. In addition, nested loop correlation with two-level Linear Hashing requires more space than sequential scan when the input

size is small, but less space when the input size is large. This is because two-level Linear Hashing usually has wasted cells in the hash table when the input size is small. When the input size is large, not only the hash table is better utilized, but storing predicate names in the top level index can also reduce the storage requirement. The spike in the line of sort correlation is due to the irregular distribution of instantiated predicates in the prerequisite sets, which are only saved in sort correlation, but not in the nested loop correlation method. (There is a sudden increase of hyper-alerts that only have prerequisites between 3,000 and 6,000 input hyper-alerts.)

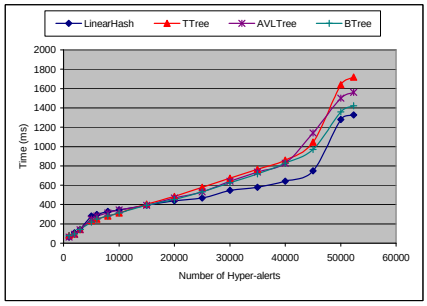
Nested-Loop Correlation with Memory Constraint

Our last set of experiments is focused on evaluating the efficiency of different indexing structures when there is memory constraint. Based on our prior experimental results, we only compare the execution time of AVL Tree, T Tree, B Tree, and Linear Hashing. We do not consider Sequential Scan and Array Binary Search because of their poor performance (in insertion and search). It's quite clear that their performance will not be comparable with the other methods.

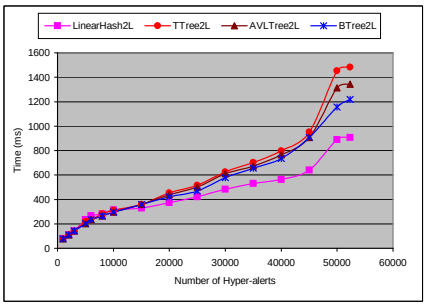
In this set of experiments, we first use a sliding window of size 30,000 to compare the execution time for different number of input hyper-alerts. As shown in Figure 3.5(a), when the two-level index structure is not used, Linear Hashing has the best performance compared with the three tree based indexing structures. Figure 3.5(b) shows a similar performance order, when the two level index structure is used. We also notice that B Trees perform the best among the tree based index structures, whereas AVL Tree is the best when there is no memory constraint. Our further investigation indicates that the deletion algorithm of AVL Tree is not only more complex than that of B Tree, but also more complex than the insertion algorithm of AVL Tree. In an AVL Tree, one deletion may trigger several subtree rotations. As a result, more operations are need to rebalance the tree. Figure 3.5(c) further shows the comparison of Linear Hashing and B Tree with and without the two-level index structure. The result shows that the two-level index does improve the efficiency of the index structures, and two-level Linear Hashing is the most efficient one among all the index structures.

To reconfirm the performance results, we perform another set of experiments with varying sliding window sizes, using all of the hyper-alerts as input. Figures 3.5(d), 3.5(e), and 3.5(f) show the results. These results indicate that two-level Linear Hashing is the most efficient and the two level index structure improves the performance for all four methods.

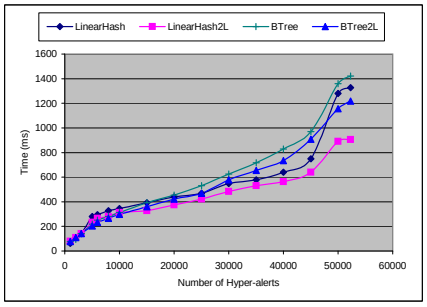
An interesting observation is that there is a bump in the line for T Tree in both Figure



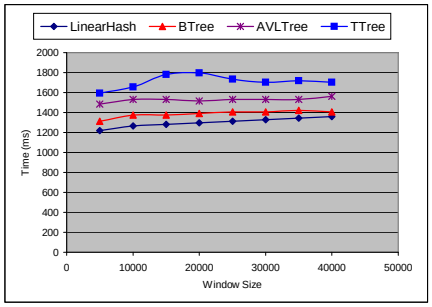
(a) Single-level index structures with fixed window size



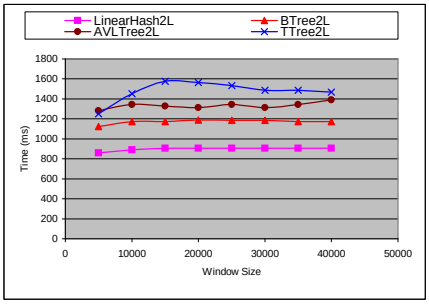
(b) Two-level index structures with fixed window size



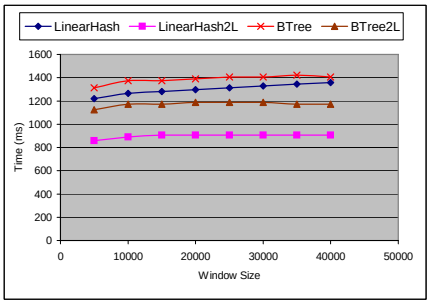
(c) Selected efficient index structures with fixed window size



(d) Single-level index structures with varying window size



(e) Two-level index structures with varying window size



(f) Selected efficient index structures with varying window size

Figure 3.5: Experimental results of correlations with memory constraint

3.5(d) and Figure 3.5(e) when the window size is between 15,000 and 25,000. Our investigation reveals that the numbers of node balancing operations for these window sizes are more than the other

window sizes. (There are 16,611, 16,684, and 16,232 node balancing operations for the window sizes 15,000, 20,000, and 25,000, respectively.)

3.3 Summary

This chapter studies main memory index structures and database query optimization techniques to facilitate timely correlation of intensive alerts. In addition to experimental study of the performance of various main memory index structures, this chapter presents three techniques named *hyper-alert container, two-level index,* and *sort correlation* by taking advantage of the characteristics of the alert correlation process. The experimental study demonstrates that (1) hyper-alert containers improve the efficiency of order-preserving index structures, with which an insertion operation involves search, (2) two-level index improves the efficiency of all index structures, (3) a two-level index structure combining Chained Bucket Hashing and Linear Hashing is most efficient for streamed alerts, (4) sort correlation with heap sort algorithm is the most efficient for alert correlation in batch, and (5) two-level Linear Hashing is the most efficient for alert correlation when sliding window is used to cope with memory constraint. Though these observations are based on the experiments for DEF CON 8 CTF event data sets, we expect some observations (e.g., (1) and (2)) are also applicable to other data sets especially when data sets are large.

Chapter 4

Learning Attack Strategies from Intrusion Alerts

It has become a well-known problem that current intrusion detection systems (IDSs) produce large volumes of alerts, including both actual and false alerts. As the network performance improves and more network-based applications are being introduced, the IDSs are generating increasingly overwhelming alerts. This problem makes it extremely challenging to understand and manage the intrusion alerts, let alone respond to intrusions timely.

It is often desirable, and sometimes necessary, to understand attack strategies in security applications such as computer and network forensics and intrusion responses. For example, attack strategies may be used to profile hackers or hacking tools in computer and network forensics. As another example, it is easier to predict attacker's next move, and reduce the damage caused by intrusions, if the attack strategy is known during intrusion response. However, in practice, it usually requires that human users analyze the data collected during intrusions manually to understand the attack strategy. This process is not only time-consuming, but also error-prone. An alternative to manual analysis is to list all possible attack strategies using vulnerability analysis tools such as attack graphs [97, 6]. However, these tools require a predefined security property so that they can use model checking techniques to identify possible attack sequences that may lead to the violation of the security property.

In this chapter, we present techniques to automatically learn attack strategies from intrusion alerts reported by IDSs. Our approach is based on the alert correlation methods [83, 29]. By examining correlated intrusion alerts, our method extracts the constraints intrinsic to the attack strategy automatically. Specifically, an attack strategy is represented as a directed acyclic graph (DAG), which we call an *attack strategy graph*, with nodes representing attacks, edges representing the (partial) temporal order of attacks, and constraints on the nodes and edges. These constraints represent the conditions that any attack instance must satisfy in order to use the strategy. To cope with variations in attacks, we use generalization techniques to hide the differences not intrinsic to the attack strategy.

To facilitate intrusion analysis in applications such as computer and network forensics, we further develop techniques to measure the similarity between sequences of intrusion alerts based on their attack strategies. Similarity measurement of alert sequences is a fundamental problem in many security applications such as profiling hackers or hacking tools, identification of undetected attacks, attack prediction, and so on. To achieve this goal, we harness the results on error tolerant graph/subgraph isomorphism in the pattern recognition field. By analyzing the semantics and constraints in similarity measurement of alert sequences, we transform this problem into error tolerant graph/subgraph isomorphism problem.

Our contribution in this chapter is three-fold. First, we develop a model to represent attack strategies as well as algorithms to extract attack strategies from correlated alerts. Second, we develop techniques to measure the similarity between sequences of alerts on the basis of the attack strategy model. Third, we perform a number of experiments to validate the proposed techniques. Our experimental results show that our techniques can successfully extract invariant attack strategies from sequences of alerts, measure the similarity between alert sequences conforming to human intuition, and identify attacks possibly missed by IDSs. The details of our approach are given in the following sections.

4.1 Modeling Attack Strategies

In this section, we present a method to represent and automatically learn attack strategies from a sequence of related intrusion alerts. Our method is developed by extending the alert correlation model [83].

4.1.1 Attack Strategy Graph

The goal of attack strategy modeling is to capture the invariants in attack strategies that do not change across multiple instances of attacks. The strategy behind a sequence of attacks is indeed about how to arrange earlier attacks to prepare for the later ones so that the attacker can reach his/her final goal. Thus, the *prepare-for* relations between the intrusion alerts (*i.e.*, detected attacks) is intrinsic to attack strategies. However, in method [83], the *prepare-for* relations are between specific intrusion alerts; they do not directly capture the conditions that have to be met by related attacks. To facilitate the representation of the invariant attack strategy, we transform the *prepare-for* relation into some common conditions that have to be satisfied by *all* possible instances of the same strategy. In the following, we formally represent such a condition as an *equality constraint*.

Definition 1 *Given a pair of hyper-alert types* (T_1, T_2)*, an* equality constraint for (T_1, T_2) *is a conjunction of equalities in the form of* $u_1 = v_1 \wedge \cdots \wedge u_n = v_n$*, where* $u_1, \cdots, u_n$ *are attribute names in* T_1 *and* $v_1, \cdots, v_n$ *are attribute names in* T_2*, such that there exist* $p(u_1, \cdots, u_n)$ *and* $p(v_1, \cdots, v_n)$*, which are the same predicate with possibly different arguments, in* $ExpConseq(T_1)$ *and* $Prereq(T_2)$*, respectively. Given a type* T_1 *hyper-alert* h_1 *and a type* T_2 *hyper-alert* h_2*,* h_1 *and* h_2 satisfy the equality constraint *if there exist* $t_1 \in h_1$ *and* $t_2 \in h_2$ *such that* $t_1.u_1 = t_2.v_1 \wedge \cdots \wedge t_1.u_n = t_2.v_n$ *evaluates to True.*

There may be several equality constraints for a pair of hyper-alert types. However, if a type T_1 hyper-alert h_1 prepares for a type T_2 hyper-alert h_2, then h_1 and h_2 must satisfy at least one of the equality constraints. Indeed, h_1 preparing for h_2 is equivalent to the conjunction of h_1 and h_2 satisfying at least one equivalent constraint and h_1 occurring before h_2. Assume that h_1 occurs before h_2. If h_1 and h_2 satisfy an equality constraint for (T_1, T_2), then by Definition 1, there must be a predicate $p(u_1, \cdots, u_n)$ in $ExpConseq(T_1)$ such that the same predicate with possibly different arguments, $p(v_1, \cdots, v_n)$, is in $Prereq(T_2)$. Since h_1 and h_2 satisfy the equality constraint, $p(u_1, \cdots, u_n)$ and $p(v_1, \cdots, v_n)$ will be instantiated to the same predicate in $ExpConseq(h_1)$ and $Prereq(h_2)$. This implies that h_1 prepares for h_2. Similarly, if h_1 prepares for h_2, there must be an instantiated predicate that appears in $ExpConseq(h_1)$ and $Prereq(h_2)$. This implies that there must be a predicate with possibly different arguments in $ExpConseq(T_1)$ and $Prereq(T_2)$ and that

this predicate leads to an equality constraint for (T_1, T_2) satisfied by h_1 and h_2.

Let us use an example from [83] to illustrate the notion of equality constraint. Consider the following hyper-alert types: *SadmindPing* = ({*VictimIP, VictimPort*}, *ExistsHost*(*VictimIP*), {*VulnerableSadmind* (*VictimIP*)}), and *SadmindBufferOverflow* = ({*VictimIP, VictimPort*}, *ExistHost* (*VictimIP*) $\wedge$ *VulnerableSadmind* (*VictimIP*), {*GainRootAccess* (*VictimIP*)}). The first hyper-alert type indicates that *SadmindPing* is a type of attacks that requires the existence of a host at the *VictimIP* to succeed, and as a result, the attacker may find out that this host has a vulnerable *Sadmind* service. The second hyper-alert type indicates that this type of attacks requires a vulnerable *Sadmind* service at the *VictimIP*, and as a result, the attack may gain root access. It is easy to see that there is a common predicate *VulnerableSadmind* in both *Prereq(SadmindBufferOverflow)* and *ExpConseq (SadmindPing)*. Thus, we have an equality constraint *VictimIP* $=$ *VictimIP* for (*SadmindPing*, *SadmindBufferOverflow*), where the first *VictimIP* comes from *SadmindPing*, and the second *VictimIP* comes from *SadmindBufferOverflow*.

We observe in many occasions that one step in a sequence of attacks may trigger multiple intrusion alerts, and the number of alerts may vary in different situations. This is partially due to the existing vulnerabilities and the hacking tools. For example, `unicode_shell` [87], which is a hacking tool against Microsoft IIS web server, checks about 20 vulnerabilities at the scanning stage and usually triggers the same number of alerts. As another example, in the attack scenario reported in [83], the attacker tried 3 different stack pointers and 2 commands in *Sadmind_Amslverify_Overflow* attacks for each victim host until one attempt succeeded. Even if not necessary, an attacker may still deliberately repeat the same step multiple times to confuse IDSs and/or system administrators. However, such variations do not change the corresponding attack strategy. Indeed, these variations make the attack scenarios unnecessarily complex, and may hinder manual or automatic analysis of the attack strategy. Thus, we decide to disallow such situations in our representation of attack strategies.

In the following, an attack strategy is formally represented as an attack strategy graph.

Definition 2 *Given a set $\mathcal{S}$ of hyper-alert types, an* attack strategy graph *over $\mathcal{S}$ is a quadruple (N, E, T, C), where (1) (N, E) is a connected DAG (directed acyclic graph); (2) T is a mapping that maps each $n \in N$ to a hyper-alert type in $\mathcal{S}$; (3) C is a mapping that maps each edge $(n_1, n_2) \in E$ to a set of equality constraints for $(T(n_1), T(n_2))$; (4) For any $n_1, n_2 \in N$, $T(n_1) = T(n_2)$*

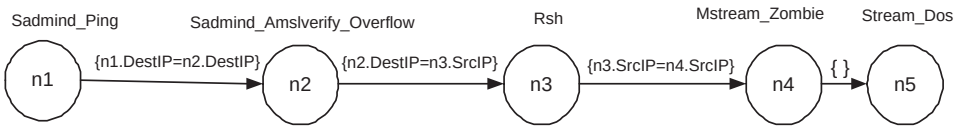


Figure 4.1: An example of attack strategy graph

implies that there exists $n_3 \in N$ *such that* $T(n_3) \neq T(n_1)$ *and* n_3 *is in a path between* n_1 *and* n_2*.*

In an attack strategy graph, each node represents a step in a sequence of related attacks. Each edge (n_1, n_2) represents that a type $T(n_1)$ attack is needed to prepare for a successful type $T(n_2)$ attack. Each edge may also be associated with a set of equality constraints satisfied by the intrusion alerts. These equality constraints indicate how one attack prepares for another. Finally, as represented by condition 4 in Definition 2, the same type of attacks should be considered as one step, unless they are in different stages of the attacks.

Note that attack strategies may also be specified manually in languages such as LAMBDA [30] and STATL [40]. However, manual specification of attack strategies requires prior knowledge of the strategies, and is also time-consuming and error-prone. Tools based on modeling checking techniques (*e.g.*, attack graphs [97, 58]) can certainly be used to build attack strategies from knowledge of individual types of attacks. However, these methods require clearly identified security properties to run the model checking tools, which may not always be available in reality. In contrast, our notion of attack strategy graph is intended to represent the strategies extracted from correlated intrusion alerts. Based on the knowledge about individual attack types, a program can automatically extract attack strategies from correlated intrusion alerts.

Now let us see an example of an attack strategy graph. Figure 4.1 is the attack strategy graph extracted from the hyper-alert correlation graph in Figure 2.1. The hyper-alert types are marked above the corresponding nodes, and the equality constraints are labeled near the corresponding edges. This attack strategy graph clearly shows the component attacks and the constraints that the component attacks must satisfy.

Learning Attack Strategies from Correlated Intrusion Alerts

As discussed earlier, our goal is to learn attack strategies automatically from correlated intrusion alerts. This requires that we extract the constraints intrinsic to attack strategy from alerts

so that the same constraints apply to all the other instances of the same strategy.

Our strategy to achieve this goal is to process the correlated intrusion alerts in two steps. First, we aggregate intrusion alerts that belong to the same step of a sequence of attacks into one hyper-alert. For example, in Figure 2.1, alerts 002 through 005 are indeed attempts of the same attack with different parameters, and thus they should be aggregated as one step in the attack sequence. Second, we extract the constraints between the attack steps and represent them as an attack strategy graph. For example, after we aggregate the hyper-alerts in the first step, we may extract the attack strategy graph shown in Figure 4.1.

The challenge lies in the first step. Because of the variations of attacks as well as the signatures that IDSs use to recognize attacks, there is no clear way to identify intrusion alerts that belong to the same step in a sequence of attacks. In the following, we first attempt to use the attack type information to do so. The notion of *aggregatable* hyper-alerts is introduced formally to clarify when the same type of hyper-alerts can be aggregated.

Definition 3 *Given a hyper-alert correlation graph $CG = (N, E)$, a subset $N' \subseteq N$ is* aggregatable*, if (1) all nodes in N' are the same type of hyper-alerts, and (2) $\forall n_1, n_2 \in N'$, if there is a path from n_1 to n_2, then all nodes in this path must be in N'.*

Intuitively, in a hyper-alert correlation graph, where intrusion alerts have been correlated together, the same type of hyper-alerts can be aggregated as long as they are not used in different stages in the attack sequence. Condition 1 in Definition 3 is quite straightforward, but condition 2 deserves more explanation. Consider the same type of hyper-alerts h_1 and h_2. If h_1 prepares for a different type of hyper-alert h' (directly or indirectly), and h' further prepares for h_2 (directly or indirectly), h_1 and h_2 obviously belong to different steps in the same sequence of attacks. Thus, we should not allow them to be aggregated together. Although we have never observed such situations, we cannot rule out such possibilities.

Based on the notion of aggregatable hyper-alerts, the first step in learning attack strategy from a hyper-alert correlation graph is quite straightforward. We only need to identify and merge all aggregatable hyper-alerts. To proceed to the second step in strategy learning, we need a hyper-alert correlation graph in which each hyper-alert represents a separate step in the attack sequence. Formally, we call such a hyper-alert correlation graph an irreducible hyper-alert correlation graph.

Definition 4 *A hyper-alert correlation graph $CG = (N, E)$ is* irreducible *if for all $N' \subseteq N$, where*

Algorithm 1. ExtractStrategy
Input: A hyper-alert correlation graph CG.
Output: An attack strategy graph ASG.
Method:
1. Let CG' = GraphReduction (CG).
2. Let $ASG = (N, E, T, C)$ be an empty attack strategy graph.
3. **for** each hyper-alert h in CG'
4. Add a new node, denoted n_h, into N and set $T(n_h)$ be the type of h.
5. **for** each edge (h, h') in CG'
6. Add $(n_h, n_{h'})$ into E.
7. **for** each $p_c \in ExpConseq(h)$ and $p_p \in Prereq(h')$
8. **if** $p_c = p_p$ **then**
9. Add into $C(n_h, n_{h'})$ the equality constraint $(u_1 = v_1) \wedge \cdots \wedge (u_n = v_n)$.
Note u_i and v_i are the ith variable of p_c and p_p before instantiation, respectively.
10. **return** $ASG(N, E, T, C)$.

Subroutine GraphReduction
Input: A hyper-alert correlation graph $CG = (N, E)$.
Output: An irreducible hyper-alert correlation graph $CG' = (N', E')$.
Method:
1. Partition the hyper-alerts in N into groups such that the same type of hyper-alerts are all in the same group.
2. **for** each group G
3. **if** there is a path $g, n_1, \cdots, n_k, g'$ in CG such that only g and g' are in G **then**
4. Divide G into G_1, G_2, and G_3 such that all hyper-alerts in G_1 occur before n_1, all hyper-alerts in G_3 occur after n_2, and all the other hyper-alerts are in G_2.
5. Repeat steps 2 to 4 until no group can be divided.
6. Aggregate the hyper-alerts in each group into one hyper-alert.
7. Let N' be the set of aggregated hyper-alerts.
8. **for** all $n_1, n_2 \in N'$
9. **if** there exists $(h_1, h_2) \in E$ and h_1 and h_2 are aggregated into n_1 and n_2, respectively
10. add (n_1, n_2) into E'.
11. **return** $CG' = (N', E')$.

Figure 4.2: An algorithm to extract attack strategy graph from a hyper-alert correlation graph

$|N'| > 1$, *N' is not aggregatable.*

Figure 4.2 shows the algorithm to extract attack strategy graphs from hyper-alert correlation graphs. The subroutine *GraphReduction* is used to generate an irreducible hyper-alert correlation graph, and the rest of the algorithm extracts the components of the output attack strategy graph. The steps in this algorithm are self-explanatory; we do not repeat them in the text. Lemma 4.1.1 ensures that the output of algorithm 1 indeed satisfies the constraints of an attack strategy graph.

Lemma 4.1.1 *The output of Algorithm 1 is an attack strategy graph.*

Proof: We first prove the output of the subroutine *GraphReduction* is an irreducible hyper-alert correlation graph by contradiction. Consider the output $CG' = (N', E')$ of *GraphReduction*. Suppose there exists $N_s \subseteq N'$, where $|N_s| > 1$, such that N_s is aggregatable. Thus, all nodes in N_s are the same type of hyper-alerts, and for any two different nodes $n_1, n_2 \in N_s$, if there is a path from n_1 to n_2, then all nodes in the path are in N_s. Since CG' is aggregated from the input hyper-alert correlation graph, for all pairs of nodes n'_1 and n'_2, where n'_1 and n'_2 are aggregated into n_1 and n_2, respectively, if there exists a path from n'_1 to n'_2 in the input graph, all the nodes in the path must be in the group of nodes aggregated into the nodes in N_s. According to steps 3 and 4 in *GraphReduction*, they should have been kept in the same group and aggregated into one node in CG'. This leads to a contradiction to the assumption that n'_1 and n'_2 are aggregated into n_1 and n_2, respectively.

Now we prove the output of Algorithm 1 is an attack strategy graph. Consider the output of Algorithm 1 $ASG = (N, E, T, C)$. It is easy to see that T is a mapping that maps each $n \in N$ to a hyper-alert type, and C is a mapping that maps each edge $e \in E$ to a set of equality constraints. In addition, because the input hyper-alert correlation graph is a DAG, (N, E) must be a directed graph. Suppose there is a cycle $n_1, n_2, \cdots, n_1$ in (N, E). There must exist two nodes n_{11}, n_{12}, and n_{21} in the input hyper-alert correlation graph such that n_{11} and n_{12} are aggregated into n_1, n_{21} is aggregated into n_2, and there exists a path $n_{11}, \cdots, n_{21}, \cdots, n_{12}$. However, according to the subroutine *GraphReduction*, n_{11} and n_{12} should have been put into two separate groups. Thus, (N, E) cannot have any cycle. Finally, for any $n_1, n_2 \in N$, since the output of *GraphReduction* is irreducible, if $T(n_1) = T(n_2)$, then there must exist $n_3 \in N$ in a path between n_1 and n_2 such that $T(n_3) \neq T(n_1)$.

4.1.2 Dealing with Variations of Attacks

Algorithm 1 in Figure 4.2 has ignored equivalent but different attacks in sequences of attacks. For example, an attacker may use either *pmap_dump* or *Sadmind_Ping* to find a vulnerable Sadmind service. As another example, an attacker may use either *SadmindBufferOverflow* or *TooltalkBufferOverflow* attack to gain remote access to a host. Obviously, at the same stage of two sequences of attacks, if an attacker uses equivalent but different attacks, Algorithm 1 will return two different attack strategy graphs, though the strategies behind them are the same.

We propose to generalize hyper-alert types so that the syntactic difference between equiv-

alent hyper-alert types is hidden. For example, we may generalize both *SadmindBufferOverflow* and *TooltalkBufferOverflow* attacks into *RPCBufferOverflow*.

A generalized hyper-alert type is created to hide the unnecessary difference between specific hyper-alert types. Thus, an occurrence of any of the specific hyper-alerts should imply an occurrence of the generalized one. This is to say that satisfaction of the prerequisite of a specific hyper-alert implies the satisfaction of the prerequisite of the generalized hyper-alert. Moreover, to cover all possible impact of all the specific hyper-alerts, the consequences of all the specific hyper-alert types should be included in the consequence of the generalized hyper-alert type. It is easy to see that this generalization may cause loss of information. Thus, generalization of hyper-alert types must be carefully handled so that information essential to attack strategy is not lost.

In the following, we formally clarify the relationship between specific and generalized hyper-alert types.

Definition 5 *Given two hyper-alert types T_g and T_s, where $T_g = (fact_g, prereq_g, conseq_g)$ and $T_s = (fact_s, prereq_s, conseq_s)$, we say T_g is* more general than T_s *(or, equivalently, T_s is* more specific than T_g*) if there exists an injective mapping f from $fact_g$ to $fact_s$ such that the following conditions are satisfied:*

- *If we replace all variables x in $prereq_g$ with $f(x)$, $prereq_s$ implies $prereq_g$, and*
- *If we replace all variables x in $conseq_g$ with $f(x)$, then all formulas in $conseq_s$ are implied by $conseq_g$.*

The mapping f is called the generalization mapping *from T_s to T_g.*

Let us look at an example. Suppose the hyper-alert types *SadmindBufferOverflow* and *TooltalkBufferOverflow* are specified as follows: *SadmindBufferOverflow* = ({*VictimIP, VictimPort*}, *ExistHost* (*VictimIP*) $\wedge$ *VulnerableSadmind* (*VictimIP*), {*GainRootAccess* (*VictimIP*)}), and *TooltalkBufferOverflow* = ({*VictimIP, VictimPort*}, *ExistHost* (*VictimIP*) $\wedge$ *VulnerableTooltalk* (*VictimIP*), {*GainRootAccess* (*VictimIP*)}). Assume that *VulnerableSadmind* (*VictimIP*) imply *VulnerableRPC* (*VictimIP*). Intuitively, this represents that if there is a vulnerable Sadmind service at *VictimIP*, then there must be a vulnerable RPC service (*i.e.*, the Sadmind service) at *VictimIP*. Similarly, we assume

VulnerableTooltalk (*VictimIP*) also implies *VulnerableRPC* (*VictimIP*). Then we can generalize both *SadmindBufferOverflow* and *TooltalkBufferOverflow* into *RPCBufferOverflow* = ($\{$*VictimIP*$\}$, *ExistHost* (*VictimIP*) $\wedge$ *VulnerableRPC* (*VictimIP*), $\{$*GainRootAccess* (*VictimIP*)$\}$), where the generalization mapping is $f(VictimIP) = VictimIP$.

By identifying a generalization mapping, we can specify how a specific hyper-alert can be generalized into a more general hyper-alert. Following the generalization mapping, we can find out what attribute values of a specific hyper-alert should be assigned to the attributes of the generalized hyper-alert. The attack strategy learning algorithm can be easily modified: We first generalize the hyper-alerts in the input hyper-alert correlation graph into generalized hyper-alerts following the generalization mapping, and then apply Algorithm 1 to extract the attack strategy graph.

Although a hyper-alert can be generalized in different granularities, it is not an arbitrary process. In particular, if one hyper-alert prepares for another hyper-alert before generalization, the generalized hyper-alerts should maintain the same relationship. Otherwise, the dependency between different attack stages, which is intrinsic in an attack strategy, will be lost.

The remaining challenge is how to get the "right" generalized hyper-alert types and generalization mappings. The simplest way is to manually specify them. For example, *Apache2, Back*, and *Crashiis* are all Denial of Service attacks. We may simply generalize all of them into one *WebServiceDOS*. However, there are often different ways to perform generalization. To continue the above example, *Apache2* and *Back* attacks are against the apache web servers, while *Crashiis* is against the Microsoft IIS web server. To keep more information about the attacks, we may want to generalize *Apache* and *Back* into *ApacheDOS*, while generalize *Crashiis* and possibly other DOS attacks against the IIS web server into *IISDOS*. Nevertheless, this does not affect the attack strategy graphs extracted from correlated intrusion alerts as long as the constraints on the related alerts are satisfied.

Automatic Generalization of Hyper-Alert Types. It is time-consuming and error-prone to manually generalize hyper-alert types. One way to partially automate this process is to use clustering techniques to identify the hyper-alert types that should be generalized into a common one. In our experiments, we use the bottom-up hierarchical clustering [55] to group hyper-alert types hierarchically on the basis of the similarity between them, which is derived from the similarity between the prerequisites and consequences of hyper-alert types. The method used to compute the similarity is described below.

To facilitate the computation of similarity between prerequisites of hyper-alert types, we convert each prerequisite into an *expanded prerequisite set*, which includes all the predicates that

appear or are implied by the prerequisite. Similarly, we can get the expanded consequence set. Consider two sets of predicates, denoted S_1 and S_2, respectively. We adopt the Jaccard similarity coefficient [54] to compute the similarity between S_1 and S_2, denoted $Sim(S_1, S_2)$. That is, $Sim(S_1, S_2) = \frac{a}{a+b+c}$, where a is the number of predicates in both S_1 and S_2, b is the number of predicates only in S_1, and c is the number of predicates only in S_2.

Consider hyper-alert types T_1 and T_2. The similarity between T_1 and T_2, denoted $Sim(T_1, T_2)$, is then computed as $Sim(T_1, T_2) = Sim(XP_1, XP_2) \times w_p + Sim(XC_1, XC_2) \times w_c$, where XP_1 and XP_2 are the expanded prerequisite sets of T_1 and T_2, XC_1 and XC_2 are the expanded consequence sets of T_1 and T_2, and w_p and $w_c = 1 - w_p$ are the weights for prerequisite and consequence, respectively. (In our experiments, we use $w_p = w_c = 0.5$ to give equal weight to both prerequisite and consequence of hyper-alert types.) We may then set a threshold t so that two hyper-alert types are grouped into the same cluster only if their similarity measure is greater than or equal to t.

4.2 Measuring the Similarity between Attack Strategies

In this section, we present techniques to measure the similarity between attack strategy graphs based on error tolerant graph/subgraph isomorphism, which has been studied extensively in pattern recognition [15, 75, 74, 76, 73]. Since the attack strategy graphs are extracted from sequences of correlated alerts, the similarity between two attack strategy graphs are indeed the similarity between the original alert sequences in terms of their strategies. Such similarity measurement is a fundamental problem in intrusion analysis; it has potential applications in incident handling, computer and network forensics, and other security management areas.

We are particularly interested in two problems. First, how similar are two attack strategies? Second, how likely is one attack strategy a part of another attack strategy? These two problems can be mapped naturally to error tolerant graph isomorphism and error tolerant subgraph isomorphism problems, respectively.

To facilitate the later discussion, we give a brief overview of error tolerant graph/subgraph isomorphism. Further details can be found in the rich literature on graph/subgraph isomorphism [15, 75, 74, 76, 73].

4.2.1 Error Tolerant Graph/Subgraph Isomorphism

In graph/subgraph isomorphism, a graph is a quadruple $G = (N, E, T, C)$, where N is the set of nodes, E is the set of edges, T is a mapping that assigns labels to the nodes, and C is a mapping that assigns labels to the edges. Given two graphs $G_1 = (N_1, E_1, T_1, C_1)$ and $G_2 = (N_2, E_2, T_2, C_2)$, a bijective function f is a *graph isomorphism* from G_1 to G_2 if

- for all $n_1 \in N_1, T_1(n_1) = T_2(f(n_1))$;
- for all $e_1 = (n_1, n_1') \in E_1$, there exists $e_2 = (f(n_1), f(n_1')) \in E_2$ such that $C(e_1) = C(e_2)$, and for all $e_2 = (n_2, n_2') \in E_2$, there exists $e_1 = (f^{-1}(n_2), f^{-1}(n_2')) \in E_1$ such that $C(e_2) = C(e_1)$.

Given a graph $G = (N, E, T, C)$, a *subgraph* of G is a graph $G_s = (N_s, E_s, T_s, C_s)$ such that (1) $N_s \subseteq N$, (2) $E_s = E \cap (N_s \times N_s)$, (3) for all $n_s \in N_s$, $T_s(n_s) = T(n_s)$, and (4) for all $e_s \in E_s$, $C_s(e_s) = C(e_s)$. Given two graphs $G_1 = (N_1, E_1, T_1, C_1)$ and $G_2 = (N_2, E_2, T_2, C_2)$, an injective function f is a *subgraph isomorphism* from G_1 to G_2, if there exists a subgraph G_{2s} of G_2 such that f is a graph isomorphism from G_1 to G_{2s}.

As a further step beyond graph/subgraph isomorphism, error tolerant graph/subgraph isomorphism (which is also known as error correcting graph/subgraph isomorphism) is introduced to cope with noises or distortion in the input graphs. There are two approaches for error tolerant graph/subgraph isomorphism: graph edit distance and maximal common graph. In this chapter, we focus on graph edit distance to study the application of error tolerant graph/subgraph isomorphism in intrusion detection.

The edit distance method assumes a set of edit operations (*e.g.*, deletion, insertion and substitution of nodes and edges) as well as the costs of these operations, and defines the similarity of two graphs in terms of the least cost sequence of edit operations that transforms one graph into the other. We denote the edited graph after a sequence of edit operations Δ as $\Delta(G)$. Consider two graphs G_1 and G_2. The *distance $D(G_1, G_2)$ from G_1 to G_2 w.r.t. graph isomorphism* is the *minimum* sum of edit costs associated with a sequence of edit operations Δ on G_1 that leads to a graph isomorphism from $\Delta(G_1)$ to G_2. Similarly, the *distance $D_s(G_1, G_2)$ from G_1 to G_2 w.r.t. subgraph isomorphism* is the *minimum* sum of edit costs associated with a sequence of edit operations Δ on G_1 that leads to a *subgraph* isomorphism from $\Delta(G_1)$ to G_2. An *error tolerant graph/subgraph isomorphism* from G_1 to G_2 is a pair (Δ, f), where Δ is a sequence of edit operations on G_1, and f is a graph/subgraph isomorphism from $\Delta(G_1)$ to G_2.

It is well known that subgraph isomorphism detection is an NP-complete problem [46]. Error tolerant subgraph isomorphism detection, which involves subgraph isomorphism detection, is also in NP and generally harder than exact subgraph isomorphism detection [74]. Nevertheless, error tolerant subgraph isomorphism has been widely applied in image processing and pattern recognition [15, 75, 74, 76, 73]. In our application, all the attack strategy graphs we have encountered are small graphs with less than 10 nodes. We argue that it is very unlikely to have very large attack strategy graphs in practice. Thus, we believe error tolerant graph/subgraph isomorphism can be applied to measure the similarity between attack strategy graphs with reasonable response time. Indeed, we did not observe any noticeable delay in our experiments.

4.2.2 Working with Attack Strategy Graphs

To successfully use error tolerant graph/subgraph isomorphism detection techniques, we need to answer at least the following three questions. What are the edit operations on an attack strategy graph? What are reasonable edit costs of these edit operations? What is the right similarity measurement between attack strategy graphs?

All the edit operations on a labeled graph are applicable to attack strategy graphs. Specifically, an *edit operation* on an attack strategy graph $ASG = (N, E, T, C)$ is one of the following:

1. Inserting a node n: $\$ \rightarrow n$. This represents adding a stage into an attack strategy. This edit operation is only needed for error-tolerant graph isomorphism.

2. Deleting a node n: $n \rightarrow \$$. This represents removing a stage from an attack strategy. Note that this implies deleting all edges adjacent with n.

3. Substituting the hyper-alert type of a node n: $T(n) \rightarrow t$, where t is a hyper-alert type. This represents changing the attack at one stage of the attack strategy.

4. Inserting an edge $e = (n_1, n_2)$: $\$ \rightarrow e$, where $n_1, n_2 \in N$. This represents adding dependency (*i.e.*, prepare-for relation) between two attack stages.

5. Deleting an edge $e = (n_1, n_2)$: $e \rightarrow \$$. This represents removing dependency (*i.e.*, prepare-for relation) between two attack stages.

6. Substituting the label of an edge $e = (n_1, n_2)$: $C(e) \rightarrow c$, where c is a set of equality constraints. This represents changing the way in which two attack stages are related to each other. (Note that c is not necessarily a set of equality constraints for $(T(n_1), T(n_2))$.)

These edit operations do not necessarily transform one attack strategy graph into another attack strategy graph. Indeed, a labeled graph must satisfy some constraints to be an attack strategy graph. For example, all the equality constraints in the label associated with (n_1, n_2) must be valid equality constraints for $(T(n_1), T(n_2))$. It is easy to see that the edit operations may violate some of these constraints.

One may suggest these constraints be enforced throughout the transformation of attack strategy graphs. As an additional benefit, this can be used to reduce the search space required for graph/subgraph isomorphism. However, this approach may not find the least expensive sequence of edit operations, and may even fail to find a transformation from one attack strategy graph to (the subgraph of) another. Indeed, editing distance is one way to measure the difference between attack strategy graphs; it is not necessary to require that all the intermediate edited graphs are attack strategy graph. As long as the final edited graph is isomorphic to an attack strategy graph, it is guaranteed to be an attack strategy graph. Thus, we do not require the intermediate graphs during graph transformation be attack strategy graphs.

Assignment of edit costs to edit operations is a critical step in error tolerant graph/subgraph isomorphism. The actual costs are highly dependent on the domain in which these techniques are applied. In our application, there are multiple reasonable ways to assign the edit costs. In the following, we attempt to give some constraints that the cost assignment must satisfy.

In an attack strategy graph, a node represents a stage in an attack strategy, while an edge represents the causal relationship between two steps in the strategy. Obviously, changing the stages in an attack strategy affects the attack strategy significantly more than modifying the causal relationships between stages. Thus, the edit costs of node related operations should be significantly more expensive than those of the edge related operations.

Inserting or deleting a node implies having one more or fewer step in the strategy, while substituting a node type implies to replace the attack in one step in the strategy. Thus, inserting or deleting a node has at least the same impact on the strategy as substituting the node type. Moreover, deleting a node and inserting a node are both manipulations of a stage; there is no reason to say one operation has more impact than the other. Therefore, they should have the same cost. Both inserting and deleting an edge changes the causal relationship between two attack stages, and they should have the same impact on the attack strategy. However, substituting the label of an edge is just to change the way in which two attack stages are related. Thus, it should have less cost than edge insertion and deletion. In summary, we can derive the following constraint in edit cost assignments.

Constraint 4.2.1 $Cost_{n\to\$} = Cost_{\$\to n} \geq Cost_{T(n)\to t} >> Cost_{\$\to e} = Cost_{e\to\$} \geq Cost_{C(e)\to c}$.

The labels in an attack strategy graph is indeed a set of equality constraints. As a result, labels are not entirely independent of each other. This further implies that edit costs for edge label substitution should not be uniformly assigned. For example, substituting an edge label $\{A, B\}$ for $\{A, C\}$ should have less cost than substituting $\{A, B\}$ for $\{C, D\}$. This observation leads to another constraint.

Constraint 4.2.2 *Assume that the edit operation $C(e) \to c$ replaces $C(e) = c_{old}$ with c_{new}. The edit cost $Cost_{C(e)\to c}$ should be smaller when c_{old} and c_{new} have more equality constraints in common.*

Here we give a simple way to accommodate Constraint 4.2.2. We assume there is a maximum edit cost for label substitution operation, denoted as $MaxCost_{C(e)\to c}$. The edit cost of a label substitution is then $Cost_{C(e)\to c} = MaxCost_{C(e)\to c} \times \frac{|c_{old} \cap c_{new}|}{|c_{old} \cup c_{new}|}$, where c_{old} and c_{new} are the labels (*i.e.*, sets of equality constraints) before and after the operation.

Error tolerant graph/subgraph isomorphism detection techniques can conveniently give a distance between two labeled graphs, which is measured in terms of edit cost. As we discussed earlier, we use these techniques to help answer two questions: (1) How similar are two sequences of attacks in terms of their attack strategy? (2) How likely does one sequence of attacks use a part of attack strategy in another sequence of attacks? In the following, we transform the edit distance measures into more direct similarity measures.

Consider an attack strategy graph ASG. We refer to the distance from ASG to an empty graph as the *reductive weight of ASG*, denoted as $W_r(ASG)$. Similarly, we refer to the distance from an empty graph to ASG as the *constructive weight of ASG*, denoted $W_c(ASG)$.

Definition 6 *Consider two attack strategy graphs ASG_1 and ASG_2. The* similarity between ASG_1 and ASG_2 w.r.t. (attack) strategy *is* $Sim(ASG_1, ASG_2) = \frac{Sim(ASG_1\to ASG_2)+Sim(ASG_2\to ASG_1)}{2}$, *where* $Sim(ASG_x \to ASG_y) = 1 - \frac{D(ASG_x, ASG_y)}{W_r(ASG_x)+W_c(ASG_y)}$.

Definition 7 *Consider two attack strategy graphs ASG_1 and ASG_2. The* similarity between ASG_1 and ASG_2 w.r.t. (attack) sub-strategy *is* $Sim_{Sub}(ASG_1, ASG_2) = 1 - \frac{D_s(ASG_1, ASG_2)}{W_r(ASG_1)+W_c(ASG_2)}$.

Simple analysis of the impact of edit costs on the similarity measurement. Suppose we have two graph G_a and G_b, which have n_a and n_b nodes, and e_a and e_b edges, respectively. Assume we perform an error tolerant graph isomorphism from G_a to G_b, the node operations have the same cost C_N, and edge operations have the same cost C_E, where $C_N \gg C_E$. In the sequence of edit operations, suppose there are N_N node operations, and N_E edge operations. Then the similarity measure can be simplified as follows.

$$Sim(G_a, G_b) = 1 - \frac{D(G_a, G_b)}{W_r(G_a) + W_c(G_b)} = 1 - \frac{C_N \times N_N + C_E \times N_E}{C_N \times (n_a + n_b) + C_E \times (e_a + e_b)}$$

Further let $e_a + e_b = k \times (n_a + n_b)$, and $N_E = s \times N_N$. Then we have

$$Sim(G_a, G_b) = 1 - \frac{C_N \times N_N + C_E \times s \times N_N}{C_N \times (n_a + n_b) + C_E \times k \times (n_a + n_b)} = 1 - \frac{N_N \times (C_N + C_E \times s)}{(n_a + n_b) \times (C_N + C_E \times k)}$$

When k and s are not large, since $C_N \gg C_E$, the formula can be further simplified.

$$Sim(G_a, G_b) = 1 - \frac{N_N}{n_a + n_b}$$

Thus, under the above assumptions, the similarity is approximately determined by the proportion of the number of edited nodes to the total number of nodes. In summary, when the number of edges are not substantially more than the number of nodes, and the number of edge operations are not substantially more than the number of node operations, the similarity measure is mainly determined by the number of nodes and node operations rather than the edit costs.

4.3 Experiments

We have performed a series of experiments to study the proposed techniques. In our experiments, for alert correlation method, we used the implementation of [83]. We used GraphViz [9] to visualize graphs. In addition, we used *GUB* [73], A Toolkit for Graph Matching, to perform error tolerant graph/subgraph isomorphism detection and compute distances between attack strategy graphs. We used RealSecure Network Sensor [52] and Snort [94] as our IDS sensors (Accordingly, we labeled the alerts generated by different sensors with *RealSecure* or *Snort*, respectively).

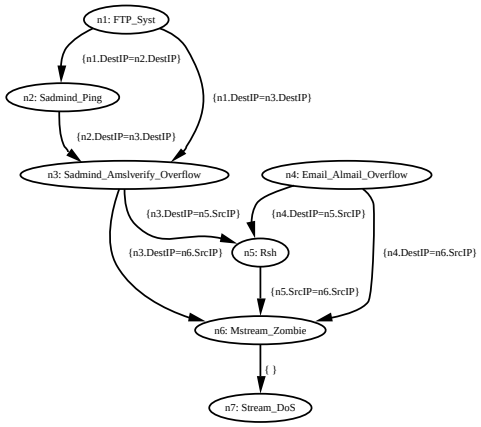
Our test data sets include the 2000 DARPA intrusion detection scenario specific data sets [77]. The data sets contain two scenarios: LLDOS 1.0 and LLDOS 2.0.2. In LLDOS1.0, the sequence of attacks includes IPsweep, probes of sadmind services, breakins through sadmind exploits, installations of DDoS programs, and finally the DDoS attack. LLDOS 2.0.2 is similar to LLDOS 1.0; however, the attacks in LLDOS 2.0.2 are more stealthy than those in LLDOS 1.0. In addition to the DARPA data sets, we also performed three sequences of attacks in an isolated network. In all these three attack sequences, the attacker started with nmap [44] scans of the victim. Then, in the first sequence, the attacker sent malformed urls [24] to the victim's Internet Information Services (IIS) to get a cmd.exe shell. In the second sequence, the attacker took advantage of the flaws of IP fragment reassembly on Windows 2000 [23] to launch a DoS attack. In the third sequence, the attacker launched a buffer overflow attack against the Internet Printing Protocol accessed via IIS 5.0 [25, 18].

4.3.1 Learning Attack Strategies from Correlated Intrusion Alerts

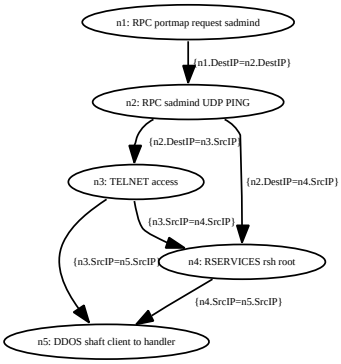
Our first goal is to evaluate the effectiveness of our approach on extracting the attack strategies. Figure 4.3 shows all of the attack strategy graphs extracted from the test data sets. The label of each node is the node ID followed by the hyper-alert type of the node. The label of each edge describes the set of equality constraints for the hyper-alert types associated with the two end nodes.

The attack strategy graphs we extracted from LLDOS 1.0 (inside part) are shown in Figure 4.3(a) and 4.3(b). Comparing them with the description of the data set [77], we know that both Figures 4.3(a) and 4.3(b) have captured most of the attack strategy. The missing parts are due to the attacks missed by the IDSs. Since we did not generalize variations of hyper-alert types, these graphs still have syntactic differences despite of their common strategy. (Note that the "RPC sadmind UDP PING" alert reported by Snort is indeed the "Sadmind_Amslverify_Overflow" alert by RealSecure, and the "RPC portmap sadmind request UDP" alert by Snort is the "Sadmind_Ping" alert by RealSecure.) Moreover, false alerts are also reflected in the attack strategy graphs. For example, the hyper-alert types "Email_Almail_Overflow" and "FTP_Syst" in Figure 4.3(a) do not belong to the attack strategy, but they are included because of the false detection.

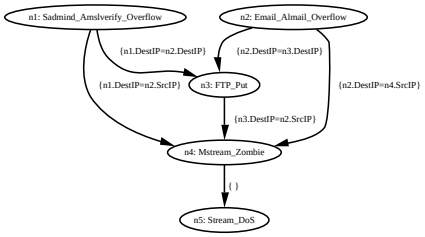
The attack strategies extracted from LLDOS 2.0.2 are shown in Figures 4.3(c) and 4.3(d). Compared with the five phases of attack scenarios [77], it is easy to see that Figure 4.3(c) reveals most of the adversary's strategy. However, Figure 4.3(d) reveals two steps fewer than Figure 4.3(c).



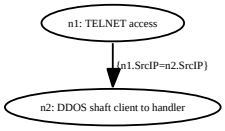
(a) LLDOS1.0 inside dataset (RealSecure)



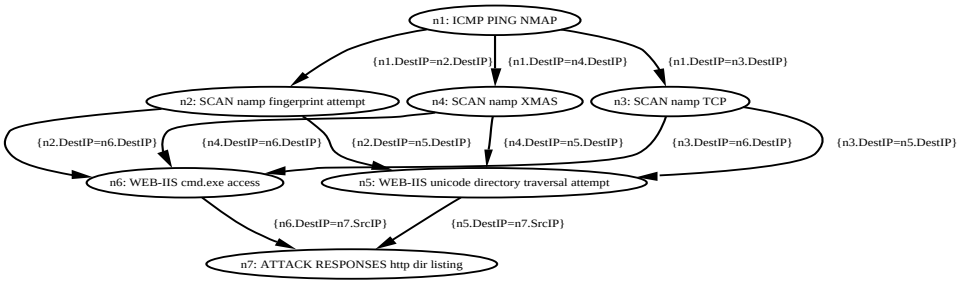
(b) LLDOS1.0 inside dataset (Snort)



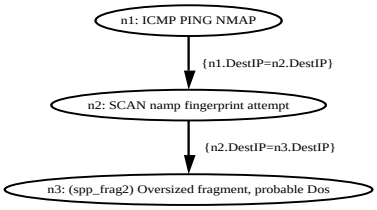
(c) LLDOS2.0.2 inside dataset (RealSecure)



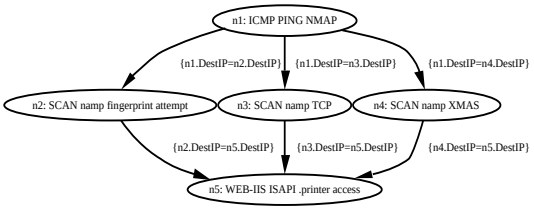
(d) LLDOS2.0.2 inside dataset (Snort)



(e) WEB-IIS unicode exploits (Snort)



(f) jolt2 DoS attack (Snort)



(g) WEB-IIS ISAPI .printer access (Snort)

Figure 4.3: Attack Strategy Graphs Extracted from Our Experiments

Our further investigation indicates that this is because one critical attack step, the buffer overflow attacks against sadmind service, was completely missed by Snort. Figures 4.3(e), 4.3(f), and 4.3(g) show the attack strategies extracted from the three sequences of attacks we performed. By comparing with the attacks, we can see that the stages as well as the constraints intrinsic to these attack strategies are mostly captured by these graphs.

Though showing some potential, these experimental results also reveal a limitation of the attack strategy learning method. That is, our method depends on the underlying IDSs as well as the alert correlation method. If the hyper-alert correlation graphs do not reveal the entire attack strategy, or include false alerts, the attack strategy graphs generated by our method will not be perfect. Nevertheless, our technique is intended to automate the analysis process typically performed by human analysts, who may make the same mistake if no other information is used. More research is clearly needed to mitigate the impact of imperfect IDS and correlation.

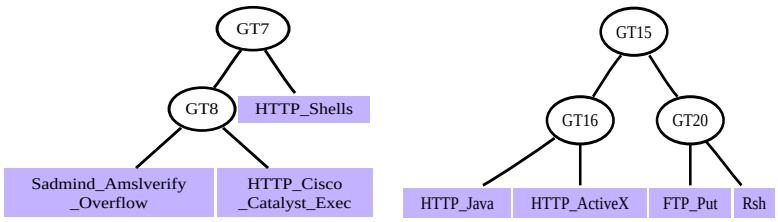
Another observation is that alerts from heterogeneous IDSs can help complete the attack strategies. For example, combining Figures 4.3(c) and 4.3(d), we know that an attacker may launch buffer overflow attacks against sadmind service and then use telnet to access the victim machine.

Note that we do not give a quantitative performance evaluation of attack strategy extraction (*i.e.*, the false positive and false negative of the extracted attack strategies). This is because such measures are indeed determined by the underlying intrusion alert correlation algorithm. As long as correlation is performed correctly, our method can always extract the strategy reflected by the correlated alerts.

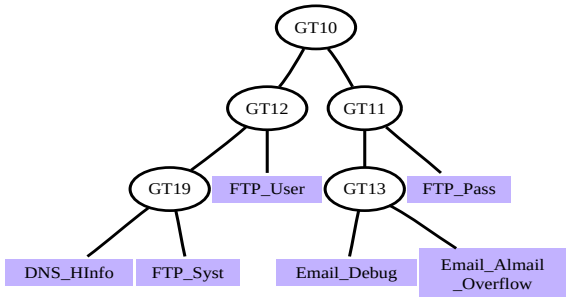
4.3.2 Measuring the Similarity between Alert Sequences

We performed experiments to measure the similarity between the previously extracted seven attack strategy graphs. To hide the unnecessary differences between alert types, we performed generalized to alert types.

The generalization of hyper-alert types. We first performed automatic generalization of hyper-alert types. Figure 4.4 shows the results we obtained for the hyper-alert types in the 2000 DARPA data sets. Here the string inside the non-leaf node means *Generalization Type* followed by an ID. From Figure 4.4(b), we know that *FTP_Put* and *Rsh* can be generalized to the same type. These results were used in our later experiments when we computed the similarity measures between attack strategy graphs. Besides Figure 4.4, additional generalization hierarchies of hyper-alert types in our experiments are shown in Figure 4.5.

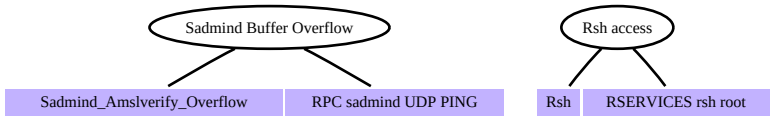


(a) Hierarchy 1 (b) Hierarchy 2

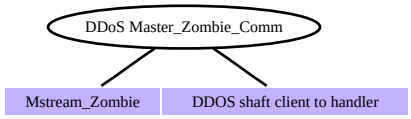


(c) Hierarchy 3

Figure 4.4: Generalization hierarchies for hyper-alert types in DARPA 2000 datasets. Threshold $t = 0.5$.



(a) Hierarchy 4 (b) Hierarchy 5



(c) Hierarchy 6

Figure 4.5: Additional generalization hierarchies of hyper-alert types in our experiments

Similarity Measurement. We assume the edit costs for node operations are all 10, and the edit costs for the edge operations are all 1. Tables 4.1 and 4.2 show the similarity measurements between each pair of attack strategy graphs w.r.t. attack strategy and attack sub-strategy, respec-

Table 4.1: The similarity w.r.t. attack strategy between attack strategy graphs in Figure 4.3

	$G_{4.3(a)}$	$G_{4.3(b)}$	$G_{4.3(c)}$	$G_{4.3(d)}$	$G_{4.3(e)}$	$G_{4.3(f)}$	$G_{4.3(g)}$
$G_{4.3(a)}$	/	0.72	0.73	0.21	0.29	0.31	0.25
$G_{4.3(b)}$	0.72	/	0.66	0.55	0.25	0.25	0.29
$G_{4.3(c)}$	0.73	0.66	/	0.40	0.34	0.38	0.30
$G_{4.3(d)}$	0.21	0.55	0.40	/	0.21	0.40	0.38
$G_{4.3(e)}$	0.29	0.25	0.34	0.21	/	0.48	0.74
$G_{4.3(f)}$	0.31	0.25	0.38	0.40	0.48	/	0.61
$G_{4.3(g)}$	0.25	0.29	0.30	0.38	0.74	0.61	/

Table 4.2: The similarity w.r.t. attack sub-strategy between attack strategy graphs in Figure 4.3

	$G_{4.3(a)}$	$G_{4.3(b)}$	$G_{4.3(c)}$	$G_{4.3(d)}$	$G_{4.3(e)}$	$G_{4.3(f)}$	$G_{4.3(g)}$
$G_{4.3(a)}$	/	0.72	0.66	0.31	0.53	0.31	0.43
$G_{4.3(b)}$	0.89	/	0.67	0.55	0.61	0.38	0.51
$G_{4.3(c)}$	0.90	0.68	/	0.40	0.61	0.38	0.52
$G_{4.3(d)}$	0.89	1.00	0.86	/	0.79	0.60	0.73
$G_{4.3(e)}$	0.51	0.58	0.58	0.21	/	0.48	0.26
$G_{4.3(f)}$	0.72	0.65	0.65	0.40	0.91	/	0.89
$G_{4.3(g)}$	0.59	0.51	0.48	0.27	0.93	0.61	/

tively. Each subscript in the tables denotes the graph it represents. We notice that $Sim_{Sub}(G_i, G_j)$ may not necessarily be equal to $Sim_{Sub}(G_j, G_i)$.

Table 4.1 indicates that Figure 4.3(a) is more similar to Figures 4.3(b) and 4.3(c). In addition, Figure 4.3(g) is more similar to Figures 4.3(e) and 4.3(f) than the other graphs. Based on the description of these attack sequences, we can see these similarity measures conform to human perceptions.

Table 4.2 shows the similarity between attack strategy graphs w.r.t. attack sub-strategy. We can see that Figures 4.3(b), 4.3(c), and 4.3(d) are very similar to a sub-strategy of Figure 4.3(a). In addition, Figure 4.3(d) is exactly a sub-strategies of Figure 4.3(b). Similarly, Figures 4.3(g) and 4.3(f) are both similar to sub-strategies of Figure 4.3(e), and Figure 4.3(f) is also similar to a sub-strategy of Figure 4.3(g). Comparing these measure values with these attack sequences, we can see these measures also conform to human perceptions.

The experiments also reveal some remaining problems that have not been addressed by our techniques. First, the similarity measures make sense in terms of their relative values. However, we still do not understand what a specific similarity measure represents. Second, false alerts generated by IDSs have a negative impact on the measurement. It certainly requires further research to address

these issues.

4.3.3 Identification of Missing Detections

Our last set of experiments is intended to study the possibility to apply the similarity measurement method to identify attacks missed by IDSs. For the sake of presentation, we first introduce two terms: precedent set and successive set. Intuitively, the *precedent set* of a node n in an attack strategy graph is the set of nodes from which there are paths to n, while the *successive set* of n is the set of nodes to which n has a path. In the following, we show two examples we encountered in our experiments.

Example 1 *The attack strategy graph in Figure 4.3(c) has no network probe phase, but Figure 4.3(a) does. The similarity measurement* $Sim_{Sub}(G_{4.3(c)}, G_{4.3(a)}) = 0.90$ *and* $Sim(G_{4.3(c)}, G_{4.3(a)}) = 0.73$ *indicate that these two strategies are very similar and it's very likely that Figure 4.3(c) is a sub-strategy of Figure 4.3(a). Thus, it is possible that some probe attacks are missed by the IDS when the IDS detected the attacks corresponding to Figure 4.3(c). Indeed, this is exactly what happened in LLDOS 2.0.2. The adversary used some stealthy attacks (*i.e., *HINFO query to the DNS server) to get the information about the victim host.*

Example 2 *Consider Figures 4.3(d) and 4.3(b). We have* $Sim_{Sub}(G_{4.3(d)}, G_{4.3(b)}) = 1.0$. *Thus,* $G_{4.3(d)}$ *is exactly a sub-strategy of* $G_{4.3(b)}$. *By checking the LLDOS2.0.2 alerts reported by Snort, we know that there are also "RPC portmap sadmind request UDP" alerts as in Figure 4.3(b). However, since Snort did not detect the later buffer overflow attack, these "RPC portmap sadmind request UDP" alerts are not correlated with the later alerts.*

We then perform the following steps, trying to identify attacks possibly missed in LLDOS 2.0.2. We pick node $n1$ *in Figure 4.3(d), and find its corresponding node* $n3$ *in Figure 4.3(b), which is mapped to* $n1$ *by the subgraph isomorphism. It is easy to see that in Figure 4.3(b), the*

precedent set of $n3$ is $\{n1, n2\}$, and $n1$ has the type "RPC portmap sadmind request UDP". We then go back to LLDOS 2.0.2 alerts, and find "RPC portmap sadmind request UDP" alerts before "TELNET ACCESS". By comparing the precedent set of $n1$ in Figure 4.3(d) and the precedent set of $n3$ in Figure 4.3(b), we suspect that "RPC sadmind UDP PING" (which corresponds to node $n2$ in Figure 4.3(b)) has been missed in LLDOS 2.0.2. If we add such an alert, we may correlate it with "RPC portmap sadmind request UDP" and further with "TELNET access" in Figure 4.3(d). Indeed, "RPC sadmind UDP PING" is the buffer overflow attack missed by Snort in LLDOS 2.0.2.

The later part of Example 2 is very similar to the abductive correlation proposed in [29]. The additional feature provided by the similarity measurement is the guidelines about what attacks may be missed. In this sense, the similarity measurement is complementary to the abductive correlation. Moreover, these examples are provided to demonstrate the potential of identifying missed attacks through measuring similarity of attack sequences. It is also possible that the attacker did not launch those attacks. Additional research is necessary to improve the performance and reduce false identification rate.

4.4 Summary

In this chapter, we develop techniques to extract attack strategies from correlated intrusion alerts based on the alert correlation methods [83, 29]. We propose a model to represent and algorithms to extract attack strategies from intrusion alerts. Moreover, to accommodate variations in attacks that are not intrinsic to attack strategies, we propose to generalize different types of intrusion alerts to hide the unnecessary difference between them. Finally, we develop techniques to measure the similarity between sequences of attacks based on their strategies. Our experimental results have shown that our techniques can successfully extract invariant attack strategies from sequences of alerts, measure the similarity between alert sequences in a way conforming to human intuition, and has a potential to identify attacks missed by IDSs.

Notice that our techniques on attack strategy extraction depend on the underlying alert correlation approaches. If alert correlation methods such as [83] discover attack scenarios from alert data sets, then our approach can extract attack strategy graphs from them. In the worst case, if

no attack scenarios can be identified, our techniques cannot work well. Fortunately, our preliminary experimental results demonstrate the potential of our techniques.

Chapter 5

Hypothesizing and Reasoning about Attacks Missed by Intrusion Detection Systems

With the development of the Internet, more and more organizations manage their data in networked information systems. Due to the open nature of the Internet, network intrusions have become an increasingly serious problem in recent years. Intrusion detection, which is aimed at detecting activities violating the security policies of the networked information systems, has been considered a necessary component to protect these systems along with other prevention-based security mechanisms such as access control.

As we mentioned in Chapter 2, intrusion detection techniques are generally classified into two categories: *anomaly detection* and *misuse detection*. Anomaly detection builds profiles (e.g., statistical models) for normal activities, and raises alerts when the monitored behaviors significantly deviate from the normal operations. Misuse detection constructs signatures (patterns) based on known attacks or vulnerabilities, and reports alerts if the monitored activities match the signatures.

Despite over 20 years' efforts on intrusion detection, current intrusion detection systems (IDSs) still have several well-known problems. First, existing IDSs cannot detect all intrusions.

While a misuse detection system cannot detect an unknown attack (or an unknown variation of a known attack), an anomaly detection system may fail to recognize stealthy malicious activities, too. Second, current IDSs cannot ensure that all alerts reflect actual attacks; *true positives* (attacks detected as intrusive) are usually mixed with *false positives* (benign activities detected as intrusive). Third, an IDS usually produces a large number of alerts [11, 59, 60, 61]. As indicated in [59], five IDS sensors reported 40MB of alert data within ten days, and a large fraction of these alerts are false positives. The high volumes and low quality (i.e., missed attacks and false positives) of the intrusion alerts make it very challenging for human users or intrusion response systems to understand the alerts and take appropriate actions. Thus, it is necessary to develop techniques to deal with the large volumes and low quality of intrusion alerts.

Besides the aforementioned problems, current IDSs are not sufficiently prepared for several trends in attacks. According to a 2002 CERT report [20], there are increasingly more automated attack tools, which typically consist of several (evolving) phases such as scanning for potential victims, compromising vulnerable systems, propagating the attacks, and coordinated management of attack tools. Moreover, attack tools are increasingly more sophisticated. In particular, "today's automated attack tools can vary their patterns and behaviors based on random selection, predefined decision paths, or through direct intruder management" [20]. These attack trends require more capable systems than the current IDSs to handle large volumes of alerts that potentially belong to different complex attack scenarios.

As we discussed in Chapter 2, several alert correlation techniques have been proposed in recent years to facilitate the analysis of intrusion alerts. These methods attempt to correlate IDS alerts based on the similarity between alert attributes [98, 109, 33, 28], previously known (or partially known) attack scenarios [36, 34], or prerequisites and consequences of known attacks [29, 83, 82]. A common requirement of these approaches is that they all heavily depend on the underlying IDSs for alerts. As a result, the performance of alert correlation is strictly limited by the performance of IDSs. In particular, if the IDSs miss critical attacks, the correlated alerts cannot reflect the actual attack scenarios due to the lack of the corresponding alerts, and thus may provide misleading information.

In this chapter, we develop a series of techniques to hypothesize and reason about attacks possibly missed by IDSs, aiming at constructing high-level attack scenarios even if the underlying IDSs miss critical attacks. Our approach is to integrate the potentially relevant attack scenarios generated by the alert correlation technique in [83], and use the intrinsic relationships between related attacks to hypothesize and reason about attacks missed by the IDSs. We observe that if two attacks

are causally related, they usually satisfy certain constraints (e.g., sharing the same destination IP address), even if they are not directly adjacent to each other in a sequence of attacks. If the IDSs miss some critical attacks, alerts from the same attack scenario could be split into multiple attack scenarios. Thus, combining different attack scenarios and verifying the above constraints over possibly related alerts can potentially overcome the problem introduced by missed attacks.

Our approach works as follows. We first obtain (multiple) attack scenarios through a correlation method based on prerequisites and consequences of attacks such as those in [29, 83], and identify what attack scenarios (and possibly individual, uncorrelated alerts) may be combined by examining the attributes of the alerts in different attack scenarios. If those attribute values satisfy the aforementioned constraints, we consider integrating the corresponding attack scenarios. We assume the missed attacks are most likely unknown variations of known attacks, or attacks equivalent to some known attacks. We then hypothesize and reason about attacks missed by IDSs based on possible causal relationships between known attacks, aiming at constructing more complete attack scenarios. The hypothesized attacks can be further validated through raw audit data. For example, we might hypothesize that variations of *IMAP_Authen_Overflow* and/or *RPC_Cmsd_Overflow* were missed by the IDSs. However, if during the target time frame, there is only IMAP traffic but no RPC traffic related to the target host, we can conclude that the latter hypothesis is incorrect. Finally, to improve the usability of the constructed attack scenarios, we consolidate the hypothesized attacks and generate concise representations of the combined attack scenarios.

Our main contribution in this chapter is a series of techniques to combine multiple attack scenarios and to hypothesize and reason about attacks possibly missed by the IDSs. These techniques are critical to constructing high-level attack scenarios from low-level intrusion alerts in situations where the IDSs cannot detect all attacks. These techniques complement the underlying IDSs by hypothesizing and reasoning about missed attacks, and thus provide valuable additional evidence to support further intrusion investigation and response.

5.1 Hypothesizing and Reasoning about Attacks Missed by IDSs

If IDSs miss some critical attacks, an attack scenario (represented as a correlation graph) may be split into multiple smaller ones, each of which only reflects a part of the original attack scenario. To better understand the whole attack scenario, it is desirable to integrate related attack scenarios, and hypothesize and reason about the attacks possibly missed by the IDSs. In this section,

(a) CG_1 (b) CG_2

Figure 5.1: Two correlation graphs

we develop a sequence of techniques for these purposes. We assume we have applied the causal correlation method to the alerts before using the newly proposed techniques.

In the following, we start with a straightforward approach to integrating possibly related attack scenarios, and gradually develop more sophisticated techniques to enhance this approach.

5.1.1 Integrating Possibly Related Correlation Graphs

We observe that the causal correlation method can be potentially enhanced by a similarity-based correlation method (e.g., [109, 98, 60, 61]), which clusters alerts based on the similarity between alert attribute values. Intuitively, when the IDSs miss certain attacks, though the causal correlation method may split the alerts from the same attack scenario into several correlation graphs, a similarity-based correlation method still has the potential to identify the common features shared by these alerts, and thus help re-integrate the related correlation graphs together. To take advantage of this observation, we integrate correlation graphs based on the alert clusters generated by a similarity-based correlation method.

The integration process may be conceptually divided into two steps: (1) identify the correlation graphs to be integrated, and (2) determine possible causal relationships between alerts in different correlation graphs. In this chapter, we choose a simple technique for the first step: we integrate two correlation graphs when they both contain alerts from a common cluster generated by a similarity-based correlation method. For example, given the two correlation graphs shown in Figures 5.1(a) and 5.1(b)[1], if the clustering method groups `SCAN_NMAP_TCP2` and `Rsh3` in the same cluster based on their common source and destination IP addresses, we consider integrating these two graphs together.

The first step is pretty straightforward once we select a similarity-based correlation method.

[1]The string inside each node is a hyper-alert type name followed by an alert ID.

However, the second step remains challenging, since we must deal with missed attacks that cause an attack scenario to split into multiple correlation graphs. Thus, we focus on the second step in the following discussion. As we will see later, the first step becomes unnecessary as we develop our approach. Without loss of generality, we assume that we integrate two correlation graphs.

We propose to harness the prior knowledge of attacks and the alert timestamp information to hypothesize about possible causal relationships between alerts in different correlation graphs. For example, suppose an attacker uses *nmap* [44] to find out a vulnerable service, then uses a buffer overflow attack to compromise that service, and finally installs and starts a DDoS daemon program. When we observe an earlier *SCAN_NMAP_TCP* and a later *Mstream_Zombie* alert in two correlation graphs that are identified for integration, we may hypothesize that the *SCAN_NMAP_TCP* alert indirectly prepares for the *Mstream_Zombie* alert through an unknown attack (or an unknown variation of the above buffer overflow attack). As a result, we would hypothesize an indirect causal relationship between these two alerts.

To further characterize this intuition and facilitate later discussion, we introduce two definitions. (Note that Definition 8 is based on the model in [83].) For convenience, we denote the type of an alert t (or an hyper-alert h) as $Type(t)$ (or $Type(h)$).

Definition 8 *Given two hyper-alert types T and T', we say T may prepare for T' if* ExpConseq(T) *and* Prereq(T') *share at least one predicate (with possibly different arguments).*

Example 3 *Consider two hyper-alert types* SadmindPing $=$ *(*fact, prereq, conseq*) and* Sadmind-BufferOverflow $=$ *(*fact', prereq', conseq'*), where* fact $=$ {VictimIP, VictimPort}, prereq $=$ ExistHost(VictimIP), conseq $=$ {VulnerableSadmind (VictimIP)}, fact' $=$ {VictimIP, VictimPort}, prereq' $=$ ExistHost(VictimIP) $\wedge$ VulnerableSadmind(VictimIP), *and* conseq' $=$ {GainRootAccess (VictimIP)}. *We observe both* ExpConseq*(*SadmindPing*) and* Prereq*(*SadmindBufferOverflow*) include the predicate* VulnerableSadmind(VictimIP). *Then we know that* SadmindPing *may prepare for* SadmindBufferOverflow.

Definition 9 *Given a set $\mathcal{T}$ of hyper-alert types, we say T may indirectly prepare for T' w.r.t. $\mathcal{T}$ if there exists a sequence of hyper-alert types $T, T_1, ..., T_k, T'$ such that (1) all these hyper-alert types*

are in $\mathcal{T}$, and (2) T may prepare for T_1, T_i may prepare for T_{i+1}, where $i = 1, 2, ..., k-1$, and T_k may prepare for T'.

Let us look at an example. Suppose we have a set $\mathcal{T}$ of hyper-alert types, where $\mathcal{T}$ =$\{$*ICMP_PING_NMAP, SCAN_NMAP_TCP, FTP_Glob_Expansion, Rsh, Mstream_Zombie*$\}$. Further assume the following *may-prepare-for* relations exist: *ICMP_PING_NMAP may prepare for SCAN_NMAP_TCP, SCAN_NMAP_TCP may prepare for FTP_Glob_Expansion, FTP_Glob_Expansion may prepare for Rsh*, and *Rsh may prepare for Mstream_Zombie*. Thus it is clear *ICMP_PING_NMAP may indirectly prepare for Mstream_Zombie* w.r.t. $\mathcal{T}$.

Intuitively, given two hyper-alert types T and T', *T may prepare for T'* if there exist a type T alert t and a type T' alert t' such that *t prepares for t'*. *May-indirectly-prepare-for* relation, which is a natural extension of *may-prepare-for* relation, is defined through a sequence of *may-prepare-for* relations.

Definition 10 *Given a set $\mathcal{T}$ of hyper-alert types and two alerts t and t', where $Type(t)$ and $Type(t') \in \mathcal{T}$ and $t.end\_time < t'.begin\_time$, t* may indirectly prepare for *t' if $Type(t)$ may indirectly prepare for $Type(t')$ w.r.t. $\mathcal{T}$. Given a sequence of alerts t, t_1, ..., t_k, t' where $k > 0$, t indirectly prepares for t' if t prepares for t_1, t_i prepares for t_{i+1} for $i = 1, ..., k-1$, and t_k prepares for t'.*

Intuitively, *t may indirectly prepare for t'* if there *may* exist a path from t to t' in an alert correlation graph (with additional alerts), while *t indirectly prepares for t'* if such alerts do exist. We are particularly interested in the case where *t may indirectly prepare for t'* but there do not exist additional alerts showing that *t indirectly prepares for t'*. Indeed, a possible reason for such a situation is that the IDSs miss some critical attacks, which, if detected, would lead to additional alerts showing that *t indirectly prepares for t'*.

A simple way to take advantage of the above observation is to assume a possible causal relationship between alerts t and t' if they belong to different correlation graphs and *t may indirectly prepare for t'*. Let us continue the example in Figure 5.1. If the hyper-alert type *SCAN_NMAP_TCP may prepare for FTP_Glob_Expansion*, which *may prepare for Rsh*, then we have *SCAN_NMAP_TCP may indirectly prepare for Rsh*. Thus, we may hypothesize that *SCAN_NMAP_TCP2 indirectly pre-*

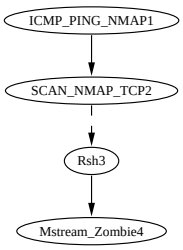


Figure 5.2: A straightforward combination of CG_1 and CG_2

pares for Rsh3. We add a *virtual edge*, displayed in a dashed line, from *SCAN_NMAP_TCP2* to *Rsh3* in Figure 5.2, indicating that there may be some attacks between them that are missed by the IDSs.

Though this simple approach can identify and integrate related correlation graphs and hypothesize about possible causal relationships between alerts, it is limited in several ways. First, the virtual edges generated with this approach provide no information about attacks possibly missed by the IDSs. Second, the virtual edges are determined solely on the basis of prior knowledge about attacks. There is no "reality check." It is possible that the hypothesized virtual edges are not true due to the limitations of the expert knowledge and the lack of information about the missed attacks.

5.1.2 Hypothesizing about Missed Attacks

The *may-prepare-for* and *may-indirectly-prepare-for* relations identified in Definitions 8, 9 and 10 provide additional opportunities to hypothesize and reason about missed attacks, especially unknown variations of known attacks.

Consider two alerts t and t' that belong to different correlation graphs prior to integration. If t *may indirectly prepare for* t', we can then identify possible sequences of hyper-alert types in the form of $T_1, T_2, ..., T_k$ such that $Type(t)$ *may prepare for* T_1, T_i *may prepare for* T_{i+1}, $i = 1, 2, ..., k-1$, and T_k *may prepare for* $Type(t')$. These sequences of hyper-alert types are candidates of attacks possibly missed by the IDSs. (More precisely, variations of these attacks, which could be used by an attacker and then missed by the IDSs, are the actual candidates of missed attacks.) We can then search in the alerts and/or the raw audit data between t and t' to check for signs of these attacks (or their variations). For example, to continue the example in Figure 5.2, we may hypothesize that variations of either *IMAP_Authen_Overflow*, or *FTP_Glob_Expansion*, or both may have been missed by the IDSs based on our prior knowledge about attacks. To better present these hypotheses,

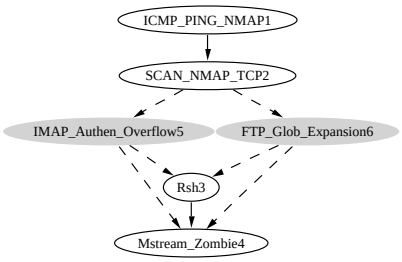


Figure 5.3: Integration of CG_1 and CG_2 with hypotheses of missed attacks

we may add the hypothesized attacks into the correlation graph as virtual nodes (displayed in gray). Figure 5.3 shows the resulting correlation graph.

To facilitate hypothesizing about missed attacks, we encode our knowledge of the relationships between hyper-alert types in a *hyper-alert type graph*, or simply a *type graph*. Let us first introduce the concept of *equality constraint*, which was adapted from our techniques on learning attack strategies (Chapter 4), to help formally describe the notion of type graph.

Definition 11 *Given a pair of hyper-alert types T_1 and T_2, an* equality constraint for (T_1, T_2) *is a conjunction of equalities in the form of $u_1 = v_1 \wedge \cdots \wedge u_n = v_n$, where $u_1, \cdots, u_n$ are attribute names in T_1 and $v_1, \cdots, v_n$ are attribute names in T_2, such that there exist $p(u_1, \cdots, u_n)$ and $p(v_1, \cdots, v_n)$, which are the same predicate with possibly different arguments, in* ExpConseq(T_1) *and* Prereq(T_2)*, respectively. Given a type T_1 alert t_1 and a type T_2 alert t_2, t_1 and t_2* satisfy the equality constraint *if $t_1.u_1 = t_2.v_1 \wedge \cdots \wedge t_1.u_n = t_2.v_n$ evaluates to True.*

Here we give an example. Consider the hyper-alert types *SadmindPing* and *SadmindBufferOverflow* in Example 3. *ExpConseq*(*SadmindPing*) and *Prereq*(*SadmindBufferOverflow*) both contain the predicate *VulnerableSadmind(VictimIP)*. Thus, it is easy to see that *SadmindPing.VictimIP = SadmindBufferOverflow.VictimIP* is an equality constraint for (*SadmindPing*, *SadmindBufferOverflow*). Further consider a type *SadmindPing* alert t_1 and a type *SadmindBufferOverflow* alert t_2. If t_1 and t_2 both have *VictimIP* $= 152.1.19.5$, we can conclude that t_1 and t_2 satisfy the equality constraint.

An equality constraint characterizes the equality relations between attribute values of two alerts when one *prepares for* the other. There may exist several equality constraints for a pair of hyper-alert types. However, if a type T_1 alert t_1 *prepares for* a type T_2 alert t_2, then t_1 and t_2 must satisfy at least one equality constraint. Indeed, t_1 *preparing for* t_2 is equivalent to the conjunction of t_1 and t_2 satisfying at least one equivalent constraint and t_1 occurring before t_2.

Given a set of hyper-alert types (representing the known attacks), by matching all possible predicates in the expanded consequence set and the prerequisite set, we can derive all possible *may-prepare-for* relations between them together with the corresponding equality constraints. This information can help us understand how these known attacks may be combined to launch sequences of attacks, and thus hypothesize about which attacks (more precisely, their variations) may be missed when we observe alerts that *may indirectly prepare for* each other. The following definition formally captures this intuition.

Definition 12 *Given a set $\mathcal{T}$ of hyper-alert types, a* (hyper-alert) type graph *TG over $\mathcal{T}$ is a quadruple (N, E, T, C), where (1) (N, E) is a DAG (directed acyclic graph), (2) T is a bijective mapping from N to $\mathcal{T}$, which maps each node in N to a hyper-alert type in $\mathcal{T}$, (3) there is an edge (n_1, n_2) in E if and only if $T(n_1)$ may prepare for $T(n_2)$, and (4) C is a mapping that maps each edge (n_1, n_2) in E to a set of equality constraints associated with $(T(n_1), T(n_2))$.*

Example 4 *Consider a set of hyper-alert types:* $\mathcal{T}$ = {ICMP_PING_NMAP, SCAN_NMAP_TCP, IMAP_Authen_Overflow, FTP_Glob_Expansion, Rsh, Mstream_Zombie}. *(The specification of these hyper-alert types are given in Table 5.1.) We can compute the type graph over $\mathcal{T}$ as shown in Figure 5.4. The string inside each node is the node name followed by the hyper-alert type name. The label of each edge is the corresponding set of equality constraints.*

Obviously, given multiple correlation graphs that may be integrated together, we can hypothesize about possibly missed attacks that break the attack scenario according to the type graph. Let us revisit the example in Figure 5.1. Given the type graph in Figure 5.4, we can *systematically* hypothesize that the IDSs may have missed variations of *IMAP_Authen_Overflow* and/or

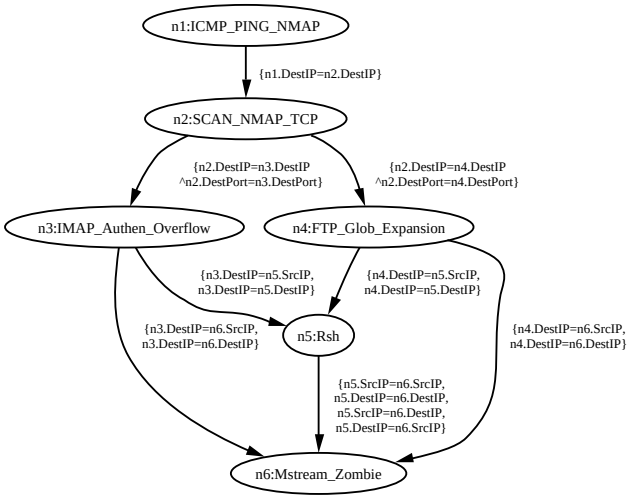


Figure 5.4: An example type graph

Table 5.1: Hyper-alert types used in Example 4 (The set of *fact* attributes for each hyper-alert type is {*SrcIP, SrcPort, DestIP, DestPort*})

Hyper-alert Type	Prerequisite	Consequence
ICMP_PING_NMAP		ExistHost(DestIP)
SCAN_NMAP_TCP	ExistHost(DestIP)	{ExistService(DestIP,DestPort)}
IMAP_Authen_Overflow	ExistService(DestIP,DestPort) ∧VulnerableAuthenticate(DestIP)	{GainAccess(DestIP)}
FTP_Glob_Expansion	ExistService(DestIP,DestPort) ∧VulnerableFTPRequest(DestIP)	{GainAccess(DestIP)}
Rsh	GainAccess(DestIP) ∧GainAccess(SrcIP)	{SystemCompromised(DestIP), SystemCompromised(SrcIP)}
Mstream_Zombie	SystemCompromised(DestIP) ∧SystemCompromised(SrcIP)	{ReadyForDDOSAttack(DestIP), ReadyForDDOSAttack(SrcIP)}

FTP_Glob_Expansion attacks. As a result, we obtain the integrated correlation graph shown in Figure 5.3.

5.1.3 Reasoning about Missed Attacks

In a type graph, the label of an edge encodes all possible equality constraints for the corresponding pair of hyper-alert types. Moreover, even if two hyper-alert types are not adjacent to

each other, they may still satisfy some constraints if they are connected through some intermediate nodes (hyper-alert types) in the type graph (due to the equality constraints those intermediate nodes must satisfy). For example, consider nodes $n2$, $n3$, and $n5$ in Figure 5.4. There is an equality constraint $n2.DestIP = n3.DestIP \wedge n2.DestPort = n3.DestPort$ for ($n2$, $n3$), and two equality constraints $n3.DestIP = n5.SrcIP$ and $n3.DestIP = n5.DestIP$ for ($n3$, $n5$). Take together, these imply $n2.DestIP = n5.SrcIP$ or $n2.DestIP = n5.DestIP$. In other words, if a type *SCAN_NMAP_TCP* alert *indirectly prepares for* a type *Rsh* alert (through a type *IMAP_Authen_Overflow* alert), together they must satisfy one of these two constraints. We obtain the same constraints if we consider nodes $n2$, $n4$, and $n5$ in Figure 5.4. In general, we can derive constraints for two hyper-alert types when one *may indirectly prepare for* the other. Informally, we call such a constraint an *indirect equality constraint*. These constraints can be used to study whether two alerts in two different correlation graphs could be indirectly related. This in turn allows us to filter out incorrectly hypothesized attacks.

Indirect equality constraints can be considered a generalization of the equality constraints specified in Definition 11. In this chapter, we combine the terminology and simply refer to an indirect equality constraint as an equality constraint when it is not necessary to distinguish between them.

To take advantage of the above observation, we must derive indirect equality constraints. In the following, we will first present an algorithm to compute indirect equality constraints for *two* hyper-alert types where one *may indirectly prepare for* the other, and then extend it to compute indirect equality constraints for *all pairs* of hyper-alert types at the same time. We will also discuss how to use such indirect equality constraints to reason about missed attacks.

Computing Indirect Equality Constraints between Two Hyper-Alert Types

Before we discuss the algorithm on how to compute indirect equality constraints, let $\langle T_1, T_2, \cdots, T_k \rangle$ denotes a directed path $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_k$ in a type graph. For convenience, we use $Label(ec)$ to denote the corresponding path in a type graph that produce the equality constraint ec. Algorithm 2 shown in Figure 5.5 outlines an approach for generating the set of indirect equality constraints between two hyper-alert types T and T' where T *may indirectly prepare for* T'. We assume a type graph TG is already constructed from a set of hyper-alert types, which are specified based on all the attacks known to the IDSs (or equivalently, the set of signatures). For each pair of hyper-alert types T and T', this algorithm identifies all paths from T to T' in the type

Algorithm 2: Computation of Indirect Equality Constraints for Two Hyper-Alert Types.
Input: A type graph TG, and two hyper-alert types T and T' in TG,
where T *may indirectly prepare for* T'.
Output: A set of equality constraints for T and T'.
Method:

1. Let $Result = \emptyset$.
2. **For** each path $\langle T, T_1, \cdots, T_k, T' \rangle$ in TG
3. Denote T as T_0, and T' as T_{k+1}.
4. **For** each combination of constraints $ec_1, ec_2, \cdots, ec_{k+1}$,
where ec_i is an equality constraint for (T_{i-1}, T_i)
5. Let $S(T_0.a_i) = \{T_0.a_i\}$,
where $T_0.a_i$, $i = 1, 2, \cdots, l$, are all the attributes of T_0 that appear in ec_1.
6. **For** $j = 1$ to $k + 1$
7. **For** each conjunct $T_{j-1}.a = T_j.b$ in ec_j
8. **For** each $S(T_0.a_i)$ that contains $T_{j-1}.a$
9. Let $S(T_0.a_i) = S(T_0.a_i) \cup \{T_j.b\}$.
10. Remove variables of T_{j-1} from each $S(T_0.a_i)$, $i = 1, 2, \cdots, l$.
11. Let $temp = \emptyset$.
12. **For** each non-empty $S(T_0.a_i)$ and each $T_{k+1}.b$ in $S(T_0.a_i)$
13. Let $temp = temp \cup \{T_0.a_i = T_{k+1}.b\}$.
14. Let ec be the conjunction of all elements in $temp$.
15. **If** ec is in $Result$ **Then**
16. Let $Label(ec) = Label(ec) \cup \{\langle T, T_1, \cdots, T_k, T' \rangle\}$
17. **Else** Let $Label(ec) = \{\langle T, T_1, \cdots, T_k, T' \rangle\}$, and $Result = Result \cup \{ec\}$.
18. **Return** $Result$.

End

Figure 5.5: Algorithm to compute indirect equality constraints for two hyper-alert types

graph, and computes an indirect equality constraint for each combination of equality constraints between consecutive hyper-alert types along the path. The basic idea is to propagate the equality relations between attributes of hyper-alert types (in the algorithm, we use $S(T_0.a_i)$ to propagate equality relations and get attribute names along the path $\langle T_0, T_1, \cdots, T_{k+1} \rangle$). We also label each indirect equality constraint with the corresponding path that produces the constraint. This provides guidelines for hypothesizing about missed attacks. The usefulness of this algorithm is demonstrated by Lemma 5.1.1.

Lemma 5.1.1 *Consider a type graph TG and two alerts t and t', where $Type(t)$ and $Type(t')$ are in TG. Assume Algorithm 2 (Figure 5.5) outputs a set C of equality constraints for $Type(t)$ and*

$Type(t')$. *If* C *is non-empty and* t *indirectly prepares for* t'*, then* t *and* t' *must satisfy at least one equality constraint in* C.

Proof: According to Definition 10, if t *indirectly prepares for* t', there must exist a sequence of alerts t_1, ..., t_k, where $k > 0$, such that t *prepares for* t_1, t_i *prepares for* t_{i+1} for $i = 1, ..., k-1$, and t_k *prepares for* t'. Thus, we have $Type(t)$ *may prepare for* $Type(t_1)$, $Type(t_i)$ *may prepare for* $Type(t_{i+1})$ for $i = 1, ..., k-1$, and $Type(t_k)$ *may prepare for* $Type(t')$. Following the convention of Algorithm 2, we denote $Type(t)$ as T_0, $Type(t_i)$ as T_i, where $i = 1, ..., k$, and $Type(t')$ as T_{k+1}. It is easy to see there must be a path T_0, T_1, ..., T_{k+1} in the corresponding type graph TG. For convenience, we also denote t as t_0, and t' as t_{k+1}.

If t_i *prepares for* t_{i+1}, we can conclude t_i and t_{i+1} must satisfy at least one equality constraint for (T_i, T_{i+1}). This is because if t_i and t_{i+1} does not satisfy any equality constraints for (T_i, T_{i+1}), then none of the instantiated predicates in *ExpConseq*(t_i) can match any in *Prereq*(t_{i+1}), which violates the given condition that t_i *prepares for* t_{i+1}. For $i = 0, 1, ..., k$, we denote the constraint t_i and t_{i+1} satisfy as ec_{i+1}. According to Figure 5.5, Algorithm 2 will process the path T_0, T_1, ..., T_{k+1} (in step 2) and the combination of equality constraints ec_1, ec_2, ..., ec_{k+1} that t_0, t_1, ..., t_{k+1} satisfy (in step 4).

Now consider the process of the above sequence of equality constraints in steps 5 to 10. For each $S(T_0.a_i)$, we can prove by induction that all attributes $T_j.b$ added into $S(T_0.a_i)$ are equal to $T_0.a_i$, since each addition is based on a conjunct $T_{j-1}.a = T_j.b$, where $T_{j-1}.a$ is already in $S(T_0.a)$. Further because step 10 removes the attributes of T_{j-1}, only attributes of T_{k+1} remain in $S(T_0.a_i)$, $i = 1, 2, ..., l$. Thus, after step 10, each $S(T_0.a_i)$ includes all the attributes of T_{k+1} that are equal to $T_0.a_i$, where $i = 1, 2, ..., l$. Steps 11 to 14 then transform these equality relations into a conjunctive formula ec. Since the sequence of constraints ec_i, $i = 1, 2, ..., k+1$, where each ec_i is satisfied by t_{i-1} and t_i, is used in the above process, we can easily conclude that t_0 (t) and t_{k+1} (t') satisfy ec. Thus, if t *indirectly prepares for* t', they must satisfy at least one equality constraint in C.

Example 5 *Consider the type graph in Figure 5.4 and two hyper-alert types* SCAN_NMAP_TCP *(node* $n2$*) and* Rsh *(node* $n5$*). Using Algorithm 2 in Figure 5.5, we can easily compute the indirect equality constraints for them:* $\{n2.DestIP = n5.DestIP, n2.DestIP = n5.SrcIP\}$.

Both indirect equality constraints are labeled with two paths: one path is ⟨SCAN_NMAP_TCP, IMAP_Authen_Overflow, Rsh⟩, *and the other is* ⟨SCAN_NMAP_TCP, FTP_Glob_Expansion, Rsh⟩.

Given two hyper-alert types T and T' in a type graph, Algorithm 2 in Figure 5.5 derives the indirect equality constraints between them by considering all combinations of (direct) equality constraints between two adjacent hyper-alert types in each path from T to T'. Theoretically, there is a potential problem of combinatorial explosion. However, in practice, because of the limited number of predicates and hyper-alert types, this problem should be tractable. Moreover, this algorithm only needs to be executed once for two given hyper-alert types in a type graph. Thus, the cost of this algorithm does not have significant impact on alert correlation.

Computing Indirect Equality Constraints for All Pairs of Hyper-Alert Types

Algorithm 2 in Figure 5.5 focuses on the problem of computing indirect equality constraints for two hyper-alert types. An extension to this problem is: given a set $\mathcal{T}$ of n hyper-alert types, how to calculate indirect equality constraints for all pairs of hyper-alert types where the first one in the pair *may indirectly prepare for* the second one? This is a realistic problem, since we do need to get the equality constraints between all pairs of hyper-alert types to reason about missed attacks.

We can apply Algorithm 2 for up to n^2 times, once for each pair of hyper-alert types (T_i, T_j) where T_i *may indirectly prepare for* T_j ($1 \leq i, j \leq n$). Unfortunately, this is not an efficient solution. To see the inefficiency more clearly, consider a path from T_i to T_j where T_j is further connected to T_k by an edge. If we compute the indirect equality constraints between all pairs of hyper-alert types with Algorithm 2, the computation for T_i and T_k with the path involving T_i, T_j, and T_k will repeat the computation for T_i and T_j with the same path from T_i and T_j. A better approach is to reuse the equality constraints already computed for T_i and T_j to derive those for T_i and T_k.

To take advantage of the above observation, Algorithm 3 (shown in Figure 5.6) outlines a method to compute equality constraints for all pairs of hyper-alert types at the same time. The output of Algorithm 3 is a constraint matrix. Given n hyper-alert types $T_1, T_2, \cdots, T_n$, a *constraint matrix* M is a $n \times n$ table, where the cell $M(i, j)$ ($1 \leq i, j \leq n$) consists of all and only equality constraints for (T_i, T_j) if T_i *may (indirectly) prepare for* T_j.

In Algorithm 3, for convenience, we use $Label(ec)$ to denote the corresponding path in a

Algorithm 3. Computation of Equality Constraints for All Pairs of Hyper-alert Types
Input: A type graph TG over a set of hyper-alert types $\{T_1, T_2, \cdots, T_n\}$.
Output: A $n \times n$ constraint matrix M with each cell $M(i, j)$ containing a set of equality constraints for (T_i, T_j).
Method:

1. Create a $n \times (n-1)$ matrix L, and initialize each cell of L to empty.
   // Each cell $L(i, j)$ is intended to contain the equality constraints (marked with path
   // labels) for the length j paths in TG starting from type T_i.
2. $k = 1$. // The variable k represents the possible lengths of the paths in TG
3. **For** each edge (T_i, T_j) in the type graph TG
4. **For** each equality constraint ec for (T_i, T_j)
5. $Label(ec) = \langle T_i, T_j \rangle$; $L(i, 1) = L(i, 1) \cup \{ec\}$.
6. **For** $k = 2$ to $n - 1$
7. **For** $i = 1$ to n
8. **For** each equality constraint ec in $L(i, k-1)$
9. Get the last hyper-alert type T in $Label(ec)$.
10. Get the set $\mathcal{T}$ of hyper-alert types where T has edges to each type in $\mathcal{T}$.
11. **For** each hyper-alert type T' in $\mathcal{T}$
12. **For** each equality constraint ec' where $Label(ec') = \langle T, T' \rangle$
13. Get a constraint ec'' via **InferredConstraint** (ec, ec').
14. Let $Label(ec'') = \langle Label(ec), T' \rangle$.
15. Let $L(i, k) = L(i, k) \cup \{ec''\}$.
16. **For** $i = 1$ to n
17. **For** $j = 1$ to n
18. In row i of L, find all equality constraints where the last type in their labels is T_j, and put these constraints into the cell $M(i, j)$.
19. **Output** the matrix M.

End.

Subroutine InferredConstraint
Input: An equality constraints ec for (T_x, T_y) and an equality constraint ec' for (T_y, T_z).
Output: An equality constraint ec'' for (T_x, T_z) derived from ec and ec'.
Method:

1. Let $ec'' = \{\}$.
2. **For** each conjunct $T_x.u = T_y.v$ in ec
3. **If** there exists a conjunct $T_y.v = T_z.w$ in ec', **then**
4. Let $T_x.u = T_z.w$ be a conjunct in ec''.
5. **Output** ec''.

End.

Figure 5.6: Algorithm to compute indirect equality constraints for all pairs of hyper-alert types

type graph that produces the equality constraint ec, and we use $Label(ec_2) = \langle Label(ec_1), T\rangle$ to denote that ec_2's label is ec_1's label appended by a type T. As we discussed in Algorithm 2, a path between hyper-alert types T_i and T_j in a type graph represents that T_i *may indirectly prepare for* T_j, and we can derive equality constraints for (T_i, T_j) by reasoning about the equality constraints along the path. The basic idea behind Algorithm 3 is to reuse the equality constraints derived from short paths to compute those for long paths. In a type graph with n hyper-alert types (n nodes), the possible lengths of paths range from 1 to $n-1$. To compute the indirect equality constraints for a path with length k ($1 < k \leq n-1$), it is always possible to first carry out the computation of the (indirect) equality constraints for length $k-1$ paths, and then combine the results for such paths with the equality constraints for individual edges to get the constraints for length k paths. Lemma 5.1.2 ensures that Algorithm 3 can derive all and only equality constraints for T_i and T_j.

Lemma 5.1.2 *Given a type graph TG over a set of hyper-alert types $\{T_1, T_2, \cdots, T_n\}$, Algorithm 3 outputs all and only equality constraints for (T_i, T_j) in the cell $M(i, j)$ $(1 \leq i, j \leq n)$.*

Proof: According to the definition of (indirect) equality constraint, there may be one or more equality constraints for (T_i, T_j) if T_i *may (indirectly) prepare for* T_j. In other words, there may be one or more equality constraints for (T_i, T_j) if there is a path between T_i and T_j in TG. In the following, we first prove by induction that each equality constraint that can be derived for (T_i, T_j) from a length k path p ($k = 1, 2, \cdots, n-1$) is labeled with p and stored in $L(i, k)$.

1. When $k = 1$, for each length 1 path $p = \langle T_i, T_j\rangle$ (which is an edge in TG), lines 3 through 5 put all equality constraints for (T_i, T_j) into cell $L(i, 1)$, each of which is labeled with the corresponding edge.

2. Assume for any T_i, T_j in TG, all the equality constraints that can be derived for (T_i, T_j) from a length m path are in $L(i, m)$ with the corresponding path labels. Now we show that for any T_i, T_j in TG, all the equality constraints that can be derived for (T_i, T_j) from a length $m+1$ path are in $L(i, m+1)$ with the corresponding path labels.

   Consider lines 7 through 10. For each T_i, these lines find all the edges that can follow each length m path starting with T_i. Thus, they can enumerate all length $m+1$ paths. For convenience, we denote each length $k = m+1$ path $p = \langle T_i, \cdots, T_s, T_j\rangle$ as composed of two connected paths: $p' = \langle T_i, \cdots, T_s\rangle$ and $p'' = \langle T_s, T_j\rangle$, where the length of p' is m and p'' is

an edge in TG. According to the induction assumption, the equality constraints that can be derived for (T_i, T_s) from p' are in $L(i, m)$ with the label p'.

Consider lines 11 through 15. For each equality constraint ec in $L(i, m)$ with label $\langle T_i, \cdots, T_s \rangle$ and each equality constraint ec' for (T_s, T_j), the subroutine **InferredConstraint** (ec, ec') (line 13) derives the equality ec'' inferred by ec and ec', which is an equality constraint for (T_i, T_j). This equality constraint is then labeled with the path $p = \langle T_i, \cdots, T_s, T_j \rangle$ and then added into $L(i, m + 1)$. Since lines 11 through 15 consider all combinations of the equality constraints in $L(i, m)$ and the (direct) equality constraints for each edge that follow a length m path, they can find the equality constraints that can be derived for all length $m+1$ paths starting from T_i. Therefore, for any T_i, T_j in TG, all the equality constraints that can be derived for (T_i, T_j) from a length $m + 1$ path are in $L(i, m + 1)$ with the corresponding path labels.

Further considering lines 16 through 18, which copy all equality constraints derived from paths between T_i and T_j into $M(i, j)$, and that the possible path length in TG is from 1 to $n - 1$, we can conclude that all equality constraints for (T_i, T_j) are in $M(i, j)$. Moreover, during Algorithm 3, since we only add the inferred (indirect) equality constraints into L with path labels (lines 14 and 15), and we only move equality constraints derived from paths from T_i to T_j into $M(i, j)$ (lines 16 through 18), $M(i, j)$ only contains equality constraints for (T_i, T_j).

Example 6 *To continue Example 5, we may use Algorithm 3 to derive the sets of equality constraints for all pairs of hyper-alert types in Figure 5.4 where one of the pair may (indirectly) prepare for the other. The results are given in Table 5.2, in which each cell contains the equality constraints for the hyper-alert types in the given row and the column. (To save space, we use node names to represent the corresponding hyper-alert types and omit the labels for each equality constraint.)*

Similar to Algorithm 2, Algorithm 3 is also executed only once before alert correlation, and thus does not introduce significant overhead during alert correlation.

Using (Indirect) Equality Constraints

The equality constraints derived for indirectly related hyper-alert types can be used to determine if two correlation graphs can be integrated. Given two correlation graphs CG_1 and CG_2,

Table 5.2: Equality constraints for hyper-alert types in Figure 5.4 where one *may (indirectly) prepare for* the other.

	n1	n2	n3	n4	n5	n6
n1	/	{n1.DestIP = n2.DestIP}	{n1.DestIP = n3.DestIP}	{n1.DestIP = n4.DestIP}	{n1.DestIP = n5.DestIP, n1.DestIP = n5.SrcIP}	{n1.DestIP = n6.DestIP, n1.DestIP = n6.SrcIP}
n2	/	/	{n2.DestIP = n3.DestIP ∧ n2.DestPort = n3.DestPort}	{n2.DestIP = n4.DestIP ∧ n2.DestPort = n4.DestPort}	{n2.DestIP = n5.DestIP, n2.DestIP = n5.SrcIP}	{n2.DestIP = n6.DestIP, n2.DestIP = n6.SrcIP}
n3	/	/	/	/	{n3.DestIP = n5.DestIP, n3.DestIP = n5.SrcIP}	{n3.DestIP = n6.DestIP, n3.DestIP = n6.SrcIP}
n4	/	/	/	/	{n4.DestIP = n5.DestIP, n4.DestIP = n5.SrcIP}	{n4.DestIP = n6.DestIP, n4.DestIP = n6.SrcIP}
n5	/	/	/	/	/	{n5.SrcIP = n6.SrcIP, n5.DestIP = n6.DestIP, n5.SrcIP = n6.DestIP, n5.DestIP = n6.SrcIP}
n6	/	/	/	/	/	/

we can integrate CG_1 and CG_2 if there exist an alert t_1 in CG_1 and an alert t_2 in CG_2 such that (1) t_1 and t_2 satisfy at least one equality constraint for $(Type(t_1), Type(t_2))$ and (2) t_1's timestamp is before t_2's timestamp.

Moreover, such equality constraints can also facilitate the hypotheses of missed attacks. When integrating two correlation graphs CG and CG', we can hypothesize missed attacks only for such pairs of alerts t and t' that (1) t and t' belong to CG and CG', respectively, and (2) t *indirectly prepare for* t'. Specifically, for each equality constraint ec that t and t' satisfy, we can add the paths in $Label(ec)$ into the integrated correlation graph. Since each path in $Label(ec)$ is in the form of $\langle Type(t), T_1, ..., T_k, Type(t')\rangle$, $Type(t)$ and $Type(t')$ are merged with t and t', respectively, and the rest of the path is added as a virtual path consisting of virtual nodes and edges from t to t'.

Note that this may add incorrect hypotheses into the integrated correlation graph. We will present techniques to validate these hypotheses with raw audit data in Section 5.1.5.

Let us illustrate how to take advantage of indirect equality constraints to hypothesize missed attacks. Consider two correlation graphs in Figure 5.1, if an earlier alert *SCAN_NMAP_TCP2* and a later alert *Rsh3* have the same destination IP address, they then satisfy an equality constraint ec: *SCAN_NMAP_TCP.DestIP* $=$ *Rsh.DestIP*, which is an indirect equality constraint for (*SCAN_NMAP_TCP*, *Rsh*) shown in Table 5.2. Thus, we can integrate CG_1 and CG_2 and hypothesize missed attacks based on the label associated with the above equality constraint. For instance, the label associated with the above equality constraint ($Label(ec)$) consists of two paths: $\langle$*SCAN_NMAP_TCP*, *IMAP_Authen_Overflow*, *Rsh*$\rangle$ and $\langle$*SCAN_NMAP_TCP*, *FTP_Glob_Expansion*, *Rsh*$\rangle$. Thus, we can hypothesize two missed attacks: one is *IMAP_Authen_Overflow5*, and the other is *FTP_Glob_Expansion6*. The hypothesis process may continue until all such pairs of alerts are examined.

5.1.4 Inferring Attribute Values for Hypothesized Attacks

The (direct or indirect) equality constraints not only help hypothesize about missed attacks, but also provide an opportunity to make inferences about the attribute values of hypothesized attacks. In other words, we may further hypothesize about the missed attack *instances*. For example, suppose we hypothesize an *IMAP_Authen_Overflow* attack after a *SCAN_NMAP_TCP* alert and before a *Rsh* alert such that *SCAN_NMAP_TCP prepares for IMAP_Authen_Overflow*, which then *prepares for Rsh*. From Table 5.2, we know that *SCAN_NMAP_TCP* and *IMAP_Authen_Overflow* have the same destination IP address and destination port, and the destination IP address of alert *IMAP_Authen_Overflow* is the same as either the source or the destination IP address of *Rsh*.

In general, we can use the equality constraints between the intrusion alerts and the hypothesized attacks to infer the possible attribute values of these attacks. As a special attribute, we estimate the timestamp of a hypothesized attack as a possible range. That is, if an attack t_h is hypothesized as an intermediate step between two intrusion alerts t and t', where t occurs before t', then the possible range of t_h's timestamp is $[t.end\_time, t'.begin\_time]$. Let us first look at an example.

Example 7 *Consider the integrated correlation graph shown in Figure 5.3. Let us infer attribute values for the hypothesized attack (instance)* FTP_Glob_Expansion6. *Suppose the IDS reported that*

the destination IP addresses of alerts SCAN_NMAP_TCP2 *and* Rsh3 *were both* $152.1.19.5$*, and the destination port of* SCAN_NMAP_TCP2 *was* 21*. Following the earlier convention in Figure 5.4, we use nodes* $n2$*,* $n4$*, and* $n5$ *to denote hyper-alert types* SCAN_NMAP_TCP, FTP_Glob_Expansion, *and* Rsh, *respectively. It is easy to see that* SCAN_NMAP_TCP2 *and* Rsh3 *satisfy the equality constraint* n2.DestIP $=$ n5.DestIP. *Based on the constraint matrix in Table 5.2, we can see this equality constraint is derived from the following two equality constraints:*

1. n2.DestIP =n4.DestIP $\wedge$ n2.DestPort $=$ n4.DestPort *for (*SCAN_NMAP_TCP, FTP_Glob_Expansion*), and*

2. n4.DestIP =n5.DestIP *for (*FTP_Glob_Expansion, Rsh*).*

Thus, the hypothesized attack FTP_Glob_Expansion6 *should satisfy both of these equality constraints. As a result, we can infer that the destination IP address and the destination port of* FTP_Glob_Expansion6 *are* $152.1.19.5$ *and* 21*, respectively.*

We generalize Example 7 into Algorithm 4, which is shown in Figure 5.7, to infer attribute values of hypothesized attack instances. Intuitively, given two alerts t and t', for each hypothesized attack T_i along a path of hypothesized attacks, we get the set C_i of equality constraints for $(Type(t), T_i)$) and the set C_i' of equality constraints for $(T_i, Type(t'))$, respectively. Any combination of equality constraints ec_i in C_i and ec_i' in C_i' that result in an equality constraint that t and t' satisfy can be used to infer the attribute values of the hypothesized instance of attack T_i. In other words, we infer the attributes of the hypothesized attack instance to be the same as those of t and t' as indicated by the equality constraints.

In Algorithm 4, line 1 gets the set C of equality constraints that alerts t and t' satisfy and that are associated with the given path in the type graph. Lines 2 through 7 are a loop to infer attribute values for each hypothesized attack in given path in TG, which is a possible sequence of attacks that have happened. Line 3 obtains the set of equality constraints C_i for $(Type(h), T_i)$ and C_i' for $(T_i, Type(h'))$ that are associated with the two halves of the given path (through the constraint matrix). In the following steps, the algorithm tries all combinations of equality constraints

Algorithm 4. Inferring Attribute Values for Hypothesized Attacks
Input: A type graph TG for a set $\mathcal{T}$ of hyper-alert types, a path $P = \langle T, T_1, T_2, \cdots, T_k, T' \rangle$ in TG, a type T alert t, and a type T' alert t', where t *may indirectly prepare for* t'.
Output: A set H_i of hypothesized attack instances for each type T_i, where $i = 1, 2, \cdots, k$.
Method:
// We assume the constraint matrix M for $\mathcal{T}$ has been computed, in which each equality
// constraint ec in M is labeled with the corresponding path $Label(ec)$ in TG.
1. Get a set C of equality constraints such that for each $ec \in C$, t and t' satisfy ec and $Label(ec) = P$.
2. **For** each hypothesized attack T_i
3. Get the set of equality constraints C_i for (T, T_i) whose label is $\langle T, T_1, ..., T_i \rangle$; get the set of equality constraints C'_i for (T_i, T') whose label is $\langle T_i, ..., T_k, T' \rangle$; let $H_i = \{\}$.
4. **For** each combination of $ec_i \in C_i$ and $ec'_i \in C'_i$
5. **If** ec_i and ec'_i imply any $ec \in C$ **Then**
6. Generate a type T_i alert t_i; set all the attributes of t_i that are equal to some attributes of T in ec_i to the corresponding attribute values of t; similarly, set all the attributes of t_i that are equal to some attributes of T' in ec'_i to the corresponding attribute values of t'; set the remaining attributes (if any) to *Unknown*; let the timestamp of t_i be $[t.end\_time, t'.begin\_time]$.
7. Let $H_i = H_i \cup \{t_i\}$.
8. **Output** H_i.
End.

Figure 5.7: Algorithm to infer attribute values for hypothesized attacks

in C_i and C'_i, and infers the attribute values of hypothesized type T_i attacks . Each hypothesized attack instance of Type T_i is derived through two equality constraints ($ec_i \in C_i$ and $ec'_i \in C'_i$). The condition checking in line 5 guarantees that the inferred attribute values do not conflict (otherwise, the corresponding combination of equality constraints could not lead to an equality constraint that t and t' satisfy). Finally, line 8 outputs the hypothesized attack instances for each attack type (hyper-alert type) T_i, where $i = 1, 2, ..., k$.

We make several observations about Algorithm 4. First, given two alerts t and t', the hypotheses of missed attack instances between t and t' are *specific* to the paths between t and t'. In other words, the hypothesis of each missed attack is supported by the possibility that an attacker has launched a sequence of attacks (or, more precisely, attack instances), including the hypothesized one, that leads from t to t'. This also implies that Algorithm 4 should be performed multiple times when hypothesizing about missed attack instances based on a given pair of alerts. Second, the two

alerts t and t' and a given path in the type graph may lead to multiple instances of each attack type, since there may be multiple (direct or indirect) equality constraints for each pair of hyper-alert types. Third, the hypothesized attack instances are usually not as specific as regular alerts. That is, a hypothesized attack may have unknown values on some attributes, which cannot be inferred from the available alerts.

Algorithm 4 is essentially a best-effort "guess" of what could have been missed by IDSs. The hypothesized attack instances can certainly be wrong. In the next subsection, we investigate how to prune those incorrectly "guessed" attack instances using a complementary information source, the raw audit data.

5.1.5 Pruning Hypothesized Attacks with Raw Audit Data

The hypothesized attack instances can be further validated using raw audit data. For example, we may hypothesize that there is a variation of *FTP_Glob_Expansion* attack between a *SCAN_NMAP_TCP* alert and a *Rsh* alert. However, if there is no ftp activity related to the victim host between these two alerts, we can easily conclude that our hypothesis is incorrect. By doing so we further narrow the hypothesized attacks down to the meaningful ones.

To take advantage of this observation, we extend our model to associate a "filtering condition" with each hyper-alert type. Assuming that the raw audit data set consists of a sequence of audit records, we can extract attribute values from each audit record directly, or through inference. For example, we may extract the source IP address from a tcpdump record directly, or infer the type of service using the port and payload information. For the sake of presentation, we call such attributes obtained from the raw audit data *audit attributes*.

Definition 13 *Given a hyper-alert type T and a set A of audit attributes, a* filtering condition for T w.r.t. A *is a logical formula involving audit attribute names in A, which evaluates to* True *or* False *if the audit attribute names are replaced with specific values.*

Example 8 *Consider the following set of audit attributes:* A = {SrcIP, SrcPort, DestIP, DestPort, Protocol, FrameArrivalTime}. *Given a hyper-alert type* FTP_Glob_Expansion*, we may have a simple logical formula* "Protocol = ftp" *as a filtering condition for type* FTP_Glob_Expansion *w.r.t.*

A.

Intuitively, a filtering condition for a hyper-alert type is a necessary condition for the corresponding attack or its variations. We can simply evaluate the condition to prune some incorrect hypotheses. If a filtering condition is evaluated to True, the corresponding attack may have happened; if it is evaluated to False, the corresponding hypothesized attack could not have happened, and should be ruled out.

The above filtering condition is essentially prior knowledge of known attacks. There is an additional opportunity to prune incorrect hypotheses if we further consider the inferred attribute values of the hypothesized attacks. For example, if we can infer that the destination IP address of a hypothesized *FTP_Glob_Expansion* attack is $152.1.19.5$, we may further check whether there is ftp activities destined to $152.1.19.5$ in Example 8. In other words, we can revise the filtering condition in Example 8 to "*Protocol = ftp* $\wedge$ *DestIP* = $152.1.19.5$" for the hypothesized attack instance.

To formalize this idea, we introduce the notion of *filtering template*.

Definition 14 *Given a hyper-alert type* $T = (fact, prerequisite, consequence)$ *and a set* A *of audit attributes, a* filtering template for T w.r.t. A *is a logical formula involving variables in* $fact$ *and* A*, which evaluates to* True *or* False *if these variables are replaced with specific values. Given a hypothesized attack* t *of type* T *with a set* f_s *of inferred attributes, where* $f_s \subseteq fact$*, a filtering template* $Temp_f$ *is* instantiatable by t *if all the variables in* $Temp_f$ *are either in* f_s *or in* A*. If a filtering template* $Temp_f$ *is instantiatable by* t*, we can then get an* instantiated filtering condition for t *by replacing the variables in* $Temp_f$ *with the inferred attribute values of* t.

Example 9 *Consider the set of audit attributes:* A = {SrcIP, SrcPort, DestIP, DestPort, Protocol, FrameArrivalTime}. *Given a hyper-alert type* FTP_Glob_Expansion *(See Table 5.1), we may have a filtering template* "A.DestIP = FTP_Glob_Expansion.DestIP" *as a filtering template for type* FTP_Glob_Expansion *w.r.t.* A*. Assume there is a hypothesized attack* FTP_Glob_Expansion6 *with an inferred attribute* DestIP = $152.1.19.5$*. The above filtering template is then instantiatable by* FTP_Glob_Expansion6*, and can be instantiated to* "A.DestIP = $152.1.19.5$".

Intuitively, a filtering template is a template of filtering condition for a type of attack. Given a hypothesized attack with a set of inferred attributes, we may convert a filtering template into a filtering condition if all the attack attributes that appear in the filtering template have specific inferred values. To distinguish from the filtering condition defined in Definition 13, we call those defined for hyper-alert types the *predefined filtering conditions*, and those instantiated from hypothesized attacks the *instantiated filtering conditions*. We can then use such an instantiated filtering condition in the same way as the predefined filtering conditions.

Pruning incorrectly hypothesized attacks with predefined and/or instantiated filtering conditions is pretty straightforward. Before correlating alerts, we specify filtering conditions and filtering templates for each hyper-alert type. When hypothesizing and reasoning about missed attacks, for each hypothesized attack with a possible range of its timestamp and a set of inferred attributes, we first determine whether each filtering template corresponding to the hypothesized attack is instantiatable by this hypothesis w.r.t. the raw audit attributes. If the answer is positive, we derive an instantiated filtering condition for this filtering template. We then compute the actual filtering condition as the conjunction of the predefined filtering condition and all the instantiated filtering conditions.

To validate a hypothesized attack, we can search through the raw audit records during the time period when the hypothesized attack may have happened, and evaluate the filtering condition using the values of the attributes of each raw audit record. To continue Examples 8 and 9, we can generate the final filtering condition as "*Protocol* = *ftp* $\wedge$ *DestIP* = $152.1.19.5$" to validate (or deny) *FTP_Glob_Expansion6*. If there is no ftp traffic associated with the destination IP address $152.1.19.5$ between alerts *SCAN_NMAP_TCP2* and *Rsh3*, i.e., the above filtering condition evaluates to False for all audit records, we can conclude that the *FTP_Glob_Expansion6* attack is falsely hypothesized. As a result, the integrated correlation graph in Figure 5.3 can be refined to the one in Figure 5.8.

A limitation of using filtering conditions is that human users must specify the conditions associated with each hyper-alert type. It has at least two implications. First, it could be time consuming to specify such conditions for every known attack. Second, human users may make mistakes during the specification of filtering conditions. In particular, a filtering condition could be too specific to capture the invariant among the variations of a known attack, or too general to filter out enough incorrect hypotheses. Nevertheless, we observe that any filtering condition may help reduce incorrectly hypothesized attacks, even if it is very general. In our experiments, we simply use the protocols over which the attacks are carried out and the inferred attribute values as filtering conditions. It is interesting to study how to get the "right" way to specify filtering conditions.

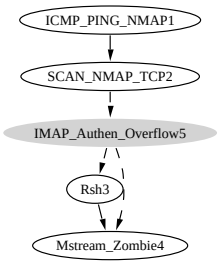


Figure 5.8: Integration of CG_1 and CG_2 after refinement with raw audit data

Another issue is the execution cost. To filter out a hypothesized attack with a filtering condition, we have to examine every audit record during the period of time when the attack could happen. Though there are many ways to optimize the filtering process (e.g., indexing, concurrent examination), the cost is not negligible, especially when the time period is large. Thus, filtering conditions are more suitable for off-line analysis.

5.1.6 Consolidating Hypothesized Attacks

In the earlier subsections, we investigated various techniques to hypothesize and reason about missed attacks. However, our method has not considered the possibility that the same attack may be hypothesized multiple times in different contexts. As a result, an integrated correlation graph may include too many hypothesized attacks. Though it is possible that the same attack are repeated multiple times (as hypothesized), having too many uncertain details reduces the usability of the integrated correlation graph.

Let us look at an example to see this problem more clearly. Consider Figure 5.9, which shows some hypothesized attacks resulting from the integration of CG_1 and CG_2 in Figure 5.1. Assume *ICMP_PING_NMAP1*, *SCAN_NMAP_TCP2* and *Rsh3* all have the same destination IP address $152.1.19.5$. Since two alerts *SCAN_NMAP_TCP2* and *Rsh3* satisfy the equality constraint *SCAN_NMAP_TCP.DestIP* $=$ *Rsh.DestIP*, based on the type graph in Figure 5.4, we hypothesize two attacks: *IMAP_Authen_Overflow5* and *FTP_Glob_Expansion6*, which both have the same destination IP address $152.1.19.5$ derived through attribute value inference. Similarly, *ICMP_PING_NMAP1* and *Rsh3* satisfy the equality constraint *ICMP_PING_NMAP.DestIP* $=$ *Rsh.DestIP*. Thus, we may hypothesize four attacks: *SCAN_NMAP_TCP7*, *IMAP_Authen_Overflow8*, *SCAN_NMAP_TCP9* and

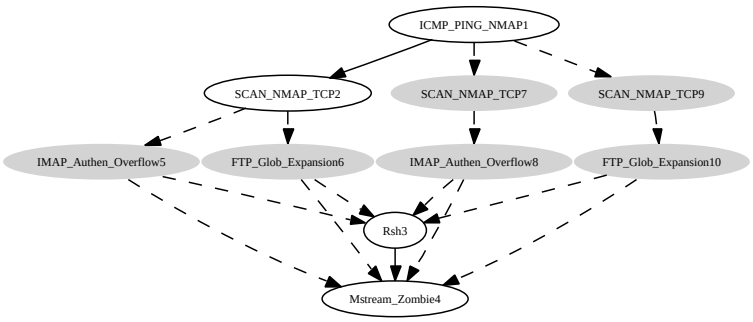


Figure 5.9: Hypothesized attacks when integrating CG_1 and CG_2

FTP_Glob_Expansion10, all with the same destination IP address $152.1.19.5$.

This example leads to two observations. First, it is possible that the hypothesized attack instance *SCAN_NMAP_TCP7* is the same attack as reflected by the existing alert *SCAN_NMAP_TCP2*, but it is also possible that the attacker launched two separate attacks. Similarly, it is equally possible for *IMAP_Authen_Overflow5* and *IMAP_Authen_Overflow8* to be the same attack or two separate attacks. Second, having all these hypothesized attacks makes the integrated correlation graph complex and difficult to understand. Since the hypothesized attacks are all uncertain, having multiple hypotheses for one attack does not give more information. Indeed, if we consider the typical goal of attack hypothesis during intrusion analysis, it is not critical to know how many times an attack has been used in one step of attacks; instead, it is usually more important to know whether an attack has been used or not.

Based on the above observations, we propose to consolidate the hypothesized attacks. Specifically, we remove a hypothesized attack if it may have been detected (as an existing alert), or aggregate a set of hypothesized attacks if they may be the same attack. Our approach is based on the "consistency" between a hypothesized attack and an alert, or the "consistency" among hypothesized attacks. Informally, a set of hypothesized attacks are *consistent* if they could be the same attack, and a hypothesized attack is *consistent* with an alert if this hypothesized attack could have been detected and reflected as the alert.

We first look at the consistency between a hypothesized attack and an alert by examining their attack types, attribute values, and timestamp information. Once a hypothesized attack t_h is identified as consistent with an alert t, we subsume t_h into t by merging t_h and t (as well as the duplicated edges resulting from this merge).

In the following, we first clarify the consistency relations.

Definition 15 *A hypothesized attack t_h is* consistent *with an alert t if (1) t_h and t are of the same type, (2) if t_h and t both have specific values on the same attribute, these two values are the same, and (3) the timestamp of t_h includes the timestamp of t (i.e., $t_h.begin\_time \leq t.begin\_time \wedge t_h.end\_time \geq t.end\_time$).*

The consistency among a set of hypothesized attacks can be defined in a similar way.

Definition 16 *A set $H_h = \{t_1, t_2, \cdots, t_n\}$ of hypothesized attacks is* consistent *if (1) all hypothesized attacks in H_h are of the same type, (2) if more than one hypothesized attacks in H_h have specific values on an attribute, then all these values must be the same, and (3) $\min\{t_i.end\_time | i = 1, 2, ..., n\} > \max\{t_i.begin\_time | i = 1, 2, ..., n\}$ (i.e., the intersection of all the interval-based timestamps is not empty).*

The intuition behind Definition 16 is that a set of hypothesized attacks are consistent if they have the same type, their attribute values do not conflict, and the possible ranges of their interval-based timestamps overlap.

Figure 5.10 outlines an algorithm to consolidate hypothesized attacks. Step 1 groups all hypothesized attacks based on their types. Step 2 starts to process each group. This processing can be divided into two stages. The first stage (steps 3 through 5) reduces the hypothesized attacks based on the consistency relations between hypothesized attacks and alerts. The second stage (steps 6 to 13) partitions each group of hypothesized attacks into subgroups so that all attacks in each subgroup are consistent, consolidates each subgroup into one hypothesized attack, and instantiates the attributes of the hypothesized attack if they are inferable. Step 14 finally outputs the consolidated version of hypothesized attacks.

Consolidating hypothesized attacks helps reduce the number of virtual nodes in an integrated correlation graph. To get a concise attack scenario, the following job is to merge the virtual edges associated with those hypothesized attacks being consolidated. This is trivial: If a hypothesized attack t_h is consolidated based on an alert t, then all virtual edges related to t_h should be

Algorithm 5. Consolidating Hypothesized Attacks
Input: A set S of alerts and a set S_h of hypothesized attacks.
Output: A set S'_h of hypothesized attacks after consolidation.
Method:

1. Partition S_h into groups such that the hypothesized attacks in each group have the same type.
2. **For** each group G_h in S_h
3. **For** each hypothesized attack t_h in G_h
4. **If** t_h is consistent with an alert t in S **then**
5. Remove t_h from G_h, and merge t_h with t.
6. **If** G_h is not empty **then**
7. Partition G_h into maximal subgroups such that the hypothesized attacks in each subgroup are consistent.
8. Replace each subgroup G_s with a hypothesized attack t_h with the same type.
9. **For** each attribute a_i of t_h
10. **If** there exists a hypothesized attack $t'_h \in G_s$ that has a specific value on a_i
11. let $t_h.a_i = t'_h.a_i$,
12. **else** let $t_h.a_i =$ *Unknown*.
13. Add t_h into S'_h.
14. **Output** S'_h.

End.

Figure 5.10: Algorithm to consolidate hypothesized attacks

re-directed to t. Likewise, given a set S_h of hypothesized attacks, if all attacks in S_h can be consolidated into a hypothesized attack t_h, then all virtual edges related to the hypothesized attacks in S_h should be re-directed to t_h.

Our consolidation technique is effective in reducing the size of integrated correlation graphs. For example, in one of our experiments, we have consolidated 137 hypothesized attacks into 5 ones. However, we shall point out that, after consolidation, each hypothesized attack may correspond to multiple instances of missed attacks. In other words, each hypothesized attack in an integrated correlation graph is indeed a place-holder for one or several possible attacks.

5.2 Experimental Results

We have implemented all the techniques we discussed in this paper. In our implementation, we used Java as the programming language, and Microsoft SQL Server 2000 as the database

to store the hyper-alert types, the alert data sets, and the analysis results. We assume the NCSU Intrusion Alert Correlator version 0.2 [27] is used to correlate IDS alerts into correlation graphs. To validate the hypothesized attacks using raw audit data, our implementation uses Ethereal (version 0.9.14) to extract audit attribute values from the raw tcpdump file (i.e., the network audit data). Finally, we use GraphViz [9] to visualize the integrated correlation graphs.

To examine the effectiveness of the proposed techniques, we performed a series of experiments using one of the 2000 DARPA intrusion detection scenario specific data sets, LLDOS 1.0 [77]. LLDOS 1.0 contains a series of attacks in which an attacker probed, broke-in, installed the components necessary to launch a Distributed Denial of Service (DDOS) attack, and actually launched a DDOS attack against an off-site server. The network audit data were collected in both the DMZ and the inside parts of the evaluation network. We used RealSecure Network Sensor 6.0 [52] as the IDS sensor to generate alerts, which are then correlated by the NCSU Intrusion Alert Correlator into correlation graphs.

On constructing the type graph for the experiments, we consider all attacks (represented as hyper-alert types) in the data sets that can be detected by RealSecure Network Sensor 6.0. The specification of these hyper-alert types is given in Table 5.3, the implication relationships between predicates are shown in Table 5.4, and the type graph is given in Figure 5.11. For space reasons, we did not put the isolated nodes (the nodes which do not have edges connecting to them) into the type graph.

Table 5.3: Hyper-alert types used in our experiments (The set of *fact* attributes for each hyper-alert type is {*SrcIP,SrcPort,DestIP,DestPort*}).

Hyper-alert Type	Prerequisite	Consequence
Admind		
DNS_HInfo	ExistService(DestIP,DestPort)	{GainOSInfo(DestIP)}
Email_Almail_Overflow	ExistService(DestIP,DestPort) ∧ VulnerableAlMailPOP3Server (DestIP)	{GainAccess(DestIP)}
Email_Debug	ExistService(DestIP,DestPort) ∧SendMailInDebugMode(DestIP)	{GainAccess(DestIP)}
Email_Ehlo	ExistService(DestIP,DestPort) ∧SMTPSupportEhlo(DestIP)	{GainSMTPInfo(SrcIP,DestIP)}
Email_Turn	ExistService(DestIP,DestPort) ∧SMTPSupportTurn(SrcIP, DestIP)	{MailLeakage(DestIP)}
FTP_Pass	ExistService(DestIP,DestPort)	
FTP_Put	ExistService(DestIP,DestPort) ∧GainAccess(DestIP)	{SystemCompromised(DestIP)}
FTP_Syst	ExistService(DestIP,DestPort)	{GainOSInfo(DestIP)}
FTP_User	ExistService(DestIP,DestPort)	
HTTP_ActiveX	ActiveXEnabledBrowser(SrcIP)	{SystemCompromised(SrcIP)}
HTTP_Cisco_Catalyst_Exec	CiscoCatalyst3500XL(DestIP)	{GainAccess(DestIP)}
HTTP_Java	JavaEnabledBrowser(SrcIP)	{SystemCompromised(SrcIP)}
HTTP_Shells	VulnerableCGIBin(DestIP) ∧OSUNIX(DestIP)	{GainAccess(DestIP)}
Mstream_Zombie	SystemCompromised(DestIP) ∧SystemCompromised(SrcIP)	{ReadyForDDOSAttack(SrcIP), ReadyForDDOSAttack(DestIP)}
Port_Scan		{ExistService(DestIP,DestPort)}
RIPAdd		
RIPExpire		
Rsh	GainAccess(DestIP) ∧GainAccess(SrcIP)	{SystemCompromised(DestIP), SystemCompromised(SrcIP)}
Sadmind_Amslverify_ Overflow	VulnerableSadmind(DestIP) ∧OSSolaris(DestIP)	{GainAccess(DestIP)}
Sadmind_Ping	OSSolaris(DestIP)	{VulnerableSadmind(DestIP)}
SSH_Detected		
Stream_DoS	ReadyForDDOSAttack	{DDOSAgainst(DestIP)}
TCP_Urgent_Data		{SystemAttacked(DestIP)}
TelnetEnvAll		{SystemAttacked(DestIP)}
TelnetTerminaltype		{GainTerminalType(DestIP)}
TelnetXdisplay		{SystemAttacked(DestIP)}
UDP_Port_Scan		{ExistService(DestIP,DestPort)}

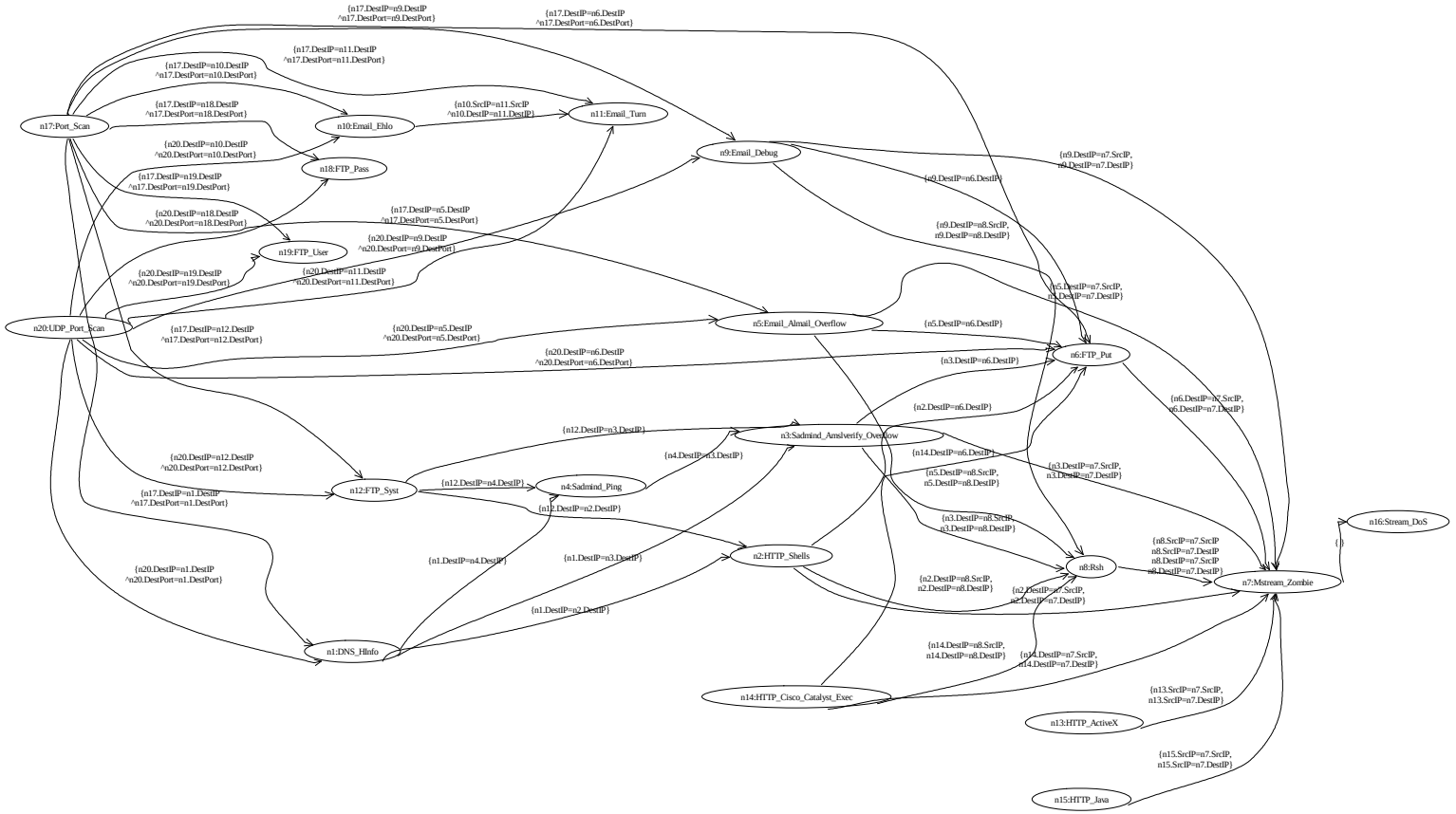


Figure 5.11: The type graph used in our experiments

Table 5.4: Implication relationships between the predicates

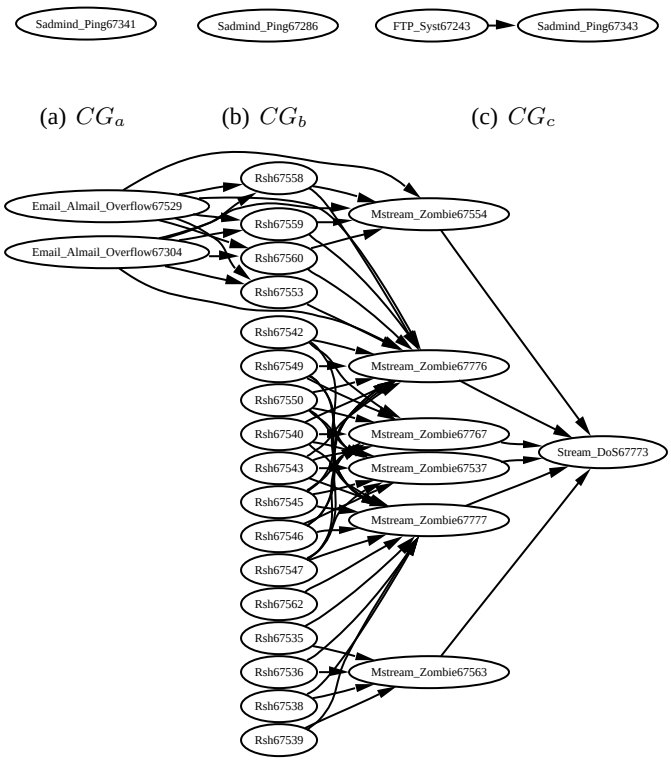
Predicate	Implied Predicate
ExistService(IP,Port)	GainInformation(IP)
GainOSInfo(IP)	GainInformation(IP)
GainOSInfo(IP)	OSSolaris(IP)
OSSolaris(IP)	OSUNIX(IP)
GainSMTPInfo(SrcIP,DestIP)	SMTPSupportTurn(SrcIP,DestIP)
GainAccess(IP)	SystemCompromised(IP)
SystemCompromised(IP)	SystemAttack(IP)
ReadyForDDOSAttack(IP)	ReadyForDDOSAttack

To test the ability of our techniques to hypothesize and reason about missed attacks, we dropped all *Sadmind_Amslverify_Overflow* alerts that RealSecure Network Sensor detected in LLDOS1.0 data set. As a result, the attack scenarios that the Intrusion Alert Correlator output before dropping these alerts are all split into multiple parts, some of which become individual, uncorrelated alerts. In our experiment with inside traffic of LLDOS 1.0 data set, before dropping *Sadmind_Amslverify_Overflow* alerts, we only got one correlation graph. After dropping, however, this correlation graph was divided into four parts. Figure 5.12 shows all these four correlation graphs.

Now let us focus on the correlation graphs in Figure 5.12. What we should do first is to determine if two correlation graphs can be integrated. The second step is to perform hypotheses, inference, validation and consolidation. For the sake of presentation, we first consider integrating two correlation graphs CG_c (Figure 5.12(c)) and CG_d (Figure 5.12(d)).

As mentioned earlier, if two alerts in two different correlation graphs satisfy at least one equality constraint associated with their types, we can combine these correlation graphs together. Since the destination IP addresses of both *Sadmind_Ping67343* (in CG_c) and *Rsh67553* (in CG_d) are $172.16.112.50$, they satisfy the constraint *Sadmind_Ping.DestIP* $=$ *Rsh.DestIP*. Thus, it is easy to see CG_c and CG_d can be integrated together.

Based on the type graph, we can easily hypothesize that variations of *HTTP_Shells*, *FTP_Put* and *Sadmind_Amslverify_Overflow* could have been missed by the IDS sensor. For example, there could be variations of *Sadmind_Amslverify_Overflow* between *Sadmind_Ping* and any later *Rsh* alert. By reasoning about the hypothesized attacks using equality constraints, we can reduce the hypotheses of missed attacks. For example, the destination IP address of *Sadmind_Ping67343* is $172.16.112.50$, which is different from either the source or the destination IP address of *Rsh67543*.



(a) CG_a (b) CG_b (c) CG_c

(d) CG_d

Figure 5.12: Four correlation graphs constructed from LLDOS 1.0 inside traffic

Thus it is easy to see *Sadmind_Ping67343* cannot *indirectly prepare for Rsh67543* through a variation of attack *Sadmind_Amslverify_Overflow*. After missed attack hypotheses and reasoning, we perform attribute value inference. For example, a hypothesized *Sadmind_Amslverify_Overflow* attack between *Sadmind_Ping67343* and *Rsh67553* has the destination IP address $172.16.112.50$.

The hypothesized attacks are further validated using the raw audit data. For example, in our experiments, the filtering condition for (variations of) *FTP_Put* is $protocol = ftp$ plus all the inferable attributes. All the hypothesized attacks are then checked using the extracted values of audit attributes from audit records between the alerts that result in the corresponding hypothesized attacks. For example, we search all the pre-fetched packet information between *Sadmind_Ping67343* and *Rsh67553* for *Sadmind* packets (related to the host $172.16.112.50$) in order to validate a hypothesized (variation of) *Sadmind_Amslverify_Overflow* attack. Finally we can get the integration result

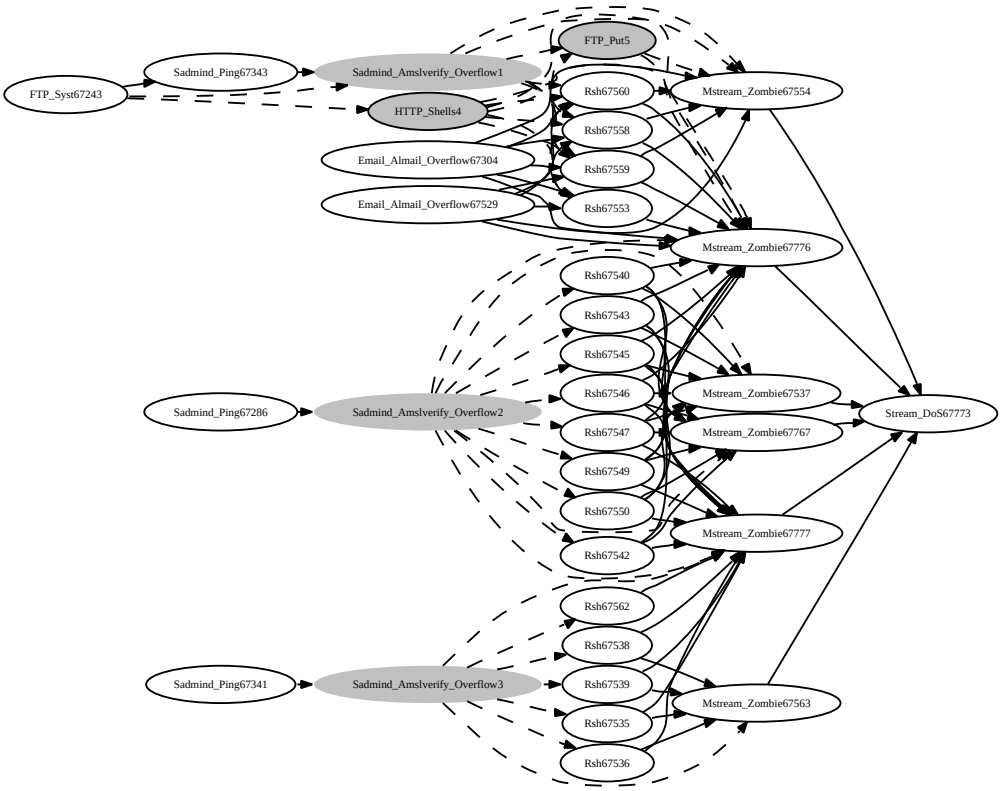
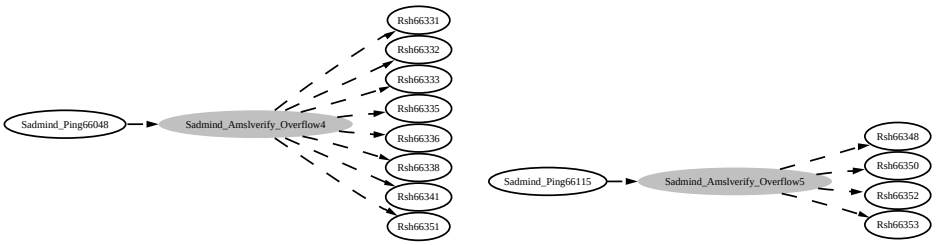


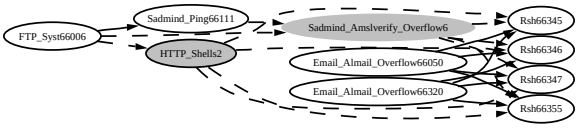
Figure 5.13: The integrated correlation graph constructed from LLDOS 1.0 inside traffic

(without consolidation) on correlation graphs CG_c and CG_d.

We continue the above process to integrate the resulting correlation graph with additional ones (CG_a in Figure 5.12(a) and CG_b in Figure 5.12(b)). The alerts in these two graphs are *Sadmind_Ping67341* and *Sadmind_Ping67286*, respectively, which are both uncorrelated alerts. As a slight difference, several instances of *FTP_Put* are hypothesized during both integration processes, but all of them are invalidated later using the extracted audit information. In other words, we find no *ftp* activities involving the corresponding host during the time frame when the hypothesized attacks might happen. Figure 5.13 shows the integrated correlation graph after the hypothesized attacks are consolidated. The consolidation reduced the number of hypothesized attacks from about 137 to 5. In the integrated correlation graph shown in Figure 5.13, the hypothesized attacks are shown in gray, and are labeled by the corresponding hyper-alert type followed by an ID to distinguish between different instances of the same type of attacks.



(a) Integrated Correlation Graph ICG_a (b) Integrated Correlation Graph ICG_b



(c) Integrated Correlation Graph ICG_c

Figure 5.14: Experimental results using the DMZ dataset in LLDOS 1.0

Now let us examine the integrated correlation graph in Figure 5.13. According to the description of the data sets [77], the three *Sadmind_Amslverify_Overflow* attacks and the *prepare-for* relations between these attacks and the other alerts are hypothesized correctly. However, the *FTP_Put* and *HTTP_Shells* attacks are hypothesized incorrectly.

We also performed the experiments using the DMZ data set in LLDOS 1.0. Similar to the inside data set, we deliberately dropped all *Sadmind_Amslverify_Overflow* alerts from those generated by RealSecure Network Sensor 6.0. Using the type graph in Figure 5.11, we generated three integrated correlation graphs in Figure 5.14, in which hypothesized attacks are shown in gray. Based on the attribute value inference, we know the destination IP addresses of *Sadmind_Amslverify_Overflow4*, *Sadmind_Amslverify_Overflow5* and *Sadmind_Amslverify_Overflow6* are 172.16.115.20, 172.16.112.10 and 172.16.112.50, respectively. Similarly, the destination IP address of *HTTP_Shells2* is 172.16.112.50. According to the description of data sets [77], the *Sadmind_Amslverify_Overflow* attacks are all hypothesized correctly, while the *HTTP_Shells* attack is hypothesized incorrectly. These experiment results (including LLDOS 1.0 inside and DMZ data sets) indicate that though the proposed techniques can identify missed attacks, they are still not perfect. Nevertheless, the proposed techniques have already exceeded the limitation of the underlying IDSs.

5.3 Discussion and Summary

In this chapter, we present a series of techniques to construct high-level attack scenarios to facilitate the analysis of intrusion alerts. Our approach is based on a key concept: equality constraint, which captures the intrinsic relationships between possibly related attacks. Moreover, to reason about hypothesized attacks, we develop techniques to compute equality constraints that indirectly related attacks must satisfy. We propose to further infer attribute values for hypothesized attacks and validate hypothesized attacks through raw audit data. Finally, we present a technique to consolidate hypothesized attacks to generate concise representations of attack scenarios. Our experimental results demonstrate the potential of these techniques.

Though the proposed techniques are aimed at improving IDSs' detection results, the actual performance is still limited by the performance of IDSs. In the worst case, if the IDSs miss all attacks, or all alerts are false ones, the proposed techniques will not perform well. Fortunately, our preliminary experiment has shown some promising results for the current generation of IDSs. We expect the proposed techniques will generate better results as the performance of IDSs is improved.

Our technique is a starting point for improving intrusion detection through alert correlation. There are still a number of problems that are worth additional investigation. One such problem is the granularity in which the attacks are modeled. If the representation of attacks is too specific, the type graph may not be general enough to allow the hypotheses about variations of missed attacks. If the representation is too general, some causal relationships may not be captured in the type graph at all. More research is necessary to understand the best way to model attacks in the proposed framework. Our approach currently is limited because it can only hypothesize intermediate attacks inside an attack scenario. We further notice that our techniques usually require a comprehensive knowledge base about different attacks, which is achievable through studying various attack signatures. Adversaries, aware of our techniques being deployed, may intentionally create scattered attacks. This may bring additional processing overhead, and sometimes may even let us hypothesize "false" attack scenarios (it can be mitigated by our hypothesized attack pruning techniques). These problems worth further investigation.

Chapter 6

Alert Correlation through Triggering Events and Common Resources

As more and more organizations and companies build networked systems to manage their information, network intrusion becomes a serious problem over recent years. At present, these is no single system capable of solving all security concerns. Different types of security systems are deployed into the networks to better protect the digital assets. Figure 6.1 shows an example network deployed with multiple heterogeneous security systems. These systems may comprise firewalls (e.g., ZoneAlarm [117]), intrusion detection systems (IDSs) (e.g., RealSecure Network 10/100 [52], Snort [16] and NIDES [57]), antivirus tools (e.g., Norton AntiVirus [101]), file integrity checkers (e.g., Tripwire [104]), and so forth. They usually serve for different security purposes, or serve for the same purpose through different methods. For example, firewalls focus on accepting, logging, or dropping network traffic, intrusion detection systems (IDSs) focus on detecting known attack patterns (signature based IDSs) or abnormal behaviors (anomaly based IDSs), antivirus tools focus on scanning viruses based on pre-defined virus signatures, and file integrity checkers monitor the activities on file systems such as file addition, deletion and modification.

Although these security systems are complementary to each other, and combining the reports (i.e., alerts) from them can potentially get more comprehensive result about the threats from outside and inside sources, it is still challenging for analysts or analysis tools to analyze these alerts

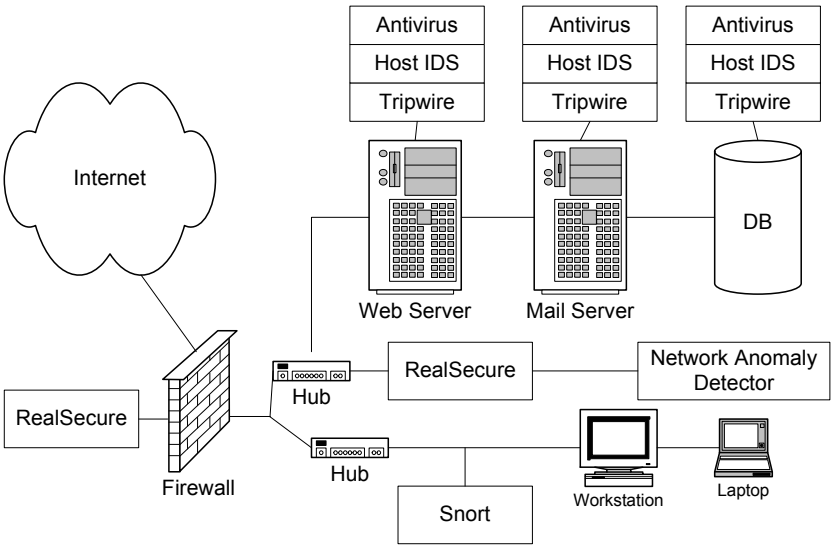


Figure 6.1: A network deployed with multiple heterogeneous security systems

due to the following reasons.

First, a single security system such as a network based IDS may flag thousands of alerts per day [59, 61], and multiple security systems make the situation even worse. Large numbers of alerts may overwhelm the analysts. Second, among a large volume of alerts, a high proportion of them are false positives [61], some of them are low-severity alerts (e.g., an attack to an inactive port), and some of them correspond to real, severe attacks. It is challenging to differentiate these alerts and take appropriate actions. The low level and high volume of the alerts also make extracting the global view of the adversary's attack strategy very challenging. Third, different security systems usually run independently and may flag different alerts for a single attack. Realizing these alerts are actually from the same attack can be time-consuming, though this is critical in assessing the severity of the alerts and the adversary's attack strategy.

To address these challenges, several alert correlation techniques have been proposed in recent years, including approaches based on similarity between alert attributes (e.g, [109, 98, 60, 61, 33, 91, 28]), methods based on pre-defined attack scenarios (e.g., [36, 34, 78]), techniques based on pre-conditions/post-conditions of attacks (e.g., [102, 29, 83, 86]), and approaches using multiple information sources [90, 79]. Though effective at addressing some challenges, none of them dominates the others. Similarity based approaches group alerts based on the similarity between

alert attributes; however, they are not good at discovering steps in a sequence of attacks. Pre-defined attack scenario based approaches work well for known scenarios; however, they cannot discover novel attack scenarios. Pre-condition/post-condition based approaches can discover novel attack scenarios; however, the procedure of specifying pre-conditions and post-conditions are time-consuming and error-prone. Multiple information sources based approaches correlate alerts from multiple information sources such as firewalls and IDSs; however, they are not good at discovering novel attack scenarios.

Our alert correlation techniques proposed in this chapter address some limitations of the current correlation techniques. We propose a novel similarity measure based on triggering events, which helps us to group alerts into clusters such that one cluster may correspond to one attack. We enhance the pre-condition/post-condition based approaches through using input and output resources to facilitate the specification of pre-conditions and post-conditions. Intuitively, the pre-condition of an attack is the necessary condition for the attack to succeed, and the post-condition is the consequence of the attack if the attack does succeed. According, the *input resources* of an attack are the necessary resources for the attack to succeed, and the *output resources* of the attack are the resources that the attack can supply if successful.

Compared with the approaches in [29, 83] which use predicates to describe pre-conditions /post-conditions, our input/output resource based approach has several advantages. (1) When using predicates to specify pre-conditions and post-conditions for each type of attacks, it may introduce too many predicates. Whereas input and output resource types are rather limited compared with the types of predicates and are easy to specify. (2) Since different experts may use different predicates to represent the same condition, or use the same predicate to represent different conditions, it is usually not easy to discover implication relationships between predicates and match post-conditions with pre-conditions especially when the number of predicates is large. Whereas input and output resource types are rather stable, straightforward to match and easy to accommodate new attacks. Our approach also enhances the multiple information sources based approaches in that we provide an input and output resource based method to build attack scenarios.

In this chapter, we propose an alert correlation approach based on triggering events and common resources. Our approach proposes to correlate alerts in three stages. The key concept in the first stage is *triggering events*, which are the (low-level) events observed by security systems that trigger an alert. We observe that although different security systems may flag different alerts for the same attack, the events that trigger these alerts must be the same. For example, for a *RPC sadmind UDP NETMGT_PROC_SERVICE CLIENT_DOMAIN overflow attempt* alert reported by

Snort and an *Sadmind_Amslverify_Overflow* alert reported by RealSecure network sensor, the event that both systems observe is the *malicious sadmind NETMGT_PROC_SERVICE request* between the source and the target hosts. For triggering events, we more focus on low-level events (e.g., a TCP connection). Based on this observation, we find triggering events for each alert, and cluster alerts that share the "similar" triggering events. The alerts in one cluster may correspond to one attack.

In the second stage, we further identify the severity of some alerts and clusters. This is done through examining whether the alerts are *consistent* with their relevant network and host configurations.

In the third stage, we build attack scenarios through input and output resources. We observe that the causal relationships between individual attacks can be discovered through identifying the "common" resources between the output resources of an earlier attack with the input resources of a later one. For example, *Sadmind_Ping* attack can output the status information of sadmind daemon (service resources), where *sadmind* service is necessary to launch *Sadmind_Amslverify_Overflow* attack. Then we can correlate these two attacks. These causal relationships can help us connect alert clusters and build attack scenarios.

6.1 The Model

We present our major techniques in this section. We start by introducing definitions such as alerts, events, configurations and resources in Subsection 6.1.1. Given a set of alerts, we are interested in what events trigger each alert, from which we can put the alerts that share the "similar" triggering events into a cluster. These techniques are presented in Subsection 6.1.2, 6.1.3 and 6.1.4. After alert clustering, we use the information about network and host configurations to examine the alerts in each cluster, which provides us opportunities to identify the severity of some alerts and clusters. This technique is presented in Subsection 6.1.5. The technique on constructing attack scenarios is presented in Subsection 6.1.6, which focuses on discovering causal relationships based on the input and output resources.

6.1.1 Alerts, Events, Configurations and Resources

Different security systems may output alerts in various formats (e.g., in a flat text file, in a relational database, or in a stream of IDMEF [31] messages). We can always extract a set of

attributes (i.e., attribute names and values) associated with the alerts. Events are security related occurrences observed by security systems, configurations encode the information about software and hardware about a host or a network, resources encode the sources an attack may require to use or can possibly supply if it succeeds, and they all can be defined as a set of attributes (attribute names and values). Formally, an *alert type* (or *event type*, or *configuration type*, or *resource type*) is a set S of attribute names, where each attribute name $a_i \in S$ has a domain D_i. A type T *alert* t (or *event* e, or *configuration* c, or *resource* r) is a tuple on attribute name set in T, where each element in the tuple is a value in the domain of the corresponding attribute name. In this paper, for the sake of presentation, we assume each alert and event respectively has at least two attributes: *StartTime* and *EndTime* (if an alert or event only has one timestamp, we assume StartTime and EndTime have the same value). For convenience, we denote the type of alert t, event e and resource r as $Type(t)$, $Type(e)$ and $Type(r)$, respectively. In the following, we may use attributes to denote either attribute names or attribute values or both if it is not necessary to differentiate them.

Here we give a series of examples and discussion related to alert types, alerts, event types, events, configuration types, configurations, resource types, and resources (we may omit the domain of each attribute). As the first example, we define an alert type *Sadmind_Amslverify_Overflow* $=$ {SrcIP, SrcPort, TargetIP, TargetPort, StartTime, EndTime}. A type *Sadmind_Amslverify_Overflow* alert t = {SrcIP = $10.10.1.10$, SrcPort = 683, TargetIP = $10.10.1.1$, TargetPort = 32773, StartTime = 03-07-2004 18:10:21, EndTime = 03-07-2004 18:10:21} describes an *Sadmind_Amslverify_Overflow* alert from IP address $10.10.1.10$ to IP address $10.10.1.1$.

Secondly, we define an event type *malicious sadmind NETMGT_PROC_SERVICE Request* = {SrcIP, SrcPort, TargetIP, TargetPort, StartTime, EndTime} and a type *malicious sadmind NETMGT_PROC_SERVICE Request* event e = {SrcIP = $10.10.1.10$, SrcPort = 683, TargetIP = $10.10.1.1$, TargetPort = 32773, StartTime = 03-07-2004 18:10:21, EndTime = 03-07-2004 18:10:21}. Though high-level events are possible, in this paper we are more interested in low-level events. For example, a TCP connection exploiting the vulnerability in a ftp server, a read operation on a protected file, and so forth. These low-level events may trigger the security systems to flag alerts. For example, a ftp connection including some special data such as "~^$.—*+()[]{}" [53] in its payload may trigger a *FTP_Glob_Expansion* alert by a network based IDS, and may trigger a *NEW_CLIENT* alert by an anomaly detector. The *malicious sadmind NETMGT_PROC_SERVICE Request* event may trigger *Sadmind_Amslverify_Overflow* alert if it is captured by a RealSecure network sensor.

As the third example, we define a configuration type *HostFTPConfig* = {HostIP, Host-

Name, OS, FTPSoftware, FTPOpenPort}, and a type *HostFTPConfig* configuration c = {HostIP = 10.10.1.7, HostName = foo, OS = Solaris 8, FTPSoftware = FTP Server, FTPOpenPort = 21}. We are particularly interested in the critical software which may have vulnerabilities, for example, a ftp server program and its open port in a host. We further classify the configurations into two categories: host configuration and network configuration. The aforementioned *HostFTPConfig* is a host configuration listing the ftp software and open port. Network configurations specify the setting about the whole network. For example, the access control list (ACL) in a firewall, which controls the inbound and outbound traffic for the whole network. A type *NetTrafficControlConfig* configuration c' = {Source = any, Destination = 10.10.1.8, DestPort = 80, Protocol = tcp, Action = accept} is an example of network configuration controlling the inbound traffic to 10.10.1.8 at TCP port 80.

As the last example, we define a resource type *file* = {HostIP, Path}, a resource type *network_service* = {HostIP, HostPort, ServiceName}, and a resource type *privilege* = {HostIP, Access}. A type *file* resource r_1 = {HostIP = 10.10.1.9, Path = /home/Bob/doc/info.txt}, a type *network_service* resource r_2 = {HostIP = 10.10.1.9, HostPort = 21, ServiceName = ftp}, and a type *privilege* resource r_3 = {HostIP = 10.10.1.9, Access = AdminAccess}.

6.1.2 Triggering Events for Alerts

As we mentioned, a single event may trigger different alerts for different security systems. Since security systems may not necessarily tell the analysts what events trigger an alert, it is usually necessary to discover the *triggering events* for alerts. *Triggering events* are the events that trigger the alert. Given an alert, we are interested in its triggering events. More specifically, we are interested in the set of event types which *may trigger* an alert type, and the attribute values for each triggering event. Domain knowledge is essential for the discovery of triggering events.

Definition 17 *Given an alert type T_t, the* set of triggering event types *for T_t is a set $\mathcal{T}$ of event types, where for each event type $T_e \in \mathcal{T}$, there is an attribute mapping function f that maps attribute names in T_t to attribute names in T_e. Given a type T_t alert t, the* triggering event set *for t is a set E of events, where for each $T_e \in \mathcal{T}$, there is a type T_e event $e \in E$, and the attribute values of e are instantiated by the attribute values in t through the corresponding attribute mapping function.*

Let us look at an example. Given an alert type T_t = *Sadmind_Amslverify_Overflow*, the set of triggering event types is $\{T_e\}$, where T_e = *Malicious sadmind NETMGT_PROC_SERVICE Request*, and the attribute mapping function f has $f(T_t$.SrcIP) $= T_e$.SrcIP, $f(T_t$.SrcPort) $= T_e$.SrcPort, $f(T_t$.TargetIP) $= T_e$.TargetIP, $f(T_t$.TargetPort) $= T_e$.TargetPort, $f(T_t$.StartTime) $= T_e$.StartTime and $f(T_t$.EndTime) $= T_e$.EndTime. Given a type *Sadmind_Amslverify_Overflow* alert $t = \{$SrcIP = $10.10.1.10$, SrcPort = 683, TargetIP = $10.10.1.1$, TargetPort = 32773, StartTime = 03-07-2004 18:10:21, EndTime = 03-07-2004 18:10:21$\}$, we know the triggering event set has one type *Malicious NETMGT_PROC_SERVICE Request* event $e = \{$SrcIP = $10.10.1.10$, SrcPort = 683, TargetIP = $10.10.1.1$, TargetPort = 32773, StartTime = 03-07-2004 18:10:21, EndTime = 03-07-2004 18:10:21$\}$.

Triggering events provide us an opportunity to find different alerts that may correspond to the same attack. Given a set of alerts, first we can discover the triggering event set for each alert, then we can put individual alerts into clusters if the alerts in the same cluster share the "similar" triggering events. The alerts in the same cluster may correspond to the same attack. In the following, we may simply use the term events instead of triggering events if it is clear from the context. We first discuss the event inference, then define "similarity" between events through which our clustering algorithm is further introduced.

6.1.3 Inference between Events

Intuitively, two events are "similar" if they have the same event type, and their attribute names and values are also the same. However, considering the existence of implication relationships between events (the occurrence of one event implies the occurrence of another event), we realize that the concept of "similarity" can be extended beyond this intuition to accommodate event implication.

We first give examples to illustrate the implication relationships. Consider two events: the recursive deletion of directory "/home/Bob/doc" and the deletion of file "/home/Bob/doc/info.txt". The first event can imply the second one because "info.txt" is one of the files in that directory. As another example, we know an event type *restricted_file_write may imply* an event type *filesystem_integrity_violation*. On the other hand, a recursive directory deletion does not necessary imply a file deletion if the file is not in the same directory. For example, the recursive deletion of directory "/home/Bob/doc" cannot imply the deletion of file "/home/Alice/doc/info.txt". This observation tells us when we introduce the implication relationships between events, we not only need to examine the semantics of event types, but also the relationships between attribute names and values.

We use *may-imply* to refer to the implication between event types, and use *imply* to refer to the implication between events (including types and their related attributes names and values). For convenience, we denote event e_1 implies event e_2 as $e_1 \rightarrow e_2$.

We introduce a binary *specific-general* relation to help us identify implication relationships. Formally, given two concepts (e.g., two event types, two attribute names, etc) a_1 and a_2, a *specific-general* relation between a_1 and a_2 maps low-level (specific) concept a_1 to high-level (general) concept a_2, and is denoted as $a_1 \preceq a_2$ (for convenience, we may also refer to specific-general relation as "$\preceq$" relation in this chapter). Specific-general relation is reflexive (we have $a \preceq a$ for any concept a), antisymmetric and transitive, and it essentially is a partial order over a set of concepts (which is modeled as *concept hierarchy* in data mining [49, 71]).

Specific-general relations can be applied to event types. For example, we can define *file_deletion* $\preceq$ *recursive_directory_deletion* and *restricted_file_write* $\preceq$ *filesystem_integrity_violation*. Here domain knowledge is necessary to determine whether *EventType1* may imply *EventType2* or *EventType2* may imply *EventType1* even if *EventType1* $\preceq$ *EventType2* or *EventType2* $\preceq$ *EventType1* is decided. For example, it is straightforward for an expert to decide *recursive_directory_deletion* may imply *file_deletion* and *restricted_file_write* may imply *filesystem_integrity_violation*. Our following job is to decide the relationships between attribute names and values.

Again, the relationships between attributes are determined through specific-general relations. As an example, for specific-general relations between attribute names, we can define $file \preceq directory$ and $host \preceq network$. In addition, we are interested in whether "$\preceq$" relation is satisfied once the attribute names are replaced by their values. For example, under $file \preceq directory$ relation, we have "/home/Bob/doc/info.txt" $\preceq$ "/home/Bob/doc", and under $host \preceq network$ relation, we have $10.10.1.10 \preceq 10.10.1.0/24$. In the following, when referring to "$\preceq$" relations, we may not distinguish between attribute names and values if it is not necessary.

It is worth mentioning that as a special case, timestamp attributes (StartTime and EndTime) have different characteristics compared with other attributes in that even if two triggering events actually refer to a same event, they may not have the exactly same timestamps due to the clock discrepancy in different systems or event propagation over the network. Thus we propose to use *temporal constraint* to evaluate whether a set of events are "similar" w.r.t. timestamps.

Definition 18 *Consider a set E of events and a time interval λ. E satisfies the* temporal constraint *λ if and only if for any $e, e' \in E$ ($e \neq e'$), $|e.StartTime - e'.StartTime| \leq \lambda$ and $|e.EndTime -$*

Algorithm: Determining if event e_1 implies event e_2
Input: Two events e_1 and e_2 and a temporal constraint λ.
Output: *True* if $e_1 \rightarrow e_2$; otherwise *false*.
Method:
Assume the attribute name sets for e_1 and e_2 are A_1 and A_2, respectively. Initialize *result=false*.
1. **If** $Type(e_1)$ may imply $Type(e_2)$
2. **If** $Type(e_1) \preceq Type(e_2)$
3. Find a mapping such that $\forall a_1 \in A_1'(A_1' \subseteq A_1)$ and $\forall a_2 \in A_2'(A_2' \subseteq A_2)$, we have $a_1 \preceq a_2$
4. **Else** Find a mapping such that $\forall a_2 \in A_2'(A_2' \subseteq A_2)$ and $\forall a_1 \in A_1'(A_1' \subseteq A_1)$, we have $a_2 \preceq a_1$
5. Replace names with values for all "$\preceq$" relations.
6. **If** all "$\preceq$" relations are satisfied in step 5
7. **If** e_1 and e_2 satisfy constraint λ, Let *result=true*
8. **Output** *result*.
End

Figure 6.2: An algorithm to discover implication relationship between events.

$e'.EndTime| \leq \lambda$.

Based on "$\preceq$" relations and a temporal constraint λ, we outline an algorithm (shown in Figure 6.2) to determine whether event e_1 implies event e_2. The basic idea is that we first identify whether $Type(e_1)$ may imply $Type(e_2)$. If this is the case, we further check "$\preceq$" relations between attribute names and values, and examine the temporal constraint to see whether e_1 implies e_2.

6.1.4 Clustering Alerts Using Triggering Events

Intuitively, we intend to group individual alerts into clusters such that all alerts in the same cluster either share the same triggering events, or their triggering events have implication relationships. To formalize this intuition, we first define *similarity* between alerts.

Definition 19 *Consider a set S of alerts $\{t_1, t_2, \cdots, t_n\}$ and a temporal constraint λ. Assume the triggering event sets for $t_1, t_2, \cdots, t_n$ are $E_1, E_2, \cdots, E_n$, respectively. All alerts in S are* similar *if and only if there exist $e_1 \in E_1$, $e_2 \in E_2$, $\cdots$, $e_n \in E_n$ such that for any two events e_i and e_j in*

$\{e_1, e_2, \cdots, e_n\}$, *we have* $e_i \rightarrow e_j$ *or* $e_j \rightarrow e_i$.

The idea behind Definition 19 can be demonstrated through an example with two alerts. Two alerts are similar if their triggering events either have the same event type, attribute names and values, and their timestamps satisfy the given temporal constraint, or their triggering events have implication relationship from one to the other. Since two events e_1 and e_2 of the same type have all attributes the same is a special case of $e_1 \rightarrow e_2$ or $e_2 \rightarrow e_1$, we can combine these two cases and only use implication relationships to define similarity.

Given a set S_u of alerts, we can perform clustering based on the similarity defined in Definition 19. Intuitively, we can iteratively pick a subset of alerts from S_u such that all the alerts in this subset are similar. However, we have to solve a problem before we can apply this operation. This problem can be demonstrated by an example. Suppose we have three alerts t_1, t_2 and t_3, they are of the same type and have the same attribute names and values except for timestamp values, and their triggering event sets are $\{e_1\}$, $\{e_2\}$, and $\{e_3\}$, respectively. Assume the temporal constraint $\lambda = 1$ second, $e_1.StartTime = e_1.EndTime$ = 03-07-2004 18:20:21, $e_2.StartTime = e_2.EndTime$ = 03-07-2004 18:20:22, and $e_3.StartTime = e_3.EndTime$ = 03-07-2004 18:20:23. Based on Definition 19, t_1 and t_2 are similar, and t_2 and t_3 are similar. Thus t_2 can be either put into a cluster with t_1, or be put into a cluster with t_3. To solve this ambiguity, we apply a rule "earlier timestamp first", where the cluster with the earlier (StartTime) alerts will get alerts as many as possible. Applying this rule to the example, we will let t_1 and t_2 be in the same cluster. The algorithm shown in Figure 6.3 outlines the alert clustering through apply this rule. In this algorithm, line 1 prepares the alert set and initializes some variables. Lines 2 through 6 are a loop, which always looks for the alerts that are similar to the first alert in the alert set and puts them into a cluster. This loop will not stop until there are no alerts in the alert set. Line 7 finally outputs all clusters.

An interesting observation about alert clustering is that after alerts are put into clusters, we may mark some clusters with low severity through examining whether the alerts are consistent with relevant configurations. This will be further discussed in Subsection 6.1.5.

6.1.5 Consistency and Inconsistency between Alerts and Relevant Configurations

Host and network configurations provide us an opportunity to verify the consistency or discover the inconsistency between alerts and their relevant configurations. The consistency between an alert and its related configurations can be verified through examining the attributes of

Algorithm: Alert Clustering via Triggering Events
Input: A set S_u of alerts and a temporal constraint λ.
Output: A set C of clusters.
Method:

1. Sort the set S_u of alerts ascendingly on StartTime, and name it S_o. Initialize $C = \emptyset$, and let $i = 1$.
2. **While** S_o is not empty
3. Let the alert with the earliest StartTime in S_o be t.
4. Find set $S' \subseteq S_o$ such that $\{t\} \cup S'$ are similar.
5. Remove $\{t\} \cup S'$ from S_o into a set C_i.
6. Put C_i into C. Let $i = i + 1$.
7. **Output** C.

End

Figure 6.3: An algorithm to perform alert clustering based on triggering events.

the alert and the configurations. For example, consider an *FTP_Glob_Expansion* alert $t = \{$SrcIP = 172.16.1.7, SrcPort = 1042, TargetIP = 10.10.1.7, TargetPort = 21, StartTime = 03-07-2004 18:20:21, EndTime = 03-07-2004 18:20:21$\}$ and a *HostFTPConfig* configuration $c = \{$HostIP = 10.10.1.7, HostName = foo, OS = Solaris 8, FTPSoftware = FTP Server, FTPOpenPort = 21$\}$. Alert t is consistent with configuration c because it exploits a host 10.10.1.7 at port 21 which is an open port listed in this host's configuration. We formalize this relationship in Definition 20.

Definition 20 *Consider an alert type T_t and a configuration type T_c. A* consistent condition *for T_t w.r.t. T_c is a logical formula including attribute names in T_t and T_c. Given a type T_t alert t and a type T_c configuration c, t* is consistent (or inconsistent, resp.) with *c if the formula is evaluated to* True (or False, resp.) *where attribute names in the formula are replaced with the values in t and c.*

Let us look at an example. Given an alert type *FTP_Glob_Expansion* (T_t) and a configuration type *HostFTPConfig* (T_c), we define $T_t.TargetIP = T_c.HostIP \wedge T_t.TargetPort = T_c.FTPOpenPort$ as a consistent condition for T_t w.r.t. T_c. Given an *FTP_Glob_Expansion* alert $t = \{$SrcIP = 172.16.1.7, SrcPort = 1042, TargetIP = 10.10.1.7, TargetPort = 21, StartTime = 03-07-2004 18:20:21, EndTime = 03-07-2004 18:20:21$\}$ and a *HostFTPConfig* configuration $c = \{$HostIP = 10.10.1.7, HostName = foo, OS = Solaris 8, FTPSoftware = FTP Server, FTPOpenPort = 21$\}$,

the consistent condition is evaluated to *True* using attribute values in t and c. Then we know t is consistent with c.

Consistent and inconsistent relationships between alerts and configurations provide us a way to classify the alerts. We can mark each alert as consistent or inconsistent with the related configurations. A consistent alert tells us the corresponding attack could be possible due to the potential vulnerabilities in the configuration. A special case worth mentioning is that sometimes a consistent alert is a low-severity alert. For example, if a firewall reports a *FWROUTE* alert saying that an inbound packet is blocked, which is consistent with the ACL configuration of the firewall, this alert is less severe because the corresponding connection is blocked. On the other hand, an inconsistent alert may be of low severity because the corresponding attack could not succeed (e.g., an adversary tries to connect to a port which is not open). A special case is that a configuration could be compromised (e.g., an adversary installs malicious programs and opens new ports) without the notice of the legitimate users, then the corresponding attack may succeed. In this case, the "inconsistent" alert (which actually is not an inconsistent alert because the configuration is changed) deserves more investigation.

We can apply consistency and inconsistency relationships to alert clusters to determine the severity of some clusters. For example, assume a *FWROUTE* alert (reported by a firewall denoting that a connection is blocked) and a *NEW_CLIENT* alert (reported by a network anomaly detector denoting that a new client requests a server) are in the same cluster, and *FWROUTE* is consistent with its configuration. Since *FWROUTE* denies the requested connection, the related attack cannot be successful and this cluster is less severe. Then we could put more efforts on investigating other possibly severe clusters.

6.1.6 Attack Scenario Construction based on Input and Output Resources

Our approach further determines the causal relationships between alert clusters. We are interested in how individual attacks (represented by alert clusters) are combined to achieve the adversary's goal. The observation tells us that in a sequence of attacks, some attacks have to be performed earlier in order to launch later attacks. For example, an adversary always installs DDoS software before actually launching DDoS attacks. If we are able to capture these causal relationships, it may help us build stepwise attack scenarios and reveal the adversary's attack strategy.

Our approach to modeling causal relationships between attacks is partially inspired by the prerequisites and consequences based alert correlation techniques [102, 29, 83]. However, since

we use resources to specify prerequisites and consequences, compared with the predicates based approach adopted by [29, 83], we have several advantages such as it is easy to specify and (partially) match input and output resources, and easy to accommodate new attacks. Our approach is based on our observation that the causal relationships between attacks can be captured through examining output resources of one attack with input resources of another. Informally, input resources are the necessary resources for an attack to succeed, and output resources are the resources an attack can supply if successful.

We extend our model for alerts (or alert types, resp.) to accommodate input and out resources (or input and output resource types, resp.). We call them *extended alerts* (or *extended alert types*, resp.) after this extension. Considering the resource attribute names may not always the same as the alert attribute names, we further use functions to map the alert attributes to resource attributes. In the following, we formalize extended alert types and extended alerts.

Definition 21 *An* extended alert type T *is a triple (*$\mathcal{T}_i$, attr_names, $\mathcal{T}_o$*), where (1)* attr_names *is a set of attribute names (including StartTime and EndTime) where each attribute name* a_j *has a domain* D_j*, (2)* $\mathcal{T}_i$ *and* $\mathcal{T}_o$ *are a set of resource types, respectively, and (3) for each* $T_i \in \mathcal{T}_i$ *and* $T_o \in \mathcal{T}_o$*, there exist attribute mapping functions* f_i *and* f_o *that map attribute names in* attr_names *to attribute names in* T_i *and* T_o*, respectively.*

A type T *(*$T =$ *(*$\mathcal{T}_i$, attr_names, $\mathcal{T}_o$*))* extended alert t *is a triple (*input, attributes, output*), where (1)* attributes *is a tuple on attr_names, (2)* input *and* output *are a set of resources, respectively, and (3) for each* $T_i \in \mathcal{T}_i$ *and* $T_o \in \mathcal{T}_o$*, there exist resources* $r_i \in$ input *and* $r_o \in$ output*, respectively, where their attribute values are instantiated by the corresponding attribute values in* attributes *through attribute mapping functions.*

Actually, *attr_names* in an extend alert type is an alert type, and *attributes* in an extended alert is an alert that we defined in Subsection 6.1.1. In the remaining part of this paper, we may simply use alert types (or alerts, resp.) when it is not necessary to differentiate extend alert types and alert types (or extended alerts and alerts, resp.).

Example 10 *Define an* Sadmind_Amslverify_Overflow *(T) extend alert type as* { {*network_service*} , {*SrcIP, SrcPort, TargetIP, TargetPort, StartTime, EndTime*}, {*privilege*} }, *where network_service (T_i) =* {*HostIP, HostPort, ServiceName*} *and privilege (T_o) =* {*HostIP, Access*}. *For attribute mapping, we have* $f_i(T.TargetIP) = T_i.HostIP$, $f_i(T.TargetPort) = T_i.HostPort$, *and* $f_o(T.TargetIP) = T_o.HostIP$.

Given a type Sadmind_Amslverify_Overflow *alert* {*SrcIP* = 10.10.1.10, *SrcPort* = 683, *TargetIP* = 10.10.1.1, *TargetPort* = 32773, *StartTime = 03-07-2004 18:10:21, EndTime = 03-07-2004 18:10:21*}, *we can get their input and output resources as* input = { {*HostIP* = 10.10.1.1, *HostPort = 32773, ServiceName = sadmind*} }, *and* output = { {*HostIP* = 10.10.1.1, *Access = AdminAccess*}}. *These three parts combined together are an extended alert.*

Please note in Definition 21, when performing attribute mapping from *attr_names* to $T_i \in \mathcal{T}_i$ and $T_o \in \mathcal{T}_o$, based on domain knowledge, we can mark some attributes in T_i and T_o as special attributes, where they have pre-determined values once the attribute values of resources in *input* and *output* are instantiated. For example, as shown in Example 10, *Access* attribute in *privilege* resource has a pre-determined *AdminAccess* value.

Similar to the implication relationship between events, one resource r_1 can imply another resource r_2 (we use $r_1 \rightarrow r_2$ to represent r_1 implies r_2). For example, a *privilege* resource {HostIP = 10.10.1.9, Access = AdminAccess} implies another *privilege* resource {HostIP = 10.10.1.9, Access = UserAccess}. Please note two resources r_1 and r_2 have their types, attribute names and values all the same is a special case of $r_1 \rightarrow r_2$ or $r_2 \rightarrow r_1$. The implication relationships between resources can be determined through specific-general relations and a similar procedure we described in Subsection 6.1.3 (The difference is that in this chapter we do not associate resources with timestamps). We do not repeat it here.

We can identify causal relationships between attacks through discovering "common" resources between input and output resources. Intuitively, if one attack's output resources include one resource in another attack's input resources, we can correlate these two attacks together. We formalize this intuition as follows.

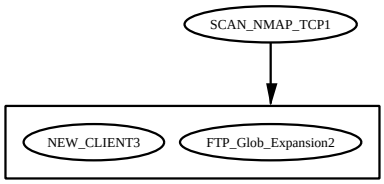


Figure 6.4: An example scenario graph

Definition 22 *Given two extended alerts* $t =$*(*input, attributes, output*) and* $t' =$ *(*input', attributes', output'*),* t causally correlates t' *if there exist* $r_o \in$ output *and* $r'_i \in$ input' *such that* r_o *implies* r'_i *and* $t.EndTime < t'.StartTime$*.*

Let us consider two alerts t (type *SCAN_NMAP_TCP*) and t' (type *FTP_Glob_Expansion*). Suppose the output resource of t is a *network_service* resource {HostIP = 10.10.1.7, HostPort = 21, ServiceName = ftp}, the input resource of t' is a *network_service* resource {HostIP = 10.10.1.7, HostPort = 21, ServiceName = ftp}, and $t.EndTime < t'.StartTime$. Since the output resource of t and the input resource of t' are exactly the same, we can conclude that t causally correlates t'.

We also refer to "causally-correlate" relations introduced in Definition 22 as causal relations, which provide us opportunities to build attack scenarios. Consider a set of alerts reported by different security systems. We can group alerts into clusters using triggering events. Each cluster may correspond to one attack. Through discovering causal relations between alerts in different clusters, we can naturally connect different clusters and construct the attack scenarios. Definition 23 further formalizes this intuition.

Definition 23 *Consider a set* C *of clusters where each cluster is a set* C_i *of alerts. A* scenario graph $SG = (V, A)$ *is a directed acyclic graph, where (1)* V *is the vertex set, and* A *is the edge set, (2) each vertex* $v \in V$ *is a cluster in* C*, and (3) there is an edge* $(v_1, v_2) \in A$ *if and only if there exist* $t_1 \in v_1$ *and* $t_2 \in v_2$ *such that* t_1 causally correlates t_2.

Here we show an example of scenario graph in Figure 6.4. The string inside each node is the alert type followed by an ID (we will follow this convention in our experiments). This scenario has two clusters: $C_1 =$*{SCAN_NMAP_TCP1}* and $C_2 =$*{NEW_CLIENT3, FTP_Glob_Expansion2}*,

where *SCAN_NMAP_TCP1* is reported by Snort, *FTP_Glob_Expansion2* is reported by a RealSecure network sensor, and *NEW_CLIENT3* is reported by a network anomaly detector. Assume *SCAN_NMAP_TCP1* in C_1 causally correlates *FTP_Glob_Expansion2* in C_2, then we can correlate C_1 and C_2 together as shown in Figure 6.4. Such graph clearly discloses an adversary's attack strategy.

6.2 Experimental Results

To evaluate the effectiveness of our techniques, we performed experiments through DARPA Cyber Panel Program Grand Challenge Problem Release 3.2 (GCP) [35, 48], which is an attack scenario simulator. GCP simulator can simulate the behavior of sensors and generate alert streams. There are totally 10 types of sensors in the networks. All the sensors generate the alerts in IDMEF [31] messages.

The current implementation is a proof-of-concept system. In our implementation, we use Java as programming language, and Microsoft SQL Server 2000 as the DBMS to save the alert data set and domain knowledge. Database access is through JDBC. The alert process in our system can be divided into four stages. In the first stage, we concentrate on data preparation. Since all alert data generated by GCP simulator are IDMEF messages, we extract the attributes from these messages and put them into the database. All the necessary domain knowledge related to triggering event types, event inference, input and output resource types, and so forth are all put into the database. The second stage is the alert clustering stage. We group alerts into different clusters based on the algorithm shown in Figure 6.3. The third stage is to examine the consistency or inconsistency between alerts and the configurations. In the last stage, we use input and output resource based correlation techniques to discover causal relationships and build attack scenarios. To save our development effort, we use GraphViz [9] to draw scenario graphs.

The experiments were performed using Attack 1 scenario in GCP attack simulator. We chose 4 network enclaves, namely HQ enclave, APC enclave, Ship enclave and ATH enclave, to play this scenario. Attack 1 is a (agent-based) worm related attack. After the agent being activated, it performs a series of malicious actions such as communicating with an external host, getting malicious code and instructions, spreading from one network enclave to another, compromising hosts in the network enclaves, sniffing the network traffic, reading and modifying the sensitive files, sending the sensitive data to the external host, getting new malicious instructions, and so forth. For this

Table 6.1: Triggering event types for each alert type.

Alert Type	Triggering Event Types	Attributes (besides StartTime and EndTime)
FTP_Globbing_Attack	{ConnectionAttempt, MaliciousFTPRequest}	SrcIP, SrcPort, TargetIP, TargetPort, Protocol
FWROUTE	{ConnectionAttempt, ACLViolation}	SrcIP, SrcPort, TargetIP, TargetPort, Protocol
Loki	{ConnectionAttempt, DataSecretTransmission}	SrcIP, SrcPort, TargetIP, TargetPort, Protocol
NEW_CLIENT	{ConnectionAttempt}	SrcIP, SrcPort, TargetIP, TargetPort, Protocol
Network Interface In Promiscuous Mode	{Network Interface In Promiscuous Mode}	SensorIP, SrcUserID, SrcProcessName, TargetProcessPath
ASSET-DEAD	{ASSET-DEAD}	TargetIP
ASSET-SICK	{ASSET-SICK}	TargetIP
ASSET-WELL	{ASSET-WELL}	TargetIP
dbschema-downloaded	{dbschema-downloaded}	SensorIP
filesystem-integrity	{filesystem-integrity}	SensorIP, TargetFilePath
registry-integrity	{registry-integrity}	SensorIP, TargetFilePath
restricted_read	{restricted_read}	SensorIP, SrcUserID, SrcProcessName, TargetProcessPath
Restricted_System_File_Scan	{Restricted_System_File_Scan}	SensorIP, SrcUserID, SrcProcessName, TargetProcessPath
restricted_write	{restricted_write}	SensorIP, SrcUserID, SrcProcessName, TargetProcessPath
RootShareMounted	{ConnectionAttempt, AdminShareAccess}	SrcIP, SrcPort, TargetIP, TargetPort, Protocol
ServiceUnavailable	{ConnectionAttempt}	SrcIP, SrcPort, TargetIP, TargetPort, Protocol

scenario, we totally got 529 alerts with 16 different types.

Our first goal is to evaluate the effectiveness of alert clustering proposed in this chapter. We list the set of triggering event types for each alert type in Table 6.1. In addition to triggering event types for each alert type, we also define the implication relationships between event types. We define *restricted_write* may imply *filesystem-integrity*, and the related specific-general relations are *restricted_write.SensorIP* $\preceq$ *filesystem-integrity.SensorIP* and *restricted_write.TargetProcessPath* $\preceq$ *filesystem-integrity.TargetFilePath*. We set the temporal constraint $\lambda = 1$ second.

Totally we get 512 clusters from alert clustering. Among them there are 17 clusters, each

Table 6.2: All 2-alert clusters.

Cluster ID	Alerts
2	NEW_CLIENT10, FWROUTE7
4	NEW_CLIENT25, FWROUTE27
54	NEW_CLIENT132, FTP_Globbing_Attack135
102	NEW_CLIENT6, FWROUTE5
116	NEW_CLIENT124, ServiceUnavailable125
132	NEW_CLIENT33, FWROUTE21
136	NEW_CLIENT122, ServiceUnavailable121
184	NEW_CLIENT24, FWROUTE34
236	NEW_CLIENT32, FWROUTE20
238	NEW_CLIENT49, FWROUTE57
242	NEW_CLIENT153, FTP_Globbing_Attack154
281	NEW_CLIENT29, FWROUTE26
333	NEW_CLIENT8, FWROUTE12
335	NEW_CLIENT30, FWROUTE39
340	NEW_CLIENT54, FWROUTE53
342	NEW_CLIENT50, FWROUTE56
385	NEW_CLIENT134, FTP_Globbing_Attack133

of them comprises 2 alerts, and all other clusters are single-alert clusters. Table 6.2 lists all 2-alert clusters. From Table 6.2, we observe every cluster has a *NEW_CLIENT* alert, which is reported by network anomaly sensors denoting a new client requests a server (service). This is normal because the connection requests trigger these alerts. Both alerts in each cluster in Table 6.2 actually refer to the same network connection, which trigger different alerts for different systems.

Our next goal is to evaluate the effectiveness of consistent conditions in identifying the severity of some alerts and clusters. Among all alerts, we find 4 alerts inconsistent with their configurations. These 4 alerts are *NEW_CLIENT122*, *NEW_CLIENT124*, *ServiceUnavailable121* and *ServiceUnavailable125*. *NEW_CLIENT122* and *ServiceUnavailable121* target at port 111 on host 10.1.2.2, and *NEW_CLIENT124* and *ServiceUnavailable125* target at port 21 on host 10.1.2.2. They are also in Table 6.2 (Cluster ID = 136 and 116), which means these 4 alerts actually represent two attacks. Our investigation shows that both attacks are failed attempts (one is through sadmind exploit, and the other is through ftp globbing exploit) because the ports 111 and 21 are not open at host 10.1.2.2.

We also investigate the 2-alert clusters where one alert in the cluster is *FWROUTE*. In Table 6.2, there are 12 clusters that include *FWROUTE* alerts. These *FWROUTE* alerts are consistent

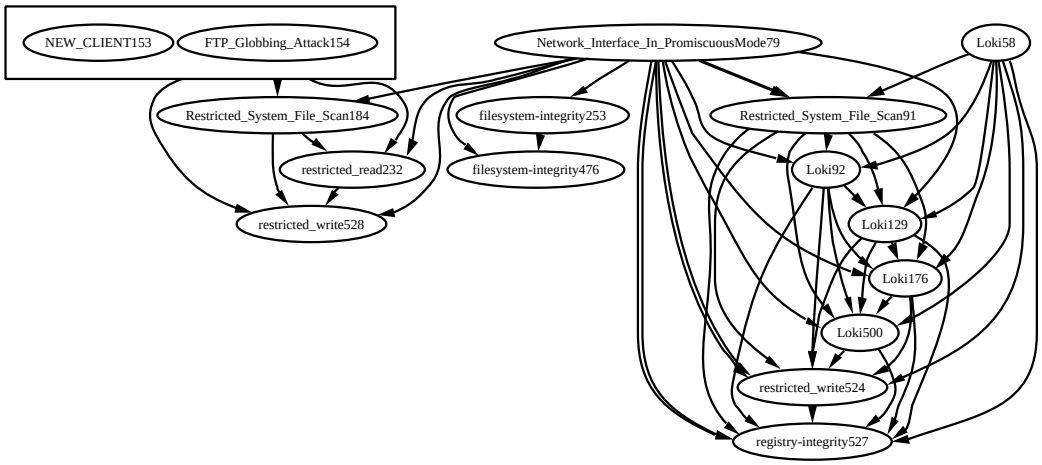


Figure 6.5: One Scenario Graph in HQ Enclave

Table 6.3: Resource types in the experiments.

Resource Type	Attributes
Privilege	HostIP, Access
ResourceUnavailable	TargetIP
NetworkSensitiveInfo	NetworkID
DBInfo	HostIP
ConnectionUnavailable	SrcIP, SrcPort, TargetIP, TargetPort, Protocol
Connection	SrcIP, SrcPort, TargetIP, TargetPort, Protocol
NetworkService	TargetIP, TargetPort, Protocol
HostSensitiveInfo	HostIP
DirectAccess	SrcIP, TargetIP
AbnormalOperation	HostIP, TargetPath
Process	HostIP, SrcUserID, ProcessName

with their configurations. Since *FWROUTE* represents connections being blocked, their impact to the network may not be severe. Thus the corresponding 2-alert clusters are low-severity clusters.

Our last goal is to evaluate the effectiveness of our techniques in building attack scenarios. We list different resource types in Table 6.3. In addition, for implication relationships, we have *DBInfo* may imply *HostSensitiveInfo* and *NetworkSensitiveInfo* may imply *HostSensitiveInfo*. For specific-general relations, we have *DBInfo.HostIP* $\preceq$ *HostSensitiveInfo.HostIP* and *HostSensitiveInfo.HostIP* $\preceq$ *NetworkSensitiveInfo.NetworkID*. The input and output resource types for each alert

Table 6.4: Input and output resource types for alert types.

Alert Type	Input Resource Types	Output Resource Types
FTP_Globbing_Attack	{NetworkService}	{Privilege}
FWROUTE	/	{ConnectionUnavailable}
Loki	{Privilege, HostSensitiveInfo}	{HostSensitiveInfo}
NEW_CLIENT	/	/
Network Interface In Promiscuous Mode	{Privilege}	{NetworkSensitiveInfo, HostSensitiveInfo}
ASSET-DEAD	/	{ResourceUnavailable}
ASSET-SICK	/	{ResourceUnavailable}
ASSET-WELL	/	/
dbschema-downloaded	{Privilege, HostSensitiveInfo}	{DBInfo}
filesystem-integrity	{Privilege, HostSensitiveInfo}	{AbnormalOperation, HostSensitiveInfo}
registry-integrity	{Privilege, HostSensitiveInfo}	{AbnormalOperation, HostSensitiveInfo}
restricted_read	{Privilege, Process, HostSensitiveInfo}	{Process, HostSensitiveInfo}
Restricted_System_File_Scan	{Privilege, Process, HostSensitiveInfo}	{Process, HostSensitiveInfo}
restricted_write	{Privilege, Process, HostSensitiveInfo}	{Process, HostSensitiveInfo}
RootShareMounted	{Privilege, HostSensitiveInfo}	{DirectAccess, HostSensitiveInfo}

type in the experiments are listed in Table 6.4. We performed the experiments on the alerts data sets and got 10 scenario graphs. Figure 6.5 shows one of them.

Figure 6.5 is a scenario graph in HQ enclave. The alerts in this figure can roughly be divided into two parts: the right side part and the left side part. The right side part reveals that the adversaries iteratively read (*Restricted_System_File_Scan*), write (*restricted_write*) and sniff (*Network_Interface_In_PromiscuousMode*) sensitive data in HQ enclave, and use tunneling techniques such as *Loki* to secretly transmit data to the external host. The adversaries also modify critical files and keys (*filesystem-integrity* and *registry-integrity*) to disrupt the operation of the network. The left side part reveals that the adversaries use *FTP_Globbing_attack* to compromise the victim hosts, and also read and write sensitive data in the enclave. The attackers' strategy disclosed in this scenario graph is consistent with the description of GCP attack scenarios.

6.3 Summary

At present, these is no single system capable of solving all security concerns. A practical way is to deploy complementary security systems into the networks. These systems usually are heterogeneous and have different strengths and weaknesses. They usually run independently and may flag different alerts for the same attack. They may trigger large numbers of alerts where false positives are mixed with low-severity alerts and severe alerts. The low level and high volume of the alerts may also overwhelm the analysts and make identifying severe attacks and extracting the adversary's attack strategy very challenging. Alert correlation is a necessary approach to address these challenges.

We propose a correlation approach based on triggering events and common resources. One key concept in our approach is triggering events, which captures the (low-level) events that trigger alerts. We propose to group different alerts into clusters if they share "similar" triggering events, through which we can identify the alerts that may correspond to the same attack. We further introduce network and host configurations into our model, and identify consistent and inconsistent alerts, which help us mark the severity of some alerts and clusters. The other key concept in our approach is input and output resources. We propose to model each attack through specifying input and output resources, and discover causal relationships between attacks through identifying "common" resources between output resources of one attack and the input resources of another. This approach helps us identify logical connections between alert clusters and build attack scenarios. Our experimental results demonstrate the effectiveness of our techniques, though our experiments only tested data sets generated by GCP simulator.

There are several future research directions. In this chapter we mainly focus on low-level events as the triggering events. An alternative way is to use high-level events, or combine low-level and high-level events to facilitate the processing. We also notice that our approach requires a knowledge base about input and output resources of different attacks, which can be obtained through studying attack signatures. How to systematically specifying input and output resources is interesting and worth further investigation.

Chapter 7

Privacy-Preserving Alert Correlation: A Generalization Based Approach

In recent years, the security threats from infrastructure attacks such as worms and distributed denial of service attacks are increasing [19]. They affect large numbers of hosts and services on the Internet, and may bring serious financial loss. To defend against these attacks, the cooperation among different organizations is necessary. Several organizations such as CERT Coordination Center [17] and DShield [106] collect data (including security incident data) over the Internet (data may come from different data owners), perform correlation analysis, and disseminate information to users and vendors. In this chapter, we assume that there are a few data repositories which collect security data from different companies, organizations, or individuals. To facilitate the collaboration among different parties on analyzing these security data, we further assume that security data in the repositories are available or partially available to different users including attackers. In addition, attackers may also compromise the repositories to gain access to the security data. To prevent the misuse of security data, and also protect the privacy of different data owners, appropriate data sanitization through which the sensitive information is obfuscated before they are shared and analyzed is highly preferable. For example, DShield [106] lets audit log submitters perform partial or complete obfuscation to destination IP addresses in the datasets, where partial obfuscation changes the first octet of an IP address to decimal 10, and complete obfuscation changes any IP address to a fixed

value 10.0.0.1.

As we mentioned in Chapter 2, there are several alert correlation methods have been proposed in recent years. Most of these alert correlation approaches generally assume all alert data (e.g., the source and destination IP addresses) are available for analysis, which is true when there are no privacy concerns. However, when multiple data owners provide sanitized alerts and incident data (because of privacy concerns) for intrusion analysis, alert correlation will be affected due to the lack of precise data. It is desirable to have techniques to perform privacy-preserving alert correlation such that the privacy of data owners is preserved, and at the same time, alert correlation can provide useful results. To our best knowledge, [69] is the only paper addressing privacy issues in alert correlation, which uses hash functions (e.g., MD5) and keyed hash functions (e.g., HMAC-MD5) to sanitize sensitive attributes in data sets. This approach is effective in detecting some high-volume events (e.g., worms). However, since hash functions destroy the semantics of alert attributes (e.g., the loss of topological information due to hashed IP addresses), the interpretation of correlation results is non-trivial. In addition, hash functions may be vulnerable to brute-force attacks due to limited possible values of alert attributes, and keyed hash functions may introduce difficulties in correlation analysis due to the different keys used by different organizations. Nevertheless, we also notice that combining hash based methods with other methods (e.g., the methods in this and next chapters) may bring potentially better results.

In this chapter, we propose a privacy-preserving alert correlation approach based on generalization. This approach works in two phases: *entropy guided alert sanitization* and *sanitized alert correlation*. The first phase focuses on maintaining the privacy of sensitive alert data. We classify alert attributes into categorical (e.g., IP addresses) and continuous ones (e.g., the total time a process runs), and sanitize them through concept hierarchies. In a concept hierarchy, original attribute values are generalized to high-level concepts. For example, IP addresses are generalized to network addresses, and continuous attributes are generalized to intervals. We then replace original attribute values with corresponding high-level concepts, thus introducing uncertainty while partially maintaining attribute semantics. To balance the privacy and utility requirements, we guide alert sanitization with *entropy* or *differential entropy* [26] of sanitized attributes, where the desirable entropy or differential entropy values are determined by privacy policy.

To examine the utility of sanitized data sets, the second phase of our approach is to correlate sanitized alerts. As we mentioned in Chapter 2, examining similarity between alert attributes and building attack scenarios are two focuses in current correlation approaches, where similarity computation usually is the first step to study the relationship between alerts, and attack scenarios

can help understand the detailed attack steps adversaries performed. We investigate both problems under the situation where alerts are sanitized. We first examine similarity functions based on original attribute values, and then show how to revise them to calculate similarity between sanitized attributes. To build attack scenarios from sanitized alerts, we propose an *optimistic approach*. As long as it is possible that two sanitized alerts have a *causal relation* (i.e., a prepare-for relation defined in Chapter 2), we link them together. Hence multiple alerts are connected through causal relations to form attack scenarios. To measure the utility of sanitized data sets, we use measures such as *correct classification rate*, *misclassification rate*, *detection rate* and *false alert rate* to see the effectiveness of our techniques on correlation analysis of sanitized alerts. Our experimental results demonstrate the effectiveness of our approach.

7.1 Entropy Guided Alert Sanitization

The first phase of our privacy-preserving alert correlation is entropy guided alert sanitization. We use a sanitization technique based on concept hierarchies for categorical and continuous attributes. To balance the privacy and usability of alert data, alert sanitization is guided by entropy (an uncertainty measure for categorical attributes) or differential entropy (an uncertainty measure for continuous attributes). Before we go into the details of our approach, we give some definitions first.

Alert Types, Original Alerts and Sanitized Alerts. Intuitively, an alert type defines the possible attributes to describe a type of alerts. Formally, an *alert type* T is a set S of attribute names, where each attribute name $a_i \in S$ has an associated domain *Dom*(a_i). Original alerts are flagged directly by security systems. Formally, an *original alert* t_o of type T is a tuple on T's attribute names S, where for each attribute name $a_i \in S$, the corresponding element v_i in the tuple is a value in a_i's domain *Dom*(a_i).

Example 11 *An* FTP_Glob_Expansion *alert type is a set of attribute names* {SrcIP, SrcPort, DestIP, DestPort, StartTime, EndTime} *, where the domain of* SrcIP *and* DestIP *is all possible IP addresses, the domain of* SrcPort *and* DestPort *consists of all possible port numbers, and* StartTime *and* EndTime *are possible time an alert begins and ends.*

An original alert with type FTP_Glob_Expansion *is given as a tuple* {*SrcIP=10.20.1.1, SrcPort=1042, DestIP=10.10.1.1, DestPort=21, StartTime =11-10-2004 15:45:10, EndTime =11-10-2004 15:45:10*}*, which indicates that there may be an* ftp *based attack targeting at host 10.10.1.1 on port 21.*

To protect the privacy of individual alerts, we need to perform alert sanitization to sensitive attribute values (e.g., transforming sensitive data into an unintelligible form). We propose two methods to identify sensitive attribute data. (1) Identify sensitive attribute names. For each alert type, we mark some attribute names (decided by privacy policy) as sensitive attributes. Their original values are not allowed to be disclosed. (2) Identify sensitive attribute values. Sensitive attribute values are prohibited to be revealed by an organization's privacy policy. For example, an organization may decide not to disclose any IP addresses inside their network no matter they are source or destination IP addresses. The alerts after sanitization are called sanitized alerts.

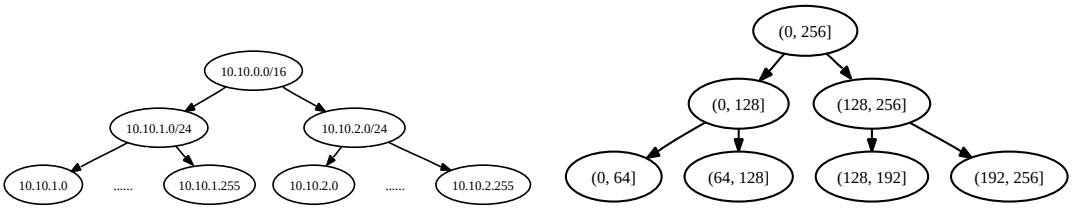
A *sanitized alert* t_s with type T is a tuple on T's attribute name set S, where for some attribute name $a_i \in S$, the corresponding element v_i in the tuple is a transformed value in domain $Dom_s(a_i)$ ($Dom_s(a_i)$ is *Dom*(a_i) or a different domain). To continue Example 11, assume *DestIP* of *FTP_Glob_Expansion* is sensitive. To sanitize the original alert, we let *DestIP*=$10.10.1.0/24$ (it is sanitized to its corresponding $/24$ network address). All the other attributes remain unchanged.

In the remainder of this chapter, we may use attributes to represent either attribute names, attribute values or both when it is clear from the context. Likewise, we may use alerts to denote either original alerts, sanitized alerts, or both. In the following, we present concept hierarchy based sanitization for categorical and continuous attributes, respectively.

7.1.1 Entropy Guided Sanitization of Categorical Attributes

Categorical attributes have discrete values. Examples of categorical attributes are IP addresses and port numbers. Concept hierarchies abstract specific (low-level) concepts into general (high-level) ones, which are widely used in data mining [49].

A concept hierarchy is based on *specific-general* relations. Given two concepts c_1 and c_2 (e.g., two attribute values), where c_1 is more specific than c_2 (or c_2 is more general than c_1), we denote the specific-general relation between c_1 and c_2 as $c_1 \preceq c_2$. As a special case, we have $c \preceq c$ for any concept c. Given an attribute name with the corresponding domain, we can define specific-



(a) A Concept Hierarchy for IP Addresses (b) A Concept Hierarchy for *CPUProcessingTime*

Figure 7.1: Two Examples of Concept Hierarchies

general relations through grouping a subset of attribute values and abstracting them into a more general concept. For example, a block of IP addresses can be organized as a subnet. Thus given an IP address $10.10.1.5$ and a subnet $10.10.1.0/24$, we have a specific-general relation $10.10.1.5 \preceq 10.10.1.0/24$.

A concept hierarchy is a set of specific-general relations, and usually is organized as a tree, where leaf nodes denote the most specific concepts (original attribute values), and the root node represents the most general concept in this hierarchy. As an example, Figure 7.1(a) shows a concept hierarchy for IP addresses. In Figure 7.1(a), IP addresses from $10.10.1.0$ to $10.10.1.255$ and from $10.10.2.0$ to $10.10.2.255$ are organized into two subnets $10.10.1.0/24$ and $10.10.2.0/24$, respectively. For each attribute (e.g., destination IP address), or a set of attributes having the same domain (e.g., both source and destination IP addresses), we can build a concept hierarchy based on the attribute domain. Then we can perform alert sanitization by replacing the original attribute values with the more general values in the hierarchy.

Example 12 *To continue Example 11, assume* DestIP *of* FTP_Glob_Expansion *is sensitive. We use the concept hierarchy in Figure 7.1(a) to perform sanitization. We replace* DestIP=*10.10.1.1 with* DestIP=*10.10.1.0/24. The other attributes remain unchanged.*

To balance the privacy and usability of alert data, we need to design a satisfactory concept hierarchy to perform sanitization, or choose appropriate general values to replace original attribute values in a given concept hierarchy. We propose to guide these processes with *entropy* [26], an *uncertainty measure* for categorical attributes.

We start with calculating the entropy of a sanitized attribute. In a concept hierarchy for a categorical attribute, given an attribute value v, which is either an original or a generalized value, we use *Node*(v) to denote the node having value v. Given a general value v_g, we use *SubTree*(v_g) to denote the subtree rooted at *Node*(v_g), and *LeafCount*(v_g) to denote the number of leaf nodes in *SubTree*(v_g). When sanitizing a categorical attribute a, an original value v_o is replaced with a general value v_g in a concept hierarchy. Notice *Node*(v_o) should be a leaf node in *SubTree*(v_g). We denote the entropy of attribute a associated with v_g as $H_a(v_g)$, where $H_a(v_g) = -\sum_{i=1}^{LeafCount(v_g)} p(a = v_i) \log_2 p(a = v_i)$. Assuming all leaf nodes in *SubTree*(v_g) have equal probabilities to be generalized to v_g, for any leaf node value v_i, the probability $p(a = v_i) = 1/LeafCount(v_g)$. Then $H_a(v_g) = \log_2 LeafCount(v_g)$. To continue Example 12, the entropy of *DestIP* associated with $10.10.1.0/24$ is $\log_2$LeafCount($10.10.1.0/24$) = $\log_2 256 = 8$.

Attribute entropy can help us design a satisfactory concept hierarchy. For example, if we want to achieve an entropy value 8 when sanitizing *DestIP* from $10.90.1.0$ to $10.90.1.255$ with equal probabilities, we can design a concept hierarchy with two levels, where the root node is a $/24$ network ($10.90.1.0/24$), and the leaf nodes are those individual IP addresses. Entropy can also help us choose an appropriate general value in a given concept hierarchy. For example, consider an original attribute *DestIP*=10.10.10.1 and a concept hierarchy in Figure 7.1(a), where leaf nodes in the hierarchy have equal probabilities. If we require an entropy value 8, we can choose the general value $10.10.1.0/24$ to sanitize the original attribute.

7.1.2 Differential Entropy Guided Sanitization of Continuous Attributes

Some attributes in an alert take continuous values, for example, the CPU time a process uses (this attribute may be sensitive, when, for example, a user makes payment based on the CPU time that his/her process consumes). To sanitize a continuous attribute, we divide the domain of the attribute into mutually exclusive intervals, and replace the original values with the corresponding intervals. Formally, if the domain of an attribute a is $Dom(a)$, we partition $Dom(a)$ into n intervals $r_1, r_2, \cdots, r_n$ such that (1) $\cup_{k=1}^{n} r_k = Dom(a)$, and (2) for any i, j, where $1 \leq i, j \leq n$ and $i \neq j$, $r_i \cap r_j = \emptyset$.

The partitions of an attribute domain can be organized into a concept hierarchy. For example, Figure 7.1(b) shows a concept hierarchy for attribute *CPUProcessingTime* (assuming its domain is interval $(0, 256]$), where the attribute domain is organized into three levels. There are several approaches that can generate concept hierarchies for continuous attributes [49]. For example, a

straightforward approach organizes a hierarchy into multiple levels, where each level has different number of equal-length intervals.

Example 13 *Consider a* JVM_Malfunction *alert with a sensitive attribute* CPUProcessingTime = *82.6 milliseconds. Using the concept hierarchy in Figure 7.1(b), we let* CPUProcessingTime = $(64, 128]$.

To design a satisfactory concept hierarchy for sanitization, or choose an appropriate interval to replace an original value in a concept hierarchy, we use *differential entropy* [26], an *uncertainty measure* for continuous attributes.

We first discuss how to compute the differential entropy of a sanitized continuous attribute. When sanitizing a continuous attribute a, an original value v_o is replaced with an interval v_g that includes value v_o. The length of interval v_g is critical to the calculation of the attribute uncertainty. We let $Length(v_g)$ denote the difference between the upper and lower bounds of interval v_g. We denote the differential entropy of a associated with v_g as $H_a(v_g)$.

$$H_a(v_g) = -\int_{v_g} f(a) \log_2 f(a) da, \tag{7.1}$$

where $f(a)$ is the probability density function for attribute a over interval v_g.

Equation 7.1 is derived and simplified from the standard form of differential entropy [26]. In the standard form, $H_a(Dom(a)) = -\int_{Dom(a)} f_o(a) \log_2 f_o(a) da$, where $f_o(a)$ is the probability density function over attribute domain $Dom(a)$. Under our sanitization technique, although we cannot know the exact value of attribute a, we are certain that it is in interval v_g, where v_g may be a part of $Dom(a)$. Then we know the probability density function $f(a)$ is 0 outside interval v_g. Thus the integration in Equation 7.1 only needs to be performed over v_g [1].

To demonstrate the uncertainty computation for sanitized continuous attributes, we derive a formula for uniformly distributed attributes. Assume an attribute a is in uniform distribution and is sanitized to interval $[\alpha, \beta]$. Thus its probability density function $f(a)$ is $1/(\beta - \alpha)$ when $\alpha \leq a \leq \beta$; otherwise $f(a) = 0$. Based on Equation 7.1, we have $H_a(v_g) = -\int_{\alpha}^{\beta} f(a) \log_2 f(a) da = \log_2(\beta - \alpha) = \log_2 Length(v_g)$.

[1] To let the probability density function $f(a)$ satisfy $\int_{v_g} f(a) da = 1$, $f(a)$ can be derived from $f_o(a)$. Assume $\int_{v_g} f_o(a) da = q \leq 1$. We can let $f(a) = f_o(a)/q$ in interval v_g; otherwise $f(a) = 0$. Another method to get $f(a)$ is to compute the distribution parameters, which is straightforward for uniformly distributed attributes.

This equation tells us that differential entropy can be greater than, equal to, or less than 0. Consider a random variable X uniformly distributed over an interval with length 1. For a sanitized continuous attribute, if its differential entropy is greater than 0, then its uncertainty is greater than variable X; if its differential entropy is equal to 0, its uncertainty is equal to X; otherwise its uncertainty is less than X. To continue Example 13, further assume attribute *CPUProcessingTime* is uniformly distributed in interval $(64, 128]$. The differential entropy of *CPUProcessingTime* associated with $(64, 128]$ is $\log_2(128 - 64) = 6$.

The differential entropy can help design a satisfactory concept hierarchy. For example, assume the domain of an attribute is $[0, 64]$ with uniform distribution. If we require a differential entropy value 5, we can build a concept hierarchy with two levels, where the root node is $[0, 64]$, and there are two leaf nodes $[0, 32]$ and $(32, 64]$. The differential entropy can also help us choose an appropriate interval to replace an original value. For example, consider an original attribute *CPUProcessingTime*=82.6 milliseconds and a concept hierarchy in Figure 7.1(b), where attributes are in uniform distribution. If we require a differential entropy value 6 for sanitization, we can choose $(64, 128]$ to replace the original value.

7.2 Correlation Analysis of Sanitized Alerts

The second phase of our approach is sanitized alert correlation. As we stated in Chapter 2, examining the similarity between alert attributes and building attack scenarios are two focuses in current correlation approaches. Similarity based correlation methods (e.g., [109, 98, 61, 28]) cluster alerts through calculating the similarity between their attributes, and methods based on predefined attack scenarios (e.g., [36, 78]) and methods based on prerequisites and consequences of attacks (e.g., [102, 29, 83]) build attack scenarios through discovering causal relations between individual alerts. To our best knowledge, these methods all assume original attribute values are available. In Subsections 7.2.1 and 7.2.2, we discuss how to compute the similarity between sanitized attributes and building attack scenarios for sanitized alerts, respectively.

7.2.1 Calculating the Similarity between Sanitized Attributes

Calculating the Similarity between Sanitized Categorical Attributes. Several functions or heuristics (e.g., techniques in [109, 98]) have been proposed to calculate the similarity

between (original) attribute values. Here we first give a simple heuristic, and then discuss how to revise this heuristic to calculate the similarity between sanitized categorical attributes. Other simple heuristics can be revised using a similar approach.

If two original attributes x_o and y_o are known, we give a similarity function between them as follows.

$$Sim(x_o, y_o) = \begin{cases} 1, & \text{if } x_o = y_o, \\ 0, & \text{otherwise.} \end{cases} \tag{7.2}$$

After sanitization, x_o and y_o become generalized values x_g and y_g, respectively. There are several ways to compute the similarity between x_g and y_g. For example, we can treat the sanitized attributes as the original ones, and use Equation 7.2 to compute their similarity. This is a coarse-grained similarity measurement because even if the sanitized values are the same, their corresponding original values may be different. In addition, even if two sanitized values are not the same, their corresponding original values are possible to be the same (e.g., two sanitized IP addresses $10.10.1.0/24$ and $10.10.0.0/16$). We propose to compute their similarity by estimating the probability that x_g and y_g have the same original value. Intuitively, in a concept hierarchy, two nodes *Node*(x_g) and *Node*(y_g) are possible to have the same original value only if they are in the same path from the root to a leaf node (*Node*(x_g) and *Node*(y_g) may be the same). In other words, there is a specific-general relation between x_g and y_g. If this is the case, the possible original values are those leaf nodes. If the probability that x_g and y_g have the same original value is large, we interpret it as a high similarity between them; otherwise their similarity is low.

Now we show how to compute the probability that x_g and y_g have the same original value hence to derive a revised similarity function. We assume x_g and y_g are generalized using the same concept hierarchy. Suppose leaf nodes in a concept hierarchy have equal probabilities. We divide the probability computation into three cases. (1) When $y_g \preceq x_g$, $SubTree(y_g)$ and $SubTree(x_g)$ may be the same, or $SubTree(y_g)$ is a subtree of $SubTree(x_g)$. $Node(x_g)$ and $Node(y_g)$ have $LeafCount(x_g)$ and $LeafCount(y_g)$ possible original values, respectively. For these two subtrees, the ratio of the number of common leaf nodes to the number of the leaf nodes in $SubTree(x_g)$ is $\frac{LeafCount(y_g)}{LeafCount(x_g)}$. Thus the probability that x_g and y_g have the same original value is $\frac{LeafCount(y_g)}{LeafCount(x_g)} \frac{1}{LeafCount(y_g)} = \frac{1}{LeafCount(x_g)}$. (2) When $x_g \preceq y_g$, we can apply a similar computation to case (1). Hence the probability that x_g and y_g have the same original value is $\frac{1}{LeafCount(y_g)}$. (3) When x_g and y_g do not have a specific-general relation, $SubTree(x_g)$ and $SubTree(y_g)$ do not have a common part in a concept hierarchy. So x_g and y_g cannot have the same original value. Their

similarity is 0. To conclude, the revised similarity function based on Equation 7.2 is as follows.

$$Sim(x_g, y_g) = \begin{cases} \frac{1}{LeafCount(x_g)}, & \text{if } y_g \preceq x_g, \\ \frac{1}{LeafCount(y_g)}, & \text{if } x_g \preceq y_g, \\ 0, & \text{otherwise,} \end{cases} \tag{7.3}$$

where "$\preceq$" denotes specific-general relations.

Calculating the Similarity between Sanitized Continuous Attributes. The similarity function between continuous attributes is different from that of categorical attributes due to various reasons. For example, due to the clock drift, two *CPUProcessingTime* may not be reported the same even if their actual time is the same. Considering these situations, here we first give a simple similarity function as follows. (Other similarity functions are possible and may be revised in a similar way to our approach.)

$$Sim(x_o, y_o) = \begin{cases} 1, & \text{if } |x_o - y_o| \leq \lambda, \\ 0, & \text{otherwise,} \end{cases} \tag{7.4}$$

where x_o and y_o are original attribute values, and λ is a predefined threshold. For example, if the *CPUProcessingTime* difference between two processes is within 5 milliseconds, we may say their similarity is 1.

When x_o and y_o are generalized to intervals x_g and y_g, respectively, there are several ways to compute the similarity between x_g and y_g. For example, assuming $Length(x_g) = Length(y_g) > \lambda$, their similarity is 1 if $x_g = y_g$, and 0 otherwise. This certainly is a rough, imprecise estimation, because even if x_g and y_g are not the same interval, it is possible that the difference between their original values is less than λ. Similar to the categorical case, we propose to compute their similarity by estimating the probability that the difference between their original values is within threshold λ.

Suppose that original values of x_g and y_g are independent and uniformly distributed over intervals x_g and y_g, respectively. To simplify our discussion, we assume *Length*(x_g) = *Length*(y_g) $> \lambda$. More sophisticated cases such as *Length*(x_g) $\neq$ *Length*(y_g) can be covered by an approach similar to the following calculation. We notice the difference between two original values may be within λ only if x_g and y_g fall into any of the following cases. (1) x_g and y_g are the same interval, (2) x_g and y_g are adjacent intervals, where adjacent intervals mean the upper bound of the lower interval and the lower bound of the higher interval are the same (e.g., $[0, 5]$ and $(5, 10]$), or (3) x_g and y_g are two intervals with a small "gap" between them, where a "gap" means the difference between the lower bound of a higher interval and the upper bound of a lower interval is greater than 0. Note

this bound difference should be within λ (e.g., $[0, 5]$ and $[6, 11]$). Now we show how to compute the probability that the difference between the original values of x_g and y_g are within λ. Suppose attribute variables for the original values of x_g and y_g are X and Y, respectively. We divide our computation into the four cases.

(1) When $x_g = y_g$, we assume x_g and y_g are interval $[\alpha, \beta]$. Based on our assumption, X and Y are independently and uniformly distributed over $[\alpha, \beta]$. We use x and y to denote attribute values of X and Y, respectively. The probability density functions of X and Y are the same. That is, $1/(\beta-\alpha)$ in $[\alpha, \beta]$, and 0 otherwise. Our goal is to get the probability $P(|x-y| \leq \lambda)$. Considering X and Y are independent, we first get the cumulative distribution function $F_{X-Y}(z)$ for $X - Y$.

$$\begin{aligned} F_{X-Y}(z) &= \iint_{x-y\leq z} f_X(x) f_Y(y) dxdy \\ &= \int_{-\infty}^{\infty} \int_{-\infty}^{y+z} f_X(y+z) f_Y(y) dxdy \\ &= \int_{-\infty}^{\infty} F_X(y+z) f_Y(y) dy. \end{aligned}$$

By differentiating $F_{X-Y}(z)$, we get the probability density function $f_{X-Y}(z)$.

$$f_{X-Y}(z) = \int_{-\infty}^{\infty} f_X(y+z) f_Y(y) dy = \int_{\alpha}^{\beta} f_X(y+z) \frac{1}{\beta-\alpha} dy.$$

Then we get

$$f_{X-Y}(z) = \begin{cases} \frac{z+\beta-\alpha}{(\beta-\alpha)^2}, & \text{if } \alpha-\beta \leq z < 0, \\ \frac{\beta-\alpha-z}{(\beta-\alpha)^2}, & \text{if } 0 \leq z \leq \beta-\alpha, \\ 0, & \text{otherwise.} \end{cases}$$

Therefore, $P(|x-y| \leq \lambda) = \int_{-\lambda}^{0} f_{X-Y}(z)dz + \int_{0}^{\lambda} f_{X-Y}(z)dz = \frac{2\lambda(\beta-\alpha)-\lambda^2}{(\beta-\alpha)^2}$. Since *Length*($x_g$)= $\beta - \alpha$, we have $P(|x-y| \leq \lambda) = \frac{2\lambda[Length(x_g)]-\lambda^2}{[Length(x_g)]^2}$.

(2) When x_g and y_g are adjacent, we assume X and Y are independently and uniformly distributed over intervals $[\alpha, \beta]$ and $(\beta, \gamma]$, respectively. We also have $\beta - \alpha = \gamma - \beta$ based on our assumption *Length*(x_g) = *Length*(y_g). Similar to case (1), we get the probability density function for $X - Y$ as follows.

$$f_{X-Y}(z) = \begin{cases} \frac{z+\gamma-\alpha}{(\beta-\alpha)^2}, & \text{if } \alpha-\gamma \leq z \leq \alpha-\beta, \\ \frac{-z}{(\beta-\alpha)^2}, & \text{if } \alpha-\beta < z < 0, \\ 0, & \text{otherwise.} \end{cases}$$

Thus $P(|x-y| \leq \lambda) = \int_{-\lambda}^{0} f_{X-Y}(z)dz = \frac{\lambda^2}{2(\beta-\alpha)^2}$. Since $Length(x_g) = \beta - \alpha$, $P(|x-y| \leq \lambda) = \frac{\lambda^2}{2[Length(x_g)]^2}$.

(3) When x_g and y_g have a "gap" between them, note we require the difference between the lower bound of the higher interval and the upper bound of the lower interval is within λ. Assume X and Y are independently and uniformly distributed over $[\alpha, \beta]$ and $[\gamma, \eta]$, respectively, where $0 < \gamma - \beta \leq \lambda$. Based on our assumption, we know $\beta - \alpha = \eta - \gamma$. Similar to case (1), the probability density function for $X - Y$ is as follows.

$$f_{X-Y}(z) = \begin{cases} \frac{z+\eta-\alpha}{(\beta-\alpha)^2}, & \text{if } \alpha - \eta \leq z < \alpha - \gamma, \\ \frac{\beta-\gamma-z}{(\beta-\alpha)^2}, & \text{if } \alpha - \gamma \leq z \leq \beta - \gamma, \\ 0, & \text{otherwise.} \end{cases}$$

Therefore, $P(|x - y| \leq \lambda) = \int_{-\lambda}^{\beta-\gamma} f_{X-Y}(z)dz = \frac{(\lambda+\beta-\gamma)^2}{2(\beta-\alpha)^2}$. For convenience, we let $d = \gamma - \beta$. Since *Length*(x_g)= $\beta - \alpha$, $P(|x - y| \leq \lambda) = \frac{(\lambda-d)^2}{2[Length(x_g)]^2}$.

(4) When x_g and y_g do not fall into any of the above three cases, based on our assumption *Length*(x_g)=*Length*(y_g)$> \lambda$, it is impossible that the difference between the original values of x_g and y_g is within λ. Hence their similarity is 0.

To conclude, the revised similarity function based on Equation 7.4 is as follows.

$$Sim(x_g, y_g) = \begin{cases} \frac{2\lambda[Length(x_g)]-\lambda^2}{[Length(x_g)]^2}, & \text{if } x_g = y_g, \\ \frac{\lambda^2}{2[Length(x_g)]^2}, & \text{if } x_g \text{ and } y_g \text{ are adjacent,} \\ \frac{(\lambda-d)^2}{2[Length(x_g)]^2}, & \text{if } x_g \text{ and } y_g \text{ have a "gap" and } 0 < d \leq \lambda, \\ 0, & \text{otherwise,} \end{cases} \tag{7.5}$$

where d is the difference between the lower bound of the higher interval and the upper bound of the lower interval. Note that similarity computation based on Equation 7.5 is symmetric ($Sim(x_g, y_g) = Sim(y_g, x_g)$).

We notice that in the probability computation, we have taken several assumptions such as *Length*(x_g)=*Length*(y_g)$> \lambda$ to simplify our calculation. However, the essential steps involved in the probability computation have already been demonstrated in our calculation. More sophisticated cases can be covered by a similar computation.

7.2.2 Building Attack Scenarios

An attack scenario is a sequence of steps adversaries performed to attack victim machines, which is helpful for security officers to learn attackers' activities and take appropriate actions. The essence of building attack scenarios from security alerts is to discover causal relations between

individual attacks. For example, there is a causal relation between an earlier *SCAN_NMAP_TCP* attack and a later *FTP_Glob_Expansion* attack if the earlier one is used to probe a vulnerable ftp port for the later one.

We extend our previous correlation method [83], which targets at building attack scenarios from original alerts, to build attack scenarios from sanitized alerts. Please refer to Chapter 2 for the formal model of the approach [83]. For convenience, before we discuss our newly proposed technique, we first give examples of prerequisites, consequences, and *prepare-for* relations, which will be used in later examples.

Example 14 *Consider alert types* T_1=SCAN_NMAP_TCP *and* T_2=FTP_Glob_Expansion. *The prerequisite of* T_1 *is* ExistHost(DestIP), *and its consequence is* $\{$ExistService(DestIP, DestPort)$\}$. *The prerequisite of* T_2 *is* ExistService(DestIP, DestPort) $\wedge$ VulnerableFtpRequest(DestIP), *and its consequence is* $\{$GainRootAccess(DestIP)$\}$.

Example 15 *To continue Example 14, consider a type* T_1 *alert* t_1 *and a type* T_2 *alert* t_2. *Assume that* t_1 *and* t_2 *both have* DestIP=*10.10.1.1 and* DestPort=*21,* t_1*'s* EndTime *is* 11-15-2004 20:15:10, *and* t_2*'s* StartTime *is* 11-15-2004 20:15:15. *Through predicate instantiation,* t_1*'s consequence is* $\{$ExistService*(*10.10.1.1, 21*)*$\}$, *and* ExistService*(*10.10.1.1, 21*)* $\wedge$ VulnerableFtpRequest*(*10.10.1.1*)* *is* t_2*'s prerequisite. Notice* t_1*.EndTime* $< t_2$*.StartTime. Then we know* t_1 *prepares for* t_2.

For convenience, we may use causal relations and *prepare-for* relations interchangeably. Given two alerts t_1 and t_2, where t_1 *prepares for* t_2, we call t_1 the preparing alert, and t_2 the prepared alert.

An Optimistic Approach to Building Attack Scenarios from Sanitized Alerts. We notice that identifying *prepare-for* relations between alerts is essential to building attack scenarios. However, after alert sanitization, we may not be certain whether *prepare-for* relations are satisfied if sanitized attributes are involved. Without loss of generality, we assume alert type data is not sanitized. We propose an *optimistic* approach to identifying *prepare-for* relations between sanitized alerts. This approach identifies a *prepare-for* relation between two alerts t_1 and t_2 as long as it is possible that (1) one of the instantiated predicates in t_1's consequence *may* imply one of the instantiated predicates in t_2's prerequisite, and (2) t_1 and t_2's timestamps *may* satisfy t_1.EndTime $<$

t_2.StartTime. In other words, based on sanitized attributes, we "guess" what possible original values are, and if these original values have a chance to satisfy the implication relationship between instantiated predicates, and also satisfy the timestamp requirement, we identify a *prepare-for* relation. Example 16 illustrates this idea.

Example 16 *To continue Examples 14 and 15, assume* DestIP *of* t_1 *and* t_2 *are sanitized based on the concept hierarchy in Figure 7.1(a), where* DestIP=$10.10.1.1$ *is replaced with* DestIP=$10.10.1.0/24$. *So* t_1*'s consequence becomes* $\{$ExistService*(*$10.10.1.0/24,\ 21$*)*$\}$, *and* ExistService*(*$10.10.1.0/24,\ 21$*)* $\wedge$ VulnerableFtpRequest *(*$10.10.1.0/24$*)* *is* t_2*'s prerequisite. It is possible that the instantiated predicate* ExistService*(*$10.10.1.0/24,\ 21$*)* *in* t_1*'s consequence implies the instantiated predicate* ExistService*(*$10.10.1.0/24,\ 21$*)* *in* t_2*'s prerequisite if both sanitized* DestIP *attributes have the same original IP address in network* $10.10.1.0/24$. *Further due to* t_1*.EndTime* $< t_2$*.StartTime, we identify a prepare-for relation between* t_1 *and* t_2.

Attack Scenario Refinement Based on Probabilities of Prepare-for Relations. Our optimistic approach certainly may introduce false *prepare-for* relations between alerts. Without knowledge of original values, we cannot guarantee that one instantiated predicate implies another if sanitized attributes are involved. To improve this approach, it is desirable to estimate how possible each pair of sanitized alerts has a *prepare-for* relation. To do so, we can first compute the probability that one instantiated predicate implies another, and then consider timestamp requirement.

Example 17 *To continue Example 16, consider* ExistService(DestIP, DestPort) *in* T_1*'s consequence and* T_2*'s prerequisite. Assume that each IP address in the concept hierarchy (Figure 7.1(a)) has equal probability in the data set. After predicate instantiation using sanitized alerts, we compute probabilities* $P(t_1$*.DestIP=*t_2*.DestIP)=*$\frac{1}{256}$, *and* $P(t_1$*.DestPort=*t_2*.DestPort)* $=1$. *Hence the probability that the instantiated predicate* ExistService*(*$10.10.1.0/24,\ 21$*)* *in* t_1*'s consequence implies the instantiated predicate* ExistService*(*$10.10.1.0/24,\ 21$*)* *in* t_2*'s prerequisite is* $\frac{1}{256}$. *Further note*

$P(t_1$.*EndTime*$< t_2$.*StartTime*$)=1$. *Then we know the probability of this prepare-for relation to be true is* $\frac{1}{256}$.

Note that sometimes computing precise probability values related to prepare-for relations is difficult when we do not know the probability distributions of attributes in original data sets. Fortunately, we can use uniform distributions to estimate lower-bound probability values. We will further discuss this problem in the next chapter (Chapter 8). We also notice that between two alerts, sometimes there may exist several pairs of instantiated predicates such that in each pair, one instantiated predicate may imply the other. If the probabilities that these pairs having implication relationships are different, it is difficult to estimate the probability that at least one implication relationship is true because we do not know the dependency among these pairs. To simplify our probability estimation, assuming n pairs of instantiated predicates that may have implication relationships are independent with probabilities p_1, p_2, $\cdots$, p_n, respectively, we use this independent case to estimate the probability that at least one implication relationship is satisfied, which is $1-(1-p_1)(1-p_2)\cdots(1-p_n)$. We then consider timestamp requirement to further compute the probability for this *prepare-for* relation. After the probabilities of *prepare-for* relations are computed, it is desirable to use these probability values to prune some false *prepare-for* relations in an alert correlation graph (e.g., remove some *prepare-for* relations with lower probability values). However, we immediately observe that this ideal case may not help much. As demonstrated by Example 17, after sanitizing the IP addresses to the corresponding $/24$ network addresses, the probability that two alerts have a *prepare-for* relation may be only $\frac{1}{256}$, which may imply that this *prepare-for* relation is *false*. However, considering that when the IP addresses in a $/24$ network are sanitized, the probabilities of all *prepare-for* relations involving these IP addresses would be small. If we remove all the low-probability *prepare-for* relations, it is very likely that some *true prepare-for* relations are pruned.

We further observe that if we calculate the probability for a set of *prepare-for* relations instead of only one, we can gain more interesting hints. Assume we have n pairs of alerts where each pair has a *prepare-for* relation with probability p. Further suppose these *prepare-for* relations are independent. Then we can compute the probability P_k that there are k pairs ($0 \leq k \leq n$) having *true prepare-for* relations: $P_k = \binom{n}{k} p^k (1-p)^{n-k}$. Based on this equation, we can compute the probability that at least one *prepare-for* relation is *true*, which is $1 - P_0$ when each *prepare-for* relation has the same probability p. When the *prepare-for* relations for n pairs of alerts have different probabilities (e.g., p_1, p_2, $\cdots$, p_n, respectively), the probability that at least one *prepare-for* relation

Algorithm: Aggregation to an alert correlation graph.
Input: An alert correlation graph CG, a temporal constraint δ, and a probability threshold θ.
Output: An aggregated correlation graph ACG.
Method:
1. Assume $CG = (N, E)$. Partition edge set E into subsets E_1, E_2, $\cdots$, E_l such that in any E_i ($1 \leq i \leq l$), all edges have the same preparing alert type, and the same prepared alert type.
2. **For** each subset E_i in E
3. Further partition E_i into groups E_{i1}, E_{i2}, $\cdots$, E_{ij} such that the preparing alerts and prepared alerts in E_{ik} ($1 \leq k \leq j$) satisfy temporal constraint δ, respectively.
4. **For** each group E_{ik} in subset E_i
5. Compute the probability $\mathcal{P}$ that at least one *prepare-for* relation in E_{ik} is *true*.
6. **If** $\mathcal{P} \geq \theta$ **Then**
7. Aggregate edges in E_{ik} into one; merge preparing and prepared alerts, respectively.
8. **Else** Remove all edges in E_{ik}.
Remove preparing and prepared alerts in E_{ik} if they are not linked by other edges.
9. Let CG after the above operations be ACG. Output ACG.
End.

Figure 7.2: An algorithm to aggregate an alert correlation graph

is *true* is $1 - (1 - p_1)(1 - p_2) \cdots (1 - p_n)$. This result may help us refine an alert correlation graph.

To further refine an alert correlation graph constructed from the optimistic approach, we propose to apply aggregation to alert correlation graphs, which is performed according to *temporal constraints* and probability thresholds.

Consider a set S of alerts and a time interval with length δ (e.g., 50 seconds), where alerts in S are sorted in increasing order based on *StartTime*. We call two alerts *consecutive alerts* if their *StartTime* timestamps are neighboring to each other in S. S satisfies temporal constraint δ if and only if for any two consecutive alerts t_i and t_j in S where t_i.StartTime $\leq t_j$.StartTime, t_j.StartTime $-t_i$.EndTime $\leq \delta$. Intuitively, a set of alerts satisfy a temporal constraint if the time intervals (in the form of [*StartTime*, *EndTime*]) of any two consecutive alerts overlap, or the "gap" between them is within δ.

Given an alert correlation graph $CG = (N, E)$ constructed from the optimistic approach, a temporal constraint δ, and a probability threshold θ, we perform aggregation to CG through the algorithm shown in Figure 7.2. The basic idea is that we aggregate the edges with the same preparing and the same prepared alert types into one such that the probability that at least one *prepare-for* relation (represented by these edges) is *true* is greater than or equal to threshold θ. (The

related nodes are merged accordingly.)

In Figure 7.2, Line 1 prepares the edge set through partitioning so that each edge subset has the same preparing alert type and the same prepared alert type. Lines 2 to 3 further partition each edge subset into groups such that the preparing and prepared alerts in each group satisfy constraint δ, respectively. Lines 4 and 5 compute the probability that at least one *prepare-for* relation is *true* in each group. If this probability is no less than threshold θ, we aggregate the edges and the related nodes in Line 7; otherwise we remove those edges and some related nodes in Line 8. Line 9 outputs the results.

As we stated earlier, the alert correlation graphs constructed from our optimistic approach may include both *false* and *true prepare-for* relations. They may also have large numbers of nodes and edges such that understanding these scenarios can be difficult and time-consuming. The algorithm in Figure 7.2 helps us improve the quality of alert correlation graphs in that it reduces the numbers of nodes and edges, and may improve the certainty about *prepare-for* relations (in the aggregated sense). Note after aggregation, a node in the aggregated correlation graph is actually a place holder which may represents multiple alerts. Our aggregation also has some limitations because we may remove some *prepare-for* relations from alert correlation graphs when the probability for them is less than the threshold. Our experiments in Subsection 7.3.2 indicate that the aggregation should be applied with caution since *true prepare-for* relations have a chance to be removed. The alert correlation graphs created from the optimistic approach and the aggregated correlation graphs are complementary to each other, and they should be referred to each other to comprehensively learn the security threats.

Though the above approach to building attack scenarios from sanitized alerts is extended from a specific correlation approach (i.e., [83]), other correlation methods such as approaches based on predefined attack scenarios can be extended to accommodate sanitized alerts in a similar way. The approaches based on predefined attack scenarios usually define constraints among alert types (e.g., a logical formula involving attribute names from several alert types). If any constraints are satisfied by alert attributes, causal relations and hence attack scenarios are created. To apply our optimistic approach to these methods, we can examine the constraints to see whether they are possible to be satisfied based on sanitized attributes. To use probabilities to refine attack scenarios, we can calculate the probabilities for the constraints to be satisfied in a similar way to our approach.

7.3 Experimental Results

To learn the effectiveness of our techniques, we performed a set of experiments to evaluate the similarity functions and the approach to building attack scenarios.

7.3.1 Evaluating Similarity Functions

In the first two experiments, we focus on the evaluation of revised similarity functions (e.g., Equations 7.3 and 7.5). We are interested in how possible sanitized datasets can provide similarity classification as that from original datasets. In our experiments, we randomly generated a set S_o of alerts with only one categorical attribute (or one continuous attribute, respectively), and then sanitized it to get a new set S_s. Next for each combination of two alerts in S_o, we used Equation 7.2 (or Equation 7.4, respectively) to calculate attribute similarity. While for each pair of alerts in S_s, we used Equation 7.3 (or Equation 7.5, respectively) to compute their similarity. Then we applied an optimistic classification. If the similarity value is greater than 0, we classify this pair of alerts as "similar" pair; otherwise we classify them as "distinct" pair. We compared the results from S_s with those from S_o. We used two quantitative measures: *correct classification rate* R_{cc} for S_s based on S_o and *misclassification rate* R_{mc} for S_s based on S_o. We define R_{cc} and R_{mc} for "similar" pairs as follows. $R_{cc} = \frac{\#\text{common "similar" pairs in both } S_o \text{ and } S_s}{\#\text{"similar" pairs in } S_o}$, and $R_{mc} = \frac{\#\text{"similar" pairs in } S_s - \#\text{common "similar" pairs in } S_o \text{ and } S_s}{\#\text{total alert pairs} - \#\text{"similar" pairs in } S_o}$. Note that R_{cc} and R_{mc} are only for sanitized datasets, and both measures can be computed for "similar" or "distinct" pairs. Likewise, we define correct classification rate and misclassification rate for "distinct" pairs by replacing "similar" with "distinct" in the above two equations.

Our first experiment is for categorical attributes. We generated a set S_o of $2,560$ alerts with *DestIP* attributes uniformly distributed over 256 IP addresses in network $10.60.1.0/24$ (from $10.60.1.0$ to $10.60.1.255$). Next we partitioned this network into 16 subnets. Each subnet ($/28$ subnet) has 16 addresses. We sanitized S_o to S_s such that *DestIP* of each alert is generalized to the corresponding $/28$ subnet ID. We applied Equation 7.2 to S_o and Equation 7.3 to S_s. The results are shown in the left part of Table 7.1.

Our second experiment is for continuous attributes. We generated a set S_o of $1,000$ alerts with *CPUProcessingTime* attributes uniformly distributed over interval $[0, 100]$. Then we divided $[0, 100]$ into 20 small equal-length intervals (the length of each small interval is 5). Next we sanitized S_o to S_s by replacing original values with the corresponding small intervals (a boundary

Table 7.1: The results of evaluating similarity functions

	Categorical attribute		Continuous Attribute	
	S_o (original)	S_s (sanitized)	S_o (original)	S_s (sanitized)
# alerts	$2,560$		$1,000$	
# total alert pairs	$3,275,520$		$499,500$	
# "similar" pairs	$12,818$	$204,585$	$24,444$	$71,705$
# common "similar" pairs	$12,818$		$24,444$	
# "distinct" pairs	$3,262,702$	$3,070,935$	$475,056$	$427,795$
# common "distinct" pairs	$3,070,935$		$427,795$	
R_{cc} for "similar" pairs	N/A	100%	N/A	100%
R_{mc} for "similar" pairs	N/A	5.88%	N/A	9.95%
R_{cc} for "distinct" pairs	N/A	94.12%	N/A	90.05%
R_{mc} for "distinct" pairs	N/A	0%	N/A	0%

value between two adjacent intervals is put into the lower interval). Let $\lambda = 2.5$. We applied Equation 7.4 to S_o and Equation 7.5 to S_s. The results are shown in the right part of Table 7.1.

In these two experiments, the entropy and differential entropy for attributes *DestIP* and *CPUProcessingTime* are $log_2 16 = 4$ and $log_2 5 = 2.3219$, respectively. Our correct classification rates for both "similar" and "distinct" pairs are high (greater than 90%), while the misclassification rates for both pairs are low (less than 10%). This demonstrates that the privacy of alert attributes can be protected with sacrificing the data functionality (similarity classification) slightly.

7.3.2 Building Attack Scenarios

To evaluate the techniques on building attack scenarios, we performed a set of experiments on 2000 DARPA intrusion detection scenario specific data sets [77]. The datasets include two scenarios: LLDOS 1.0 and LLDOS 2.0.2, where each scenario includes two parts (inside and DMZ). In LLDOS 1.0, adversaries probed vulnerable sadmind services in the networks, broke into the hosts through sadmind buffer overflow attacks, installed mstream DDoS softwares, and finally launched DDoS attacks. The attack scenario of LLDOS 2.0.2 is similar to that of LLDOS 1.0 (but LLDOS 2.0.2 is more stealthier).

In the first set of experiments, our goal is to evaluate the effectiveness of our optimistic approach to building attack scenarios. We first used RealSecure network sensor 6.0 to generate alerts from four datasets: LLDOS 1.0 inside, LLDOS 1.0 DMZ, LLDOS 2.0.2 inside, and LLDOS 2.0.2 DMZ. Next we defined the prerequisites and consequences for all alert types reported by the

network sensor. Such information can be obtained in Tables 5.3 and 5.4. We first constructed alert correlation graphs for the original alert datasets using our previous method [83]. Then we sanitized the destination IP address of each alert (which is a sanitization policy applied by DShield [106]) by replacing it with its corresponding $/24$ network ID (e.g., $172.16.112.50$ is sanitized to $172.16.112.0/24$). Then we applied our optimistic approach to building alert correlation graphs for the four datasets. One alert correlation graph constructed from LLDOS 1.0 inside dataset is listed in Figure 7.3.

In Figure 7.3, the string inside each node is an alert type followed by an alert ID. Notice that to show the difference between the alert correlation graphs created from the original dataset and the sanitized one, we marked the additional nodes obtained only from the sanitized dataset in gray. From Figure 7.3, it is clear that the alert correlation graph constructed from the sanitized dataset is a supergraph of the one created from the original dataset. This observation is consistent with our intuition because our optimistic approach identifies *prepare-for* relations even if their related probabilities are low. The alert correlation graph in Figure 7.3 can be divided into multiple stages: the adversaries used *Sadmind_Ping* to probe vulnerable sadmind services, next used *Sadmind_Amslverify_Overflow* to get root privileges, then installed mstream DDoS softwares and started them via *Rsh*, and finally the mstream components communicated with each other (*Mstream_Zombie*) and launched DDoS attacks (*Stream_DoS*). They are consistent with the major steps adversaries performed.

We notice that false alerts may be involved in an alert correlation graph (e.g., the alert *Email_Debug67705* in Figure 7.3). To further evaluate the effectiveness of our approach, similar to [83], we used two quantitative measures: *soundness* M_s and *completeness* M_c, where $M_s = \frac{\#\text{correctly correlated alerts}}{\#\text{correlated alerts}}$, and $M_c = \frac{\#\text{correctly correlated alerts}}{\#\text{related alerts}}$. We computed both measures for the correlation approach based on original datasets and the correlation approach (i.e., our optimistic approach) based on sanitized datasets. The results are in Table 7.2. Comparing both measures in Table 7.2, the correlation approach based on original datasets is slightly better than our optimistic approach, which is reasonable because original datasets are more precise than sanitized datasets. Nevertheless, our optimistic approach is relative good: the majority of soundness measures are greater than 70%, and all completeness measures are greater than 60%.

In the second set of experiments, our goal is to verify whether correlation methods can help us differentiate between true and false alerts. We conjecture that correlated alerts are more likely to be true alerts, and false alerts are more random and have less chance to be correlated. This conjecture has been experimentally verified in [83] when original alerts are available. Now

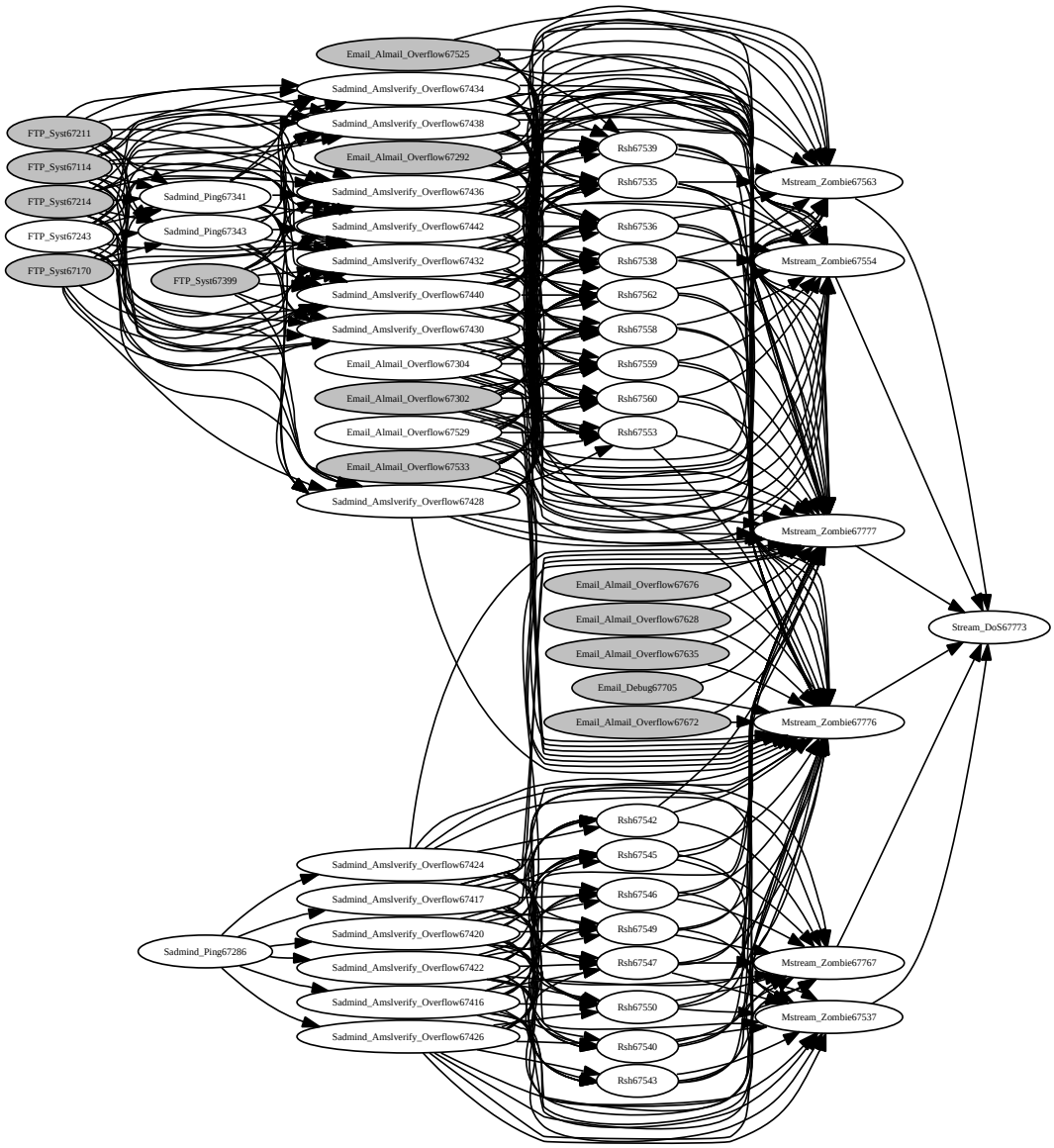


Figure 7.3: An alert correlation graph in LLDOS 1.0 inside dataset

we try to see the results when alerts are sanitized. Similar to [83], we compute detection rate as $\frac{\#\text{detected attacks}}{\#\text{observable attacks}}$, and false alert rate as $1 - \frac{\#\text{true alerts}}{\#\text{alerts}}$. In our experiments, we calculated detection rates and false alert rates for RealSecure network sensor, the correlation approach based on original datasets, and the correlation approach (i.e., our optimistic approach) based on sanitized datasets. The results are shown in Table 7.3. In Table 7.3, the numbers of alerts for correlation approaches

Table 7.2: Soundness and completeness measures in our experiments

	LLDOS 1.0		LLDOS 2.0.2	
	Inside	DMZ	Inside	DMZ
# correlated alerts for original datasets	44	57	13	5
# correctly correlated alerts for original datasets	41	54	12	5
# correlated alerts for sanitized datasets	58	63	25	6
# correctly correlated alerts for sanitized datasets	41	54	12	5
# related alerts	44	57	18	8
Soundness M_s for original datasets	93.18%	94.74%	92.31%	100%
Completeness M_c for original datasets	93.18%	94.74%	66.67%	62.50%
Soundness M_s for sanitized datasets	70.69%	85.71%	48.00%	83.33%
Completeness M_c for sanitized datasets	93.18%	94.74%	66.67%	62.50%

Table 7.3: Detection rates and false alert rates in our experiments

	Detection approach	LLDOS 1.0		LLDOS 2.0.2	
		Inside	DMZ	Inside	DMZ
# alerts	RealSecure	922	886	489	425
	Correlation for original datasets	44	57	13	5
	Correlation for sanitized datasets	58	63	25	6
Detection rate	RealSecure	61.67%	57.30%	80.00%	57.14%
	Correlation for original datasets	60.00%	56.18%	66.67%	42.86%
	Correlation for sanitized datasets	60.00%	56.18%	66.67%	42.86%
False alert rate	RealSecure	95.23%	93.57%	96.73%	98.59%
	Correlation for original datasets	6.82%	5.26%	23.08%	40.00%
	Correlation for sanitized datasets	29.31%	14.29%	60.00%	50.00%

are the numbers of correlated alerts. We observe that our optimistic approach for sanitized datasets still has the ability to greatly reduce false alert rates, while slightly sacrificing detection rates. In addition, comparing the detection rates and false alert rates, the approach based on original datasets is slightly better than our optimistic approach since original datasets have more precise information than sanitized ones.

In the third set of experiments, our goal is to evaluate the effectiveness of the aggregation to alert correlation graphs. Here we show one case for LLDOS 1.0 inside dataset. We aggregated the alert correlation graph in Figure 7.3 based on the algorithm in Figure 7.2, where we set temporal constraint $\delta = \infty$ and probability threshold $\theta = 0.1$. The result is shown in Figure 7.4. In Figure 7.4, we notice that some false alerts are ruled out (e.g., *Email_Debug67705*), which is highly

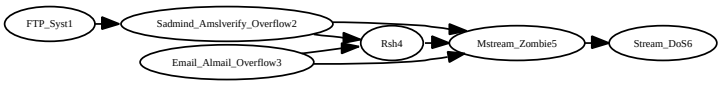


Figure 7.4: Aggregation to the alert correlation graph in Figure 7.3

preferable. However, we also observe that some true alerts are pruned (e.g., three *Sadmind_Ping* alerts), which is undesirable. Though it is possible to mitigate this undesirable case through setting a lower probability threshold, we can never guarantee that only false alerts will be ruled out. We conclude that aggregation should be applied with caution. The alert correlation graphs created from our optimistic approach and the aggregated correlation graphs should be referred to each other to comprehensively learn the security threats.

7.4 Summary

In this chapter, we propose a generalization based approach to perform privacy-preserving alert correlation. We divide our approach into two phases. The first phase is entropy guided alert sanitization, which focuses on protecting the privacy of data sets. We replace sensitive original attribute values with high-level concepts in concept hierarchies, which introduces uncertainty into data sets, and also partially maintains attribute semantics. We further propose to use entropy and differential entropy to measure the uncertainty of sanitized attributes, and also guide the generalization of original attributes. To examine the utility of sanitized alerts, the second phase of our approach is sanitized alert correlation. We concentrate on defining similarity functions between sanitized attributes, and building attack scenarios from sanitized alerts. We use various measures such as correct classification rate and false alert rate to measure the utility of sanitized data sets. Though our experiments mainly focus on 2000 DARPA intrusion detection scenario specific data sets, and we used a simple attribute sanitization policy, we would expect some observations from our experiments are also useful to other data sets. For example, attack scenarios constructed from sanitized data sets (without probability based refinement) are supergraphs of the ones constructed from original data sets, and probability based pruning may filter out both false and true prepare-for relations. We also notice that to apply our approach, some expect knowledge is necessary, for example, deciding desirable entropy values for sensitive attributes when performing sanitization, and

designing concept hierarchies with the requirement of desirable entropy values.

There are several problems worth further investigation. Our techniques in this chapter replace original attributes with general values in concept hierarchies. Generalized attributes usually have different domains compared with original attributes. Observing this may let malicious users immediately realize that attributes are sanitized, which may further let them infer privacy policy. We will propose other sanitizaiton techniques in the next chapter to address this problem. Another problem in our approach is that the probability based pruning may filter out both false and true prepare-for relations. Thus additional techniques that can better refine alert correlation graphs are worth further investigation. In addition, we are also interested in the performance of privacy-preserving alert correlation techniques.

Chapter 8

Privacy-Preserving Alert Correlation: A Perturbation Based Approach

As we mentioned in Chapter 7, to defend against large-scale distributed attacks such as worms and distributed denial of service (DDoS) attacks, it is usually desirable to deploy security systems such as intrusion detection systems (IDSs) over the Internet, monitor different networks, collect security related data, and perform analysis to the collected data to extract useful information. In addition, different organizations, institutions, and users may also have the willingness to share their data for security research as long as their privacy concerns about the data can be fully satisfied. For example, DShield [106] collects firewall logs from different users to learn global cyber threats, and Department of Homeland Security sponsors PREDICT [51] project to create a repository collecting network operational data for cyber security research. In this chapter, similar as in Chapter 7, we assume that there are a few data repositories collecting security data from different organizations, companies, and individuals. To facilitate the collaboration on security research, we further assume that these security data sets are available or partially available to different users including attackers.

Data generated by security systems may include sensitive information (e.g., IP addresses of compromised servers) that data owners do not want to disclose or share with other parties, where sensitive data are decided by data owners' privacy policy. To protect the privacy of data owners,

and prevent the misuse of these security data, it is always desirable and sometimes mandatory to anonymize sensitive data before they are shared and correlated. To address this problem, existing approaches usually perform transformation to sensitive data. Fox example, Lincoln et al. [69] propose to use hash functions and keyed hash functions to anonymize sensitive attributes such as IP addresses. Their approach is effective on detecting high-volume events, but may have limitations on alert correlation if different keys are introduced in keyed-hashing. We also notice that combining their hash based methods with other techniques (e.g., the techniques proposed in Chapter 7) may bring potentially better results. In Chapter 7, we propose to abstract original values to more general values (e.g., IP addresses are replaced by network addresses). Since general values may usually take different formats compared with original values (i.e., they have different attribute domains), observing this may let malicious users immediately realize that attributes are sanitized, which may infer organizations' privacy policy.

In this chapter, we address this problem in another complementary direction. We start to hide original sensitive values through injecting more data into data sets. We also perform transformation to sensitive attributes, but this is carried over the same attribute domains. In this chapter, we propose three perturbation based schemes (Schemes I, II and III) to flexibly anonymize sensitive attributes of intrusion alerts. These schemes are closely related and can also be applied independently. In Scheme I, we intentionally generate artificial alerts and mix them with original alerts, thus given any alert in the mixed set, it is not clear that this alert is original (i.e., IDS-reported) or artificial, which means its attributes may or may not be real. To protect data privacy, artificial alerts should not be obviously distinguishable from original alerts. On the other hand, we also need to maintain the utility of the data. With both requirements in mind, during artificial alert generation, we preserve frequency distributions of attack types and non-sensitive attributes, while use concept hierarchies to facilitate the generation of sensitive attributes. Notice that concept hierarchies can help us abstract attribute values, for example, IP addresses can be generalized to the corresponding network addresses. In Scheme II, we propose to map original sensitive attributes to random values based on concept hierarchies. And in Scheme III, we propose to partition an alert set into multiple subsets based on time constraints and perform Scheme II independently in each subset. To measure data privacy hence to guide the procedure of alert anonymization, we propose two measures: *local privacy* and *global privacy*, where local privacy is related to original values of sensitive attributes for individual alerts, and global privacy is related to distributions of sensitive attributes in alert sets. Both privacy values are computed based on entropy, and desirable entropy values are decided by privacy policy.

Though we emphasize alert anonymization techniques in this chapter, we also perform alert correlation to anonymized alerts to examine the utility of the data. Similar as in Chapter 7, we focus on two problems: estimating similarity values between anonymized attributes and building attack scenarios from anonymized alert set. Our method on similarity measurement is a probability based approach, which estimates how possible two anonymized attributes may have the same original values. Our approach on building attack scenarios extends from our existing method [83], where the probabilities related to the matching of prerequisites and consequences among different attacks are estimated. Though it is closely related to the optimistic approach in Chapter 7, the probability estimation is based on the anonymization schemes proposed in this chapter. Based on these probability values, we can construct and further "polish" attack scenarios. Similar as in Chapter 7, we also use various measures such as *correct classification rate* to measure the utility of anonymized data sets. Our experimental results demonstrated the effectiveness of our techniques in terms of various measures.

8.1 Three Schemes for Alert Anonymization

In this chapter, we emphasize our alert anonymization techniques, which can flexibly protect data privacy. Before we go into the details of our techniques, we clarify some notions and definitions first.

An alert type is a type name T and a set $\mathcal{S}$ of attribute names, where each attribute name in $\mathcal{S}$ has a related domain denoting possible attribute values. As an example, an alert type *FTP_AIX_Overflow* has a set of six attribute names {*SrcIP, SrcPort, DestIP, DestPort, StartTime, EndTime*}, where the type name *FTP_AIX_Overflow* denotes that it is a buffer overflow attack targeting *AIX* ftp services, and all six attributes are used to describe this type of attacks. The domains of *SrcIP* and *DestIP* are all possible IP addresses, the domains of *SrcPort* and *DestPort* are possible port numbers (from port 0 to port 65535), and *StartTime* and *EndTime* denote the timestamps that the attack begins and finishes.

An original alert is an instance of alert types and is reported by security systems. Formally, a type T original alert t_o is a tuple on T's attribute set $\mathcal{S}$, where each attribute value in this tuple is a value in the related domain.

Example 18 *Assume we have a type* FTP_AIX_Overflow *alert* {*SrcIP*=$172.16.10.28$, *SrcPort*=1081,

DestIP=172.16.30.6, *DestPort*=21, *StartTime=01-16-2006 18:01:05, EndTime=01-16-2006 18:01:05*}.

This alert describes an FTP_AIX _Overflow *attack from IP address* 172.16.10.28 *to target* 172.16.30.6.

A type T *artificial alert* has the same format as that of an original alert. The only difference between artificial alerts and original alerts is that original alerts are reported by security systems, while artificial alerts are synthetic, and may be generated by a human user, or some programs. The purpose of generating artificial alerts is to help protect the privacy of sensitive attribute values in original alerts. We will discuss how to generate artificial alerts in Subsection 8.1.1. Similarly, a type T *anonymized alert* has the same format as that of an original alert. However, the sensitive attribute values in anonymized alerts are transformed, for example, through randomization, to protect data privacy. To continue Example 18, if *DestIP* of the alert is sensitive, we can transform *DestIP*=172.16.30.6 to *DestIP*=172.16.30.35 to hide the original value. We will discuss how to transform sensitive values in Subsection 8.1.2. In the rest of this chapter, we call the set of alerts all flagged by security systems the original alert set, and the set of alerts including both original and artificial alerts the mixed alert set. In addition, we may use attributes to represent either attribute names, attribute values, or both if it is clear from the context.

8.1.1 Scheme I: Artificial Alert Injection Based on Concept Hierarchies

Intuitively, artificial alert injection generates synthetic alerts and mixes them with original alerts. Given any alert in a mixed alert set, identifying whether it is artificial or original is difficult, and the information disclosed by any individual alert may not necessarily be true. The critical issue in artificial alert injection is how to generate attribute values for each artificial alert, with both privacy and usability requirements in mind. Here we divide alert attributes into three classes: alert types, sensitive attributes, and nonsensitive attributes, and discuss them separately.

Alert types encode valuable information about the corresponding attacks. For example, RealSecure network sensor 6.5 [52] may report an *FTP_AIX_Overflow* attack. Based on the signature of this attack, we know that it is a buffer overflow attack targeting AIX FTP services. Alert types are crucial for security officers to learning security threats. To maintain the utility of alert data sets, it is usually desirable to disclose alert type information. So when we create artificial alerts, we propose to preserve the frequency of the original data set in terms of alert types, where the frequency of an alert type T is the ratio of the number of alerts with type T to the total number of alerts. In other words, if an original alert data set has n types of alerts with frequencies $p_1, p_2, \cdots, p_n$ where

$p_1 + p_2 + \cdots + p_n = 1$, then in our artificial alert set, we will maintain this frequency distribution.

Sensitive attributes are decided by privacy policy, and their values are what we try to protect. Considering both privacy and utility concerns, we propose to use concept hierarchies to help us artificially generate sensitive attribute values. Creating attribute values based on concept hierarchies may preserve some useful information (e.g., prefixes of IP addresses), but also change attribute distributions to certain degree to protect alert privacy. Before we discuss our algorithm on artificial alert generation, we first introduce concept hierarchies.

Concept hierarchies have been used in areas such as data mining [49] and also in privacy-preserving techniques (e.g., $k-$Anonymity approach [95]). A concept hierarchy is based on *specific-general* relations. Given two concepts c_1 and c_2 (e.g., attribute values), if c_1 is more specific than c_2 (or equivalently, c_2 is more general than c_1), we say there is a *specific-general* relation between c_1 and c_2, and denote it as $c_1 \preceq c_2$. As an example, given an IP address $172.16.10.3$ and a network address $172.16.10.0/24$, we have $172.16.10.3 \preceq 172.16.10.0/24$. Note that specific-general relation is reflexive, antisymmetric and transitive. Specific-general relations can be obtained through abstracting a set of low-level concepts to a high-level concept. For example, a set of individual IP addresses can be organized into a subnet. Based on specific- general relations, a *concept hierarchy* is a set of specific-general relations and is usually organized into a tree. Figure 8.1 shows a concept hierarchy for IP addresses $172.16.11.0$, $\cdots$, $172.16.11.255$ and $172.16.12.0$, $\cdots$, $172.16.12.255$, where each IP address is generalized first to its corresponding $/24$ network address, and then to its $/16$ network address.

For continuous attributes, we can group data into bins thus continuous values can be transformed into categorical. For example, given a set of timestamps within a one-hour time interval, we may partition the whole time interval into 60 equal-length bins where each bin is a one-minute time interval, and put timestamps into the corresponding bins. Binning techniques have been extensively studied in the fields such as data mining [49] and statistics, and we do not repeat them here. In this chapter, our techniques focus on categorical data, though they can be extended to accommodate continuous data.

Given a concept hierarchy H, and two nodes v_s and v_g in H where $v_s \preceq v_g$ (this means v_s has a path to v_g in H), the *distance* between v_s and v_g is the number of edges over the path from v_s to v_g, denoted as *distance*(v_s, v_g). For example, in Figure 8.1, *distance*($172.16.11.3$, $172.16.11.0/24$)$= 1$. Given a concept hierarchy H and two nodes v_{s1} and v_{s2} in H, we call a node v_g in H the *least common parent* if (1) $v_{s1} \preceq v_g$, (2) $v_{s2} \preceq v_g$, and (3) $d_m =$*max*(*distance*(v_{s1}, v_g), *distance*(v_{s2}, v_g)) has a minimum value. In addition, if v_{s1} and v_{s2} are both leaf nodes in H and

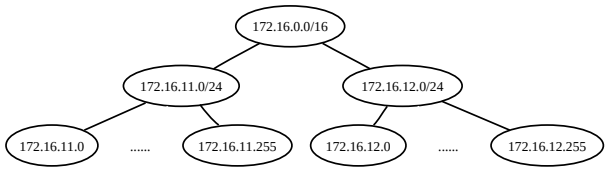


Figure 8.1: An example concept hierarchy for IP addresses

the least common parent v_g has total $\mathcal{L}$ leaf nodes including v_{s1} and v_{s2}, we call nodes v_{s1} and v_{s2} *$\mathcal{L}$-peer nodes*, or simply *$\mathcal{L}$-peers*. As an example, in Figure 8.1, the least common parent of two leaf nodes $172.16.11.3$ and $172.16.11.5$ is node $172.16.11.0/24$. Since node $172.16.11.0/24$ has totally 256 leaf nodes, two nodes $172.16.11.3$ and $172.16.11.5$ are 256-peers.

Now let us discuss how to generate sensitive attributes for artificial alerts. We assume each sensitive attribute value has a desirable general value in concept hierarchies, where these desirable general values can be derived through a given parameter $\mathcal{L}$ denoting the desirable number of peer nodes. For example, if $\mathcal{L} = 256$ for attribute *DestIP*, then the desirable general values for these IP addresses are the corresponding $/24$ network addresses. We first compute the frequency distribution of these generalized attribute values based on the original data set. Next, following the computed frequency distribution, we create generalized attribute values in the artificial alert set, finally we replace these generalized values using leaf nodes in the corresponding hierarchies (each leaf node value has equal probability to be chosen). Notice that during the above procedure, we preserve attribute frequency in terms of general values. This is because these general values partially maintain the utility of alert data (e.g., prefixes of IP address). We also replace these general values using their corresponding leaf nodes with uniform distribution. This may help us change the distribution of attributes in original sets. For example, if in an original set, the values for attribute *DestIP* is only from $172.16.11.0$ to $172.16.11.31$. Further suppose we set parameter $\mathcal{L} = 256$. Then we will generate artificial attributes uniformly distributed from $172.16.11.0$ to $172.16.11.255$ to change attribute distribution and hence protect original values. Notice that desirable general values (or, parameter $\mathcal{L}$) for sensitive attributes are decided by privacy policy. For example, if we let *DestIP*'s desirable general values be the corresponding $/24$ network addresses, this means that ideally, we want each *DestIP* in its $/24$ network to be equally likely in alert sets, so malicious users may not be able to "guess" which values are more possible.

To generate nonsensitive attributes, we first compute the frequency distribution of their

original values, then we generate artificial attribute values with the same frequency distribution in the artificial alert set. As a special case, for timestamp information, we first get the minimum and maximum timestamps for each alert type in the original alert set, then we uniformly create timestamps between the minimum and maximum values for artificial alerts.

Another important issue on artificial alert generation is to decide how many alerts to generate. Notice that injecting artificial alert data usually may change the distribution of attribute values. Intuitively, if a large number of artificial alerts are generated, we may better protect alert privacy (attributes are more uniformly distributed), but the utility of alerts may decrease. On the other hand, if only a small number of alerts are created, we may increase the utility, but alert privacy may decrease. So the ideal case is to flexibly control the difference between attribute distributions based on the requirement from privacy protection. This can help us decide the number of artificial alerts to be generated.

We propose to use the distance between probability mass functions (PMFs) to measure the difference between attribute distributions. Given one attribute A_s in both original alert set S_o and mixed alert set S_m, assume the PMFs for A_s in S_o and S_m are $f_o(x)$ and $f_m(x)$, respectively, where x is possible values for A_s. Further assume the domain of attribute A_s is $Dom(A_s)$. To calculate the distance between two PMFs, we first need define distance function $D(f_o, f_m)$ between $f_o(x)$ and $f_m(x)$. As an example, we can set $D(f_o, f_m) = \sum_{x \in Dom(A_s)} |f_m(x) - f_o(x)|$. (Other distance functions are also possible.[1]) Through setting a desirable distance threshold, we may inject the number of alerts satisfying the threshold. (In case sensitive attributes in original data sets are in uniform distributions, we may further specify a maximum number of artificial alerts to prevent infinite alert injection.)

In Figure 8.2, we summarize our algorithm on artificial alert generation. In Line 1, we prepare the mix alert set. From Line 2 to 5, we compute various distributions in the original data set for later usage. From Line 7 to 14, we generate one artificial alert. We use different strategies to generate sensitive attributes, timestamps, and other non-sensitive attributes. In Line 15, we compute distances between PMFs. These distances combined with the maximum number of artificial alerts can help us control the actual number of alerts injected. In Line 17, we output the mixed alert set.

Notice that in the above algorithm, we do not consider the dependence between different

[1] For ordinal data, we may also use cumulative distribution functions (CDFs) to compute the distance between attribute distributions. Assume the CDFs for A_s in S_o and S_m are $F_o(x)$ and $F_m(x)$, respectively. We may set $D(F_o, F_m) = \sum_{x \in Dom(A_s)} |F_m(x) - F_o(x)|$. As another example, we may also use some goodness-of-fit test statistics such as Kolmogorov-Smirnov statistic (which lets $D(F_o, F_m) = max_{x \in Dom(A_s)} |F_m(x) - F_o(x)|$) to compute the difference between CDFs.

Algorithm. Generation of Artificial Alerts.
Input: An original alert set S_o, concept hierarchies for sensitive attributes, a parameter $\mathcal{L}$ denoting the desirable number of peers, a distance function D between PMFs, a threshold value d denoting the desirable distance between PMFs, and a threshold value n_a of the maximum number of artificial alerts.
Output: A mixed set S_m of both original and artificial alerts.
Method:

1. Initialize set S_m to empty. Copy all alerts in S_o to S_m.
   Initialize a distance value $d_c = 0$. Initialize the number of artificial alerts $n_c = 0$.
2. Compute the frequency distribution FD_T of alert types in S_o.
3. Based on $\mathcal{L}$, find desirable general values in concept hierarchies for sensitive attributes in S_o.
   Compute the frequency distribution in terms of these general values for each alert type.
4. As of timestamps, get minimum and maximum timestamps for each alert type.
5. Compute frequency distributions of other nonsensitive attributes for each alert type.
6. **While** $d_c < d$ and $n_c < n_a$
7. Generate an alert type T following the distribution FD_T.
   Initialize an empty artificial alert t_a with type T.
8. **For**each attribute A in t_a
9. **If**A is a sensitive attribute
10. Create a general value v_g following the frequency distribution computed in Step 3.
    Replace v_g uniformly with v_g's leaf nodes that having $\mathcal{L}$-peers.
11. **Else if** A is a timestamp
12. Choose a timestamp uniformly from the minimum and maximum timestamps of T.
13. **Else if** A is another nonsensitive attribute
14. Create an attribute value following the frequency distribution computed in Step 5.
15. For each sensitive attribute, compute distance based on D between S_o and S_m.
16. Let the minimum distance computed in Step 15 be d_c. Let $n_c = n_c + 1$. Put t_a in S_m.
17. Output S_m.

End.

Figure 8.2: An algorithm to generate artificial alerts

attributes (e.g., the dependence between attributes *DestIP* and *DestPort*). When there are no such dependence, we can use the algorithm in Figure 8.2 to generate all alerts. However, when there does exist the dependence, we need to handle this situation with caution. As an example, if there are only one web server $172.16.10.5$ with port 80 open in a network, then usually all those web based attacks are targeting $172.16.10.5$ on port 80. This means IP address $172.16.10.5$ and port number 80 are dependent in web based attacks. Artificial alert generation, on the one hand, does not require to strictly satisfy this dependence, because the violation of this dependence may bring confusion

to and require further investigation from malicious users, which in some sense may protect data privacy. However, on the other hand, if we try to make artificial alerts and original alerts very difficult to distinguish, or the utility of data is our favorable concern, we can also maintain attribute dependence during artificial alert generation. We propose two ways to get dependence relationships between attribute values.

1. Manually collect all dependence relationships through various means. For example, based on attack signatures, and host and network configurations, we can know the hosts and ports that some attacks are targeting.

2. Compute conditional probabilities between attribute values based on original alert sets, and follow these conditional probabilities to generate attribute values in artificial alert sets. This approach is similar to the data-swapping technique proposed by Reiss [92].

To see how well our anonymization schemes can help protect alert data privacy, we classify alert data privacy into two levels: *local privacy* and *global privacy*. Local privacy is related to original attribute values in each individual alert. Intuitively, if the original attribute value for a sensitive attribute in an alert has been known, the local privacy for the sensitive attribute in this alert is compromised. The second level of alert data privacy is global privacy, which is related to the distributions of sensitive attributes in alert set. Intuitively, if the distribution of a sensitive attribute in an original alert set is known, we can derive useful information about the original set (e.g., the most possible value in the data set). For both local and global privacy, we use *entropy* [26] to measure them. Suppose that we have a value v_s for a sensitive attribute A_s in an alert t. Based on v_s, if we can estimate the possible original values of A_s in t and the corresponding probability for each possible value, then we can compute alert t's local privacy $H_l(t) = -\sum p_i \log_2 p_i$, where p_i is the probability for a possible value. Given all attribute values for a sensitive attribute A_s in an alert set S, we can estimate alert set S' global privacy $H_g(S) = -\sum P(v_i) \log_2 P(v_i)$, where v_i is a possible value for A_s in S, and $P(v_i)$ is the corresponding probability. To help us better understand global privacy, we may explain it from a random variable (or event) point of view. Assume that a sensitive attribute is a random variable, and attribute values for this sensitive attribute in an alert set is a realization of this random variable. Then the global privacy for the attribute in the given alert set is a randomness (or uncertainty) measure about the realization of this random variable. Recall that we generate uniformly distributed data (based on concept hierarchies) during artificial alert injection. The reason is that injecting uniformly distributed data generally may increase the randomness of data sets. Also notice that the distance between PMFs and the change in global privacy

are closely related because we change attribute distributions through injecting uniformly distributed data. And it is also feasible to control the number of artificial alerts through adjusting the change in global privacy.

Back to artificial alert injection, if original alert set S_o has m alerts, we inject n artificial alerts in it, thus we totally get $m + n$ alerts in mixed set S_m. In S_m, each individual alert has probability $\frac{m}{m+n}$ to be original, and probability $\frac{n}{m+n}$ to be artificial. So its local privacy is $-\sum p_i \log_2 p_i = \frac{m}{m+n} \log_2 \frac{m+n}{m} + \frac{n}{m+n} \log_2 \frac{m+n}{n}$. We can also calculate global privacy for both S_o and S_m, and compute their difference to see how well we can improve global privacy. One of our later experiments shows that through injecting 168 artificial alerts into an original set with 922 alerts, we achieve local privacy with value 0.62, and we improve global privacy from 4.696 to 5.692 (the distance between two PMFs is 0.3).

8.1.2 Scheme II: Attribute Randomization Based on Concept Hierarchies

In Scheme I, we inject artificial alerts into original data set. For any individual alert in the mixed alert set, it may be difficult to identify whether this alert is original or artificial, hence sensitive attribute values in any single alert have a chance to be faked.

Let us look at Scheme I in detail. Assume that we have m original alerts, and inject n artificial alerts into it. Hence in the mixed alert set, each alert has a probability $\frac{m}{m+n}$ to be original (or, every alert has a probability $\frac{n}{m+n}$ to be artificial). Based on probability theory, randomly pick up k alerts in the mixed set, the probability that at least one alert is original is $1 - (\frac{n}{m+n})^k$, and the probability that at least one alert is artificial is $1 - (\frac{m}{m+n})^k$. As an example, let $m = 1000$, $n = 300$, and $k = 2$, thus the probability that at least one alert is original is 94.67%, while the probability that both alerts are artificial is only 5.33%. This tells us that when the ratio of the number of artificial alerts to the total number of alerts is small, it is very likely that some alerts in an alert subset are original even if this subset is small. Closely investigating these small subsets may disclose the privacy of alert data. This problem may be mitigated by injecting more artificial alerts. In the extreme case, if we inject a much larger number of artificial alerts (i.e., $m \ll n$), then in a randomly selected k-alert subset, the probability that at least one alert is original may be very small even if k is big. For example, let $m = 1,000$ and $n = 1,000,000$. When $k = 100$, the probability that at least one alert is original is only 9.51%, and even when $k = 1,000$, the probability value only increases to 63.19%. Notice that with the increase of the number of artificial alerts, the utility of mixed data sets may be decreasing, and more overhead is introduced to analyze

Algorithm. Randomization for Sensitive Attributes.
Input: An alert set S, a sensitive attribute name A_s, a concept hierarchy H for A_s, and a parameter $\mathcal{L}$ denoting the desirable number of peers.
Output: A set S_r of alerts where attribute A_s are randomized.
Method:

1. Initialize S_r to be empty.
2. **While** the set S has more alerts
3. Pick an alert t from S and put t into S_r.
   Assume the value of A_s in t is v_s.
   Based on H, uniformly select a value v_r from H where v_r and v_s are $\mathcal{L}$-peers.
   Replace v_s with v_r in t.
4. **For**each alert t_i in S where t_i's attribute value of A_s is v_s
5. Remove t_i from S and put it into S_r. Replace t_i's value of A_s with v_r.
6. Output S_r.

End.

Figure 8.3: An algorithm to randomize sensitive attributes

mixed alert sets. Considering privacy, utility, and performance, we propose to apply randomization to sensitive attributes, which may allow us protect the privacy of original alerts without injecting a huge amount of artificial alerts. Notice that our randomization algorithm takes advantage of concept hierarchies, which can preserve some useful information in sensitive attributes (e.g., prefixes of IP addresses).

Though we motivate attribute randomization in term of mixed alert sets, it can be applied to original alert sets. Given a parameter $\mathcal{L}$ denoting the desirable number of peers in concept hierarchies, the basic idea of our algorithm is to randomize each different attribute value v_i uniformly to any of v_i's $\mathcal{L}$-peers (In other words, the mapping and mapped values has a common general value v_g where v_g has $\mathcal{L}$ leaf nodes). For example, based on the concept hierarchy in Figure 8.1, we may randomize a sensitive attribute *DestIP*=172.16.11.8 to *DestIP*=172.16.11.56 if $\mathcal{L} = 256$. During randomization, we keep *consistency* in attribute mapping, which means if two alerts has the same attribute values for an attribute A_s, then the mapped values for A_s in both alerts are the same. For convenience, we call the mapped value the *image value*, or simply the *image*. This consistency is desirable for later correlation analysis of alert data sets, and help us maintain the utility of alert data. In Figure 8.3, we sketch the algorithm for attribute randomization.

In Figure 8.3, we first initialize the result set in Line 1. In Line 3, we map an attribute value in the alert set to one of its $\mathcal{L}$-peers (with uniform distribution). From Line 4 to 5, we keep

consistency among attribute mapping. And finally in Line 6, we output the result set.

To see how attribute anonymization may help protect alert privacy, let us take a look at both local privacy and global privacy. Suppose that we randomize a sensitive attribute A_s in an original alert set. After performing randomization, given any image value v_r, we know the original value of v_r may be any of v_r's $\mathcal{L}$-peers with equal probability (i.e., $\frac{1}{\mathcal{L}}$). Thus the local privacy value is $-\sum_{i=1}^{\mathcal{L}} \frac{1}{\mathcal{L}} \log_2 \frac{1}{\mathcal{L}} = \log_2 \mathcal{L}$. If we randomize a mixed alert set, we can also derive the corresponding local privacy after considering the probability that a given alert may be original. On the other hand, based on concept hierarchies and requirements for local privacy values, we may choose desirable parameters (e.g., $\mathcal{L}$) satisfying the requirements. To consider global privacy, let us assume that there are k distinct values for sensitive attribute A_s in an alert set S. Since we keep consistency during randomization, there will be at most k distinct image values for A_s in randomized alert set S_r. If k distinct image values do exist, then S_r have the same global privacy as in S. When less than k distinct image values are generated (this is possible because two different original values may happen to be randomized to the same value), the global privacy value $H_g(S_r)$ in S_r may change compared with the value in S, where $H_g(S_r)$ may be slightly smaller. Our later experimental results confirm this conjecture. For example, in one data set, we set $\mathcal{L} = 256$, and the global privacy slightly changes from 5.692 (before randomization) to 5.671 (after randomization). To summarize, our attribute randomization may result in slight change (or no change) in global privacy, and desirable change (through choosing appropriate parameters) in local privacy.

8.1.3 Scheme III: Alert Set Partitioning and Attribute Randomization

In Scheme II, we randomize sensitive attribute values to their $\mathcal{L}$-peers and maintain consistency during randomization. A limitation related to this scheme is that once attackers get to know some *(original value, image value)* pairs, then for all the image values in the known pairs, with high probability, attackers know their corresponding original values in the whole alert set. Notice that (original value, image value) pairs can be obtained through various means, for example, deliberately attacking some special hosts and examining published alert data (e.g., counting frequencies of some attacks). To mitigate this problem, we propose to partition an alert set into multiple subsets and perform randomization (Scheme II) in each subset independently. Thus one original value may be mapped to different image values in different subsets.

Our algorithm on partitioning an alert set is based on a *time constraint*. Given a time interval $\mathcal{I}$ (e.g., 2 hours) and a set $\mathcal{S}$ of alerts, we say $\mathcal{S}$ satisfies time constraint $\mathcal{I}$ if $|max(EndTime) -$

$min(StartTime)| \leq \mathcal{I}$ (i.e., the difference between the maximum *EndTime* and the minimum *StartTime* in $\mathcal{S}$ is less than or equal to $\mathcal{I}$). Based on a time constraint, we partition an alert set into multiple subsets such that each subset satisfies the time constraint, then we randomize each subset through Scheme II independently.

Now let us look at local and global privacy under Scheme III. For local privacy, since image values are chosen uniformly from $\mathcal{L}$-peers, we have a similar analysis as in Scheme II. However, for global privacy, since it is possible that one original value in different subsets may be randomized to different image values, global privacy usually may increase after applying Scheme III compared with applying Scheme II. Our later experimental results confirm this conjecture. For example, in one experiment, we partitioned one data set into four subsets, which increases global privacy from 5.692 to 6.955.

8.2 Anonymized Alert Correlation

Through our focus in this chapter is alert anonymization techniques, we are also interested in the utility of anonymized alert sets. To understand the utility of alert data, it is usually necessary to perform intrusion alert correlation. Notice that current alert correlation approaches usually concentrate on computing similarity values between alert attributes, or building attack scenarios to reflect attackers' activities. So in this section, we will focus on these two problems under the situation that alerts are anonymized. Notice that our approaches are closed related to our previous method proposed in Chapter 7, however, the alerts that we correlate are anonymized by the schemes proposed in this chapter.

8.2.1 Similarity Estimation between Anonymized Attributes

Similarity measurement computes how similar two attributes are, usually with a value between 0 and 1. Existing approaches [109, 98, 61] focus on measuring similarity between original attributes. Our anonymization techniques (Schemes II and III) transform original attributes to random values based on concept hierarchies. Thus it is crucial and helpful to estimate similarity between original attributes only using anonymized values. In the following, we first give an example function on computing similarity between original values, then discuss how to estimate similarity values based on anonymized attributes.

Assume that we have a sensitive attribute A_s and two original attributes values x_1 and x_2 for A_s. As an example similarity function, we let $Sim(x_1, x_2) = 1$ if $x_1 = x_2$, and $Sim(x_1, x_2) = 0$ if $x_1 \neq x_2$. We further assume x_1 and x_2's images are y_1 and y_2, respectively, after performing randomization. We consider two cases regarding whether x_1 and x_2 are in the same subset (an set is partitioned into multiple subsets in Scheme III).

(1) When x_1 and x_2 are in the same subset, we have the following observation.

Observation 3 *For a sensitive attribute A_s, given two original attribute values x_1 and x_2 where they are in the same subset, and two image values y_1 and y_2 randomized from x_1 and x_2 using Scheme II (with same parameters such as $\mathcal{L}$), we know that (i) if $x_1 = x_2$, then $y_1 = y_2$; (ii) if $y_1 = y_2$, x_1 and x_2 may or may not be the same.*

We explain our observation through examples. Assume x_1 and x_2 are both destination IP addresses where $x_1 = 172.16.11.5$. Using the concept hierarchy in Figure 8.1, suppose we randomize x values to one of its 256-peer nodes, and x_1 is mapped to y_1 with value $172.16.11.98$. Since we keep consistency in Scheme II, if $x_2 = 172.16.11.5$, then we know $y_2 = y_1 = 172.16.11.98$. On the other hand, if $x_2 \neq x_1$, for example, let $x_2 = 172.16.11.9$, then y_2 can be any IP address from $172.16.11.0$ to $172.16.11.255$, which has a chance to be $172.16.11.98$. To better characterize this observation, we compute the following probabilities.

For simplicity, assume the domain of x is $\mathcal{L}$ specific values where each value has equal probability to be chosen, and two original values x_1 and x_2 are randomized using their $\mathcal{L}$-peer nodes. Based on conditional probability and total probability theorems, we can derive

$$\begin{aligned} P(x_1 = x_2 | y_1 = y_2) &= \frac{P(x_1 = x_2 \wedge y_1 = y_2)}{P(y_1 = y_2)} \\ &= \frac{P(x_1 = x_2 \wedge y_1 = y_2)}{P(y_1 = y_2 | x_1 = x_2)P(x_1 = x_2) + P(y_1 = y_2 | x_1 \neq x_2)P(x_1 \neq x_2)} \\ &= \frac{\frac{1}{\mathcal{L}}}{\frac{1}{\mathcal{L}} + \frac{\mathcal{L}-1}{\mathcal{L}}\frac{1}{\mathcal{L}}} \\ &= \frac{\mathcal{L}}{2\mathcal{L}-1}. \end{aligned}$$

Similarly, we can get $P(x_1 \neq x_2 | y_1 = y_2) = \frac{\mathcal{L}-1}{2\mathcal{L}-1}$, $P(x_1 = x_2 | y_1 \neq y_2) = 0$, and $P(x_1 \neq x_2 | y_1 \neq y_2) = 1$.

Notice that though we use the assumption of uniform distribution about original values to derive $P(x_1 = x_2|y_1 = y_2) = \frac{\mathcal{L}}{2\mathcal{L}-1}$, we can prove that for other distributions, we have $P(x_1 = x_2|y_1 = y_2) \geq \frac{\mathcal{L}}{2\mathcal{L}-1}$ as long as the attribute domain is the same. We prove it through Lemma 8.2.1.

Lemma 8.2.1 *Given a sensitive attribute A_s with domain $\{v_1, v_2, \cdots, v_n\}$ (v_i is a possible attribute value for A_s), suppose x_1 and x_2 are two original values for A_s, and y_1 and y_2 are two image values for x_1 and x_2, respectively, after applying Scheme II. Further assume the number of desirable peer nodes in Scheme II is $\mathcal{L}$. $P(x_1 = x_2|y_1 = y_2)$ has a lower bound when v_1, v_2, $\cdots$, v_n are in uniform distribution.*

Proof: First, let us assume that in the original alert set, $P(x_1 = x_2) = \alpha$. Then we have

$$\begin{aligned} & P(x_1 = x_2|y_1 = y_2) \\ = & \frac{P(x_1=x_2 \wedge y_1=y_2)}{P(y_1=y_2)} \\ = & \frac{P(x_1=x_2 \wedge y_1=y_2)}{P(y_1=y_2|x_1=x_2)P(x_1=x_2)+P(y_1=y_2|x_1\neq x_2)P(x_1\neq x_2)} \\ = & \frac{\alpha}{\alpha+\frac{1-\alpha}{\mathcal{L}}} \\ = & \frac{\mathcal{L}}{\mathcal{L}-1+\frac{1}{\alpha}} \end{aligned}$$

Now let us discuss how to compute α. Suppose the probabilities for possible attribute values v_1, v_2, $\cdots$, v_n in A_s's domain are p_1, p_2, $\cdots$, p_n, respective, where $p_1 + p_2 + \cdots + p_n = 1$. Then $\alpha = p_1^2 + p_2^2 + \cdots + p_n^2$.

Based on Cauchy's inequality $(\sum_{i=1}^n a_i b_i)^2 \leq (\sum_{i=1}^n a_i^2)(\sum_{i=1}^n b_i^2)$, we can derive $\alpha = \sum_{i=1}^n p_i^2 \geq (\sum_{i=1}^n p_i)^2/n = \frac{1}{n}$. Then we know that the minimum value of α is $\frac{1}{n}$, where $p_1 = p_2 = \cdots = p_n = \frac{1}{n}$ (uniform distribution). Next we prove $P(x_1 = x_2|y_1 = y_2) = \frac{\mathcal{L}}{\mathcal{L}-1+\frac{1}{\alpha}}$ is monotonically increasing when $0 < \alpha < 1$.

Assume $0 < \alpha_1 < \alpha_2 < 1$. Then $\frac{1}{\alpha_1} > \frac{1}{\alpha_2}$. Next we have $\mathcal{L} - 1 + \frac{1}{\alpha_1} > \mathcal{L} - 1 + \frac{1}{\alpha_2}$. Finally we get $\frac{\mathcal{L}}{\mathcal{L}-1+\frac{1}{\alpha_1}} < \frac{\mathcal{L}}{\mathcal{L}-1+\frac{1}{\alpha_2}}$.

To summarize, we know that $P(x_1 = x_2|y_1 = y_2)$ have a minimum value $\frac{\mathcal{L}}{\mathcal{L}-1+\frac{1}{\alpha}}$ when v_1, v_2, $\cdots$, v_n are in uniform distribution.

(2) When x_1 and x_2 are in different subset, x_1 and x_2 are randomized independently. Assume their randomization has the same input parameters (e.g., $\mathcal{L}$). With the similar reasoning as in (1), we know that as long as x_1 and x_2 are $\mathcal{L}$-peers, y_1 and y_2 have a chance to be the same, or

to be different. Under the same assumption as in (1), we can derive that $P(x_1 = x_2|y_1 = y_2) = \frac{1}{\mathcal{L}}$, $P(x_1 \neq x_2|y_1 = y_2) = \frac{\mathcal{L}-1}{\mathcal{L}}$, $P(x_1 = x_2|y_1 \neq y_2) = \frac{1}{\mathcal{L}}$, and $P(x_1 \neq x_2|y_1 \neq y_2) = \frac{\mathcal{L}-1}{\mathcal{L}}$.

As an example to estimate similarity between y_1 and y_2, we estimate how possible their corresponding x_1 and x_2 are the same, and use this probability value as their similarity value. Based on the above assumption and reasoning, we can get an example similarity function for anonymized attributes as follows.

$$Sim(y_1, y_2) = \begin{cases} \frac{\mathcal{L}}{2\mathcal{L}-1}, & \text{if } (y_1 \text{ and } y_2 \text{ are } \mathcal{L}\text{-peers}) \wedge (y_1 = y_2) \wedge (y_1 \text{ and } y_2 \text{ in the same subset}), \\ \frac{1}{\mathcal{L}}, & \text{if } (y_1 \text{ and } y_2 \text{ are } \mathcal{L}\text{-peers}) \wedge (y_1 \text{ and } y_2 \text{ in different subsets}), \\ 0, & \text{otherwise.} \end{cases} \tag{8.1}$$

Notice that to derive the new similarity function above, we only consider a simple case regarding how possible original attributes may be the same. For more complicated case, we may apply a similar, probability-based approach to derive new functions for anonymized attributes.

8.2.2 Building Attack Scenarios

Attack scenarios help us understand what steps adversaries take to attack victim machines. For example, an attacker may first run IP sweep to detect live IP addresses, followed by port scanning attacks to look for open ports, and finally launch buffer overflow attacks to gain root privileges on some live hosts. To build attack scenarios, it is crucial to identify causal relations between individual attacks. For example, there is a causal relation between an earlier IP sweep attack and a later port scanning attack because the IP sweep attack may detect live IP addresses, which can be further probed by the port scanning attack to determine what ports are open.

There are several approaches being proposed to build attack scenarios. They can be classified into two categories: (1) known attack scenario based approaches such as [36, 78], and (2) prerequisite and consequence based methods such as [102, 29, 83]. These approaches can build attack scenarios when original attributes are known. To build attack scenarios from anonymized alerts, we use a probability based approach, which is extended from our previous correlation method [83]. For the formal model of our approach [83], please refer to Chapter 2 for details. In the following, to facilitate our discussion, we first give examples of prerequisites, consequences, and *prepare-for* relations, which will be used in later examples.

Example 19 *Given an alert type* FTP_AIX_Overflow*, its prerequisite is* ExistService(DestIP,DestPort)*, and its consequence is* {GainAccess(DestIP)}*, which means that the necessary condition of an* FTP_AIX_Overflow *attack is that* FTP *service is running on* DestIP *at port* DestPort*, and the consequence of this attack is that attackers may gain unauthorized access to* DestIP.

Example 20 *Assume that we have two alert types* Port_Scan *and* FTP_AIX_Overflow *where the consequence of type* Port_Scan *is* EixistService(DestIP, DestPort)*. Further assume that we have two alerts* t_1 *and* t_2 *where* t_1 *is a type* Port_Scan *alert* {SrcIP=172.16.10.28, SrcPort=1073, DestIP=172.16.30.6, DestPort=21, StartTime=01-16-2006 18:00:02, EndTime=01-16-2006 18:00:02 }*, and* t_2 *is a type* FTP_AIX_Overflow *alert* {SrcIP=172.16.10.28, SrcPort=1081, DestIP=172.16.30.6, DestPort=21, StartTime=01-16-2006 18:01:05, EndTime=01-16-2006 18:01:05 }*. Thus the instantiated consequence of* t_1 *is* {ExistService(172.16.30.6, 21)}*, and the instantiated prerequisite of* t_2 *is* ExistService(172.16.30.6, 21)*. Further due to* t_1*.EndTime* $< t_2$*.StartTime, we know* t_1 *prepares for* t_2*.*

A Probability Based Approach to Building Attack Scenarios.

To build attack scenarios, it is critical to identify prepare-for relations. When all original values are known, this identification is straightforward. However, when alerts are anonymized through randomization, identifying prepare-for relations requires more efforts.

Example 21 *Let us re-consider Example 20. Assume that* DestIP *is sensitive, and we do not know their original values in both alerts* t_1 *and* t_2*. Based on the prerequisites, the consequences, and the available nonsensitive values, we know that* t_1 *prepares for* t_2 *only if* t_1 *and* t_2 *have the same original destination IP addresses. For simplicity, assume that the original values of* DestIP *are uniformly distributed from* $172.16.30.0$ *till* $172.16.30.255$*, and attribute randomization uniformly*

chooses IP addresses from $172.16.30.0$ *to* $172.16.30.255$ *to replace original values (the number of peers used in Scheme II is* $\mathcal{L} = 256$*). We consider two cases. (1) Suppose that* t_1 *and* t_2 *are in the same subset (in terms of alert set partitioning in Scheme III). If* t_1 *and* t_2*'s anonymized destination IP addresses are the same (e.g., both are* $172.16.30.52$*), then based on our reasoning in similarity estimation (Subsection 8.2.1), we know that with probability* $\frac{\mathcal{L}}{2\mathcal{L}-1} = \frac{256}{511} = 0.501$*,* t_1 *and* t_2 *may have the same original destination IP addresses. Equivalently,* t_1 *and* t_2 *may have different original destination IP addresses with probability* 0.499*. In addition, if after randomization,* t_1 *and* t_2 *have different anonymized destination IP addresses, we know that their original destination IP addresses are different. (2) Suppose that* t_1 *and* t_2 *are in different subsets. we know that* t_1 *and* t_2 *may be possible (with probability* $\frac{1}{\mathcal{L}} = \frac{1}{256}$*) to have the same original destination IP addresses as long as their anonymized values are* $\mathcal{L}$*-peers.*

Based on the above observation, we realize that we can only identify possible prepare-for relations after attribute randomization. To characterize this observation, we propose to associate a probability value to each possible prepare-for relations when building attack scenarios from anonymized alerts. Notice that sometimes precisely computing the probability that one alert prepares for another is difficult because analysts do not know probability distributions of original attribute sets. However, as we mentioned in Subsection 8.2.1 and also proved in Lemma 8.2.1, we can get lower bound probability values under the assumption of uniform distributions. We take advantage of this observation and define *possibly-prepare-for* relation. Formally, given two anonymized alerts t_1 and t_2, t_1 *possibly prepares for* t_2 with at least probability p if (1) the probability that t_1 prepares for t_2 is no less than p, and (2) $p > 0$. Our probability based approach is closely related to the optimistic approach in Chapter 7. However, here probabilities related to possibly-prepare-for relations are lower bound values and are estimated based on the anonymization schemes in this chapter. To continue Example 21, (1) when t_1 and t_2 are in the same subset, if t_1 and t_2 have the same anonymized *DestIP*, we know t_1 possibly prepares for t_2 with at least probability 0.501; (2) when t_1 and t_2 are in different subsets, if their anonymized destination IP addresses are $\mathcal{L}$-peers, t_1 possibly prepares for t_2 with at least probability $\frac{1}{256}$.

In the above example, there are only one implication relationship between instantiated predicates for two alerts. It is also possible that there may exist multiple implication relationships. To deal with this situation, similar as in Chapter 7, we assume each implication relationship are independent, and then estimate the probability that at least one implication relationship is true. In particular, if there are n implication relationships, and the probability that each implication relationship is true is at least p_1, p_2, $\cdots$, p_n, respectively, then the probability that at least one implication relationship is true has a lower-bound value $1-(1-p_1)(1-p_2)\cdots(1-p_n)$.

To build attack scenarios from anonymized alerts, we identify all possibly-prepare-for relations and connect them into a graph. For convenience, in the remainder of this chapter, we may use prepare-for relations to represent either prepare-for relations, possibly-prepare-for relations, or both, if it is clear from the context.

Lower-bound probabilities related to prepare-for relations may also be used to "polish" alert correlation graphs. Actually if we take a close look at how we compute these probability values, the basic problem involved is to decide how possible the related attributes have the same original values. In some cases, we can estimate precise probability values. For example, suppose the desirable number of peers is $\mathcal{L}$ when applying Scheme III. If two anonymized attributes are in different subsets, and they are $\mathcal{L}$ peers, then the probability that they have the same original value is $\frac{1}{\mathcal{L}}$. However, in some other cases, for example, two anonymized attributes are in the same subset and have the same anonymized values, we usually may assume uniform distribution to get lower-bound probability values. Considering that prepare-for relations are identified through this probability based approach, it is natural that some prepare-for relations may not be true. A common way to filter out false prepare-for relations is to examine their related (lower-bound) probabilities. If the probability is greater than a given probability threshold, we keep it in the correlation graph, otherwise we remove it. Notice that in Chapter 7, a probability based refinement to alert correlation graphs also has been proposed. However, the approach in Chapter 7 is to calculate the probability for a set of prepare-for relations[2], and use this probability to perform aggregation to alert correlation graphs. While the approach in this chapter is to examine and filter out individual prepare-for relations. In addition, we agree that the approach in Chapter 7 is complementary to the approach in this chapter, and may be applied here. Though our approach on polishing alert correlation graphs may help us remove false prepare-for relations, we also notice that it has a chance to prune true prepare-for relations. It is necessary to examine both alert correlation graphs with and without

[2]If there are n prepare-for relations with probabilities p_1, p_2, $\cdots$, p_n, respectively, and these prepare-for relations are independent, then the probability that at least one prepare-for relation is true is $1-(1-p_1)(1-p_2)\cdots(1-p_n)$.

probability-polishing to understand attackers' activities.

8.3 Experimental Results

To evaluate the effectiveness of our techniques, we did a set of experiments to evaluate Scheme I (artificial alert injection), Scheme II (attribute randomization), Scheme III (alert set partitioning and attribute randomization), similarity estimation and building attack scenarios. The data sets we used are 2000 DARPA intrusion detection scenario specific data sets [77]. These data sets were collected in simulation networks and include two scenarios: LLDOS 1.0 and LLDOS 2.0.2. Both scenarios have two data sets collected over different parts of the networks: the inside part and the DMZ part. In LLDOS 1.0, the attackers first probed the network, then launched buffer overflow attacks against vulnerable *Sadmind* services, next installed DDoS software on victim hosts, and finally ran DDoS attacks against a server. LLDOS 2.0.2 has a similar scenario as in LLDOS 1.0. We used RealSecure network sensor 6.0 [52] to generate alerts from the data sets. Similar as done by DShield [106], we set attribute *DestIP* (destination IP addresses) in all alerts as sensitive attribute, and anonymized them using the schemes in this chapter.

8.3.1 Experiments on Scheme I

In our first set of experiments, our goal is to evaluate the effectiveness of artificial alert injection. We injected artificial alerts into all four data sets through the algorithm in Figure 8.2. To apply this algorithm, for sensitive attribute *DestIP*, we set the desirable general value for each address to its corresponding $/24$ network address, which means that we replace a general value using one of 256 IP addresses in the corresponding network. We used distance function D between PMFs to control artificial alert injection, where $D(f_o, f_m) = \sum_{x \in Dom(A_s)} |f_m(x) - f_o(x)|$. We set distance threshold (between PMFs) $d = 0.3$ and maximum artificial alert number $n_a = 300$. The experimental results are shown in Table 8.1, Figure 8.4, and Table 8.2.

Part of data utility is related to alert type frequencies (e.g., security officers may count attack type frequencies to see what events are most possible). To learn whether our algorithm can preserve type frequencies in mixed alert sets (compared with original sets), we computed type frequencies for all data sets. For demonstration purpose, we show the results for one data set in Table 8.1. Based on Table 8.1, we observed that each alert type has very close frequencies both in

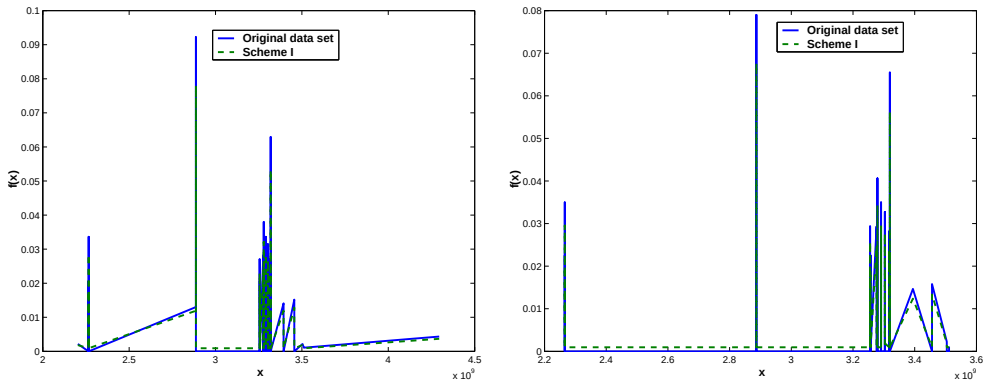
Table 8.1: Alert type frequency distribution in LLDOS 1.0 inside part

Alert type	Frequency in original set	Frequency in mixed set
Admind	.018438	.016513
Email_Almail_Overflow	.041214	.040366
Email_Debug	.002169	.001834
Email_Ehlo	.566160	.570642
FTP_Pass	.053145	.056880
FTP_Syst	.047722	.046788
FTP_User	.053145	.052293
HTTP_Cisco_Catalyst_Exec	.002169	.001834
HTTP_Java	.008676	.011009
HTTP_Shells	.016268	.015596
Mstream_Zombie	.006507	.005504
Port_Scan	.001084	.000917
RIPAdd	.001084	.000917
RIPExpire	.001084	.002752
Rsh	.018438	.016513
Sadmind_Amslverify_Overflow	.015184	.019266
Sadmind_Ping	.003253	.002752
SSH_Detected	.004338	.003669
Stream_DoS	.001084	.000917
TelnetEnvAll	.001084	.000917
TelnetTerminaltype	.136659	.131192
TelnetXdisplay	.001084	.000917

original sets and mixed sets.

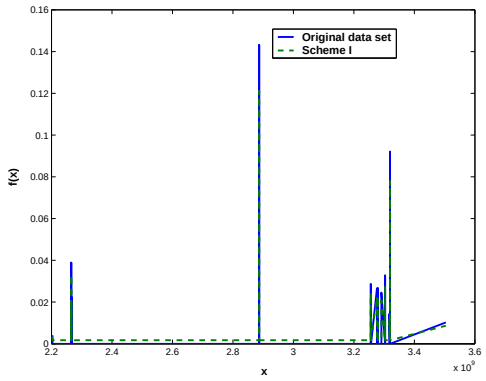
To see how our algorithm may change the distributions of sensitive attributes to protect data privacy, we further plotted the PMFs for *DestIP* in both original and mixed data sets in Figure 8.4(a), 8.4(b), 8.4(c), and 8.4(d). Note that in these figures, each destination IP address is transformed into an integer. If the format of an IP address is $A.B.C.D$ where A, B, C and D are integers between 0 to 255, then $A.B.C.D$ is transformed to an integer $x = A \times 2^{24} + B \times 2^{16} + C \times 2^8 + D$. Based on these figures, we observed that the distributions in mixed sets are changed compared with original data sets.

To more precisely evaluate alert privacy, we computed local and global privacy for each data set. The results are shown in Table 8.2. Based on this table, we noticed that through Scheme I, we can better protect alert privacy because both local and global privacy values increase. For example, in LLDOS 1.0 inside part, we injected around 15% of artificial alerts among all alerts, and

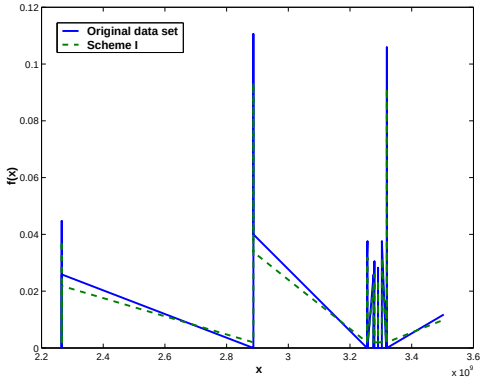


(a) LLDOS1.0Inside (Original & Scheme I)

(b) LLDOS1.0DMZ (Original & Scheme I)



(c) LLDOS2.0.2Inside (Original & Scheme I)



(d) LLDOS2.0.2DMZ (Original & DMZ)

Figure 8.4: PMFs in original alert set and after applying Scheme I

local privacy increases from 0 to 0.620, and global privacy increases from 4.696 to 5.692.

8.3.2 Experiments on Scheme II

In this set of experiments, our goal is to see the effectiveness of attribute randomization using Scheme II. We applied Scheme II to all four data sets. Particularly, we randomized destination IP addresses in mixed alert sets to their 256-peers (any IP addresses in the corresponding $/24$ networks). Figures 8.5(a), 8.5(b), 8.5(c), and 8.5(d) show the PMFs after applying Scheme II in mixed data sets (solid blue lines).

Based on these figures, we observed that the distributions of *DestIP* have been greatly

Table 8.2: Local and global privacy (S_o: original set, S_m: mixed set, attribute: *DestIP*).

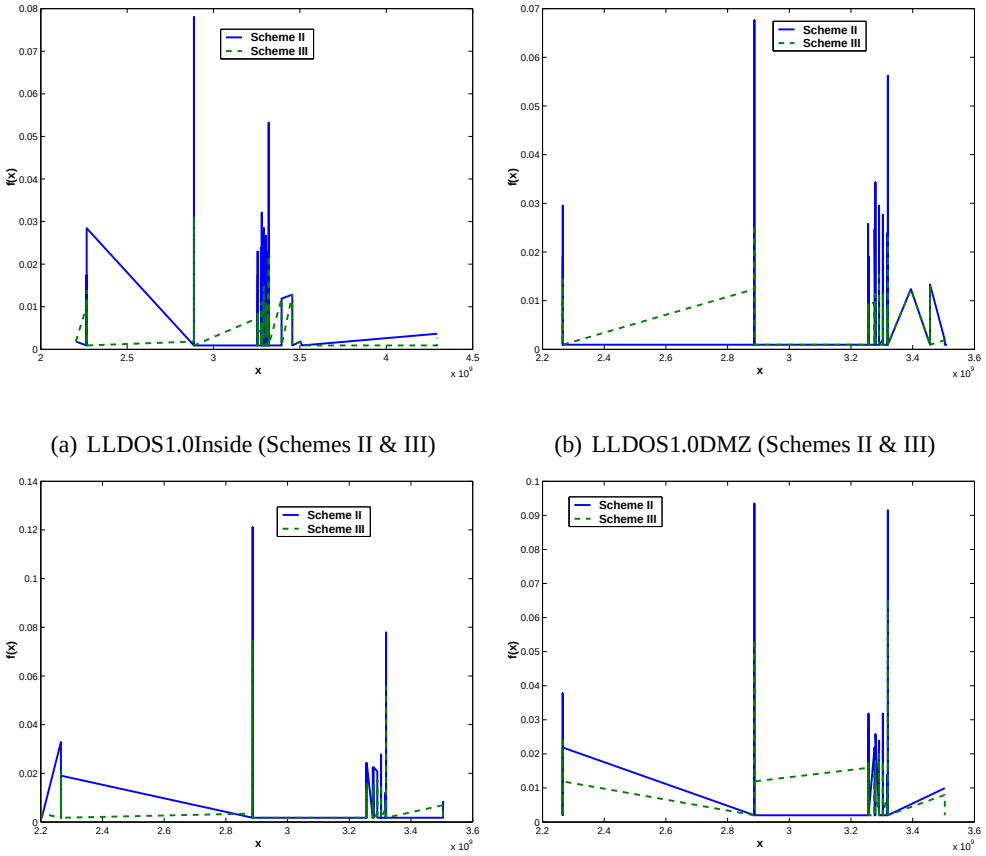
	LLDOS1.0 Inside	LLDOS1.0 DMZ	LLDOS2.0.2 Inside	LLDOS2.0.2 DMZ
# original alerts	922	886	489	425
# artificial alerts	168	164	89	78
$\frac{\text{\# artificial alerts}}{\text{\# alerts in mixed set}}$	15.41%	15.62%	15.40%	15.51%
Local privacy in S_o	0	0	0	0
Local privacy in S_m (Scheme I)	0.620	0.625	0.620	0.623
Local privacy in S_m (Scheme II)	7.387	7.376	7.388	7.382
Local privacy in S_m (Scheme III)	7.387	7.376	7.388	7.382
Global privacy for S_o	4.696	4.845	4.461	4.519
Global privacy for S_m (Scheme I)	5.692	5.806	5.372	5.383
Global privacy for S_m (Scheme II)	5.672	5.792	5.360	5.363
Global privacy for S_m (Scheme III)	6.955	7.041	6.097	6.033

changed after randomization. To further precisely evaluate alert privacy, we also calculated local and global privacy for mixed alert sets. Table 8.2 shows the results. From this table, we observed that local privacy has been significantly increased, which is highly desirable, and global privacy stays almost the same, which results from consistency keeping during randomization.

8.3.3 Experiments on Scheme III

In this set of experiments, we applied Scheme III to all four mixed alert sets. We set time interval $\mathcal{I} = 1$ hour to partition the alert sets, and we got 4 subsets in LLDOS 1.0 Inside data set, 4 subsets in LLDOS 1.0 DMZ data set, 2 subsets in LLDOS 2.0.2 Inside data set, and 2 subsets in LLDOS 2.0.2 DMZ data set. In each subset, we randomized destination IP addresses to their 256-peers. Attribute *DestIP* distributions are shown in Figures 8.5(a), 8.5(b), 8.5(c), and 8.5(d) (dashed green lines).

Compared with Scheme II, we noticed that attribute distributions after applying Scheme III have been further changed. We also computed local and global privacy values, and listed them in Table 8.2. We observed that local privacy stays the same, and global privacy further increases, which results from independent randomization in each subset.



(a) LLDOS1.0Inside (Schemes II & III)

(b) LLDOS1.0DMZ (Schemes II & III)

(c) LLDOS2.0.2Inside (Schemes II & III)

(d) LLDOS2.0.2DMZ (Schemes II & III)

Figure 8.5: PMFs after applying Schemes II and III

8.3.4 Experiments on Similarity Estimation

To examine data utility after applying our anonymization techniques, we evaluate similarity estimation for anonymized data sets in this subsection.

Similarity estimation is based on our discussion on Subsection 8.2.1. For original data sets, if two attribute values are the same, we set their similarity to 1; otherwise their similarity is 0. Next we applied three anonymization schemes independently, where we chose similar settings as in Subsections 8.3.1, 8.3.2, 8.3.3 (the only difference is that here we applied Schemes II and III to original alert sets instead of mixed alert sets). For the data sets after applying Scheme I, we measure attribute similarity using the function for original sets. And for the data sets after applying Schemes

Table 8.3: Correct classification rate and misclassification rate

		LLDOS1.0 Inside	LLDOS1.0 DMZ	LLDOS2.0.2 Inside	LLDOS2.0.2 DMZ
# all pairs		$424,581$	$392,055$	$119,316$	$90,100$
# similar pairs in original set		$18,540$	$14,558$	$6,785$	$4,485$
Scheme I	# similar pairs	$18,664$	$14,803$	$6,832$	$4,548$
	R_{cc}	100%	100%	100%	100%
	R_{mc}	0.0305%	0.0649%	0.0418%	0.0736%
Scheme II	# similar pairs	$18,540$	$14,558$	$6,785$	$4,485$
	R_{cc}	100%	100%	100%	100%
	R_{mc}	0%	0%	0%	0%
Scheme III	# similar pairs	$59,195$	$45,764$	$13,732$	$8,183$
	R_{cc}	100%	100%	100%	100%
	R_{mc}	10.01%	8.27%	6.58%	4.32%

II and III, we used the function in Subsection 8.2.1 to estimate their (lower-bound) similarity values. Next, similar as done in Chapter 7, we also used *correct classification rate* R_{cc} and *misclassification rate* R_{mc} to measure the effectiveness of similarity estimation. Given two attribute values, if their similarity value is greater than 0, we call them "*similar" pair*. Assume the alert sets before and after anonymization are S and S_r, respectively, then $R_{cc} = \frac{\text{\# common similar pairs in } S \text{ and } S_r}{\text{\# similar pairs in } S}$, and $R_{mc} = \frac{\text{\# similar pairs in } S_r - \text{\# common similar pairs in } S \text{ and } S_r}{\text{\# total pairs} - \text{\# similar pairs in } S}$. The results are shown in Table 8.3.

From Table 8.3, we observed that correct classification rate is 100%, and misclassification rate is low (the maximum is around 10%). We also noticed that the results from Scheme II are very desirable. These results tell us that the data utility is still significantly preserved after performing our anonymization techniques.

8.3.5 Experiments on Building Attack Scenarios

In this subsection, we evaluate the effectiveness of building attack scenarios when alerts are anonymized. We did experiments using all four data sets.

In the first set of experiments, we identified all possibly-prepare-for relations and built correlation graphs. The information about prerequisites and consequences for alert types, as well as implication relationships can be found in Tables 5.3 and 5.4. For demonstration purpose, Figure 8.6 shows a correlation graph from LLDOS 1.0 Inside data set (after performing Scheme I and then Scheme II).

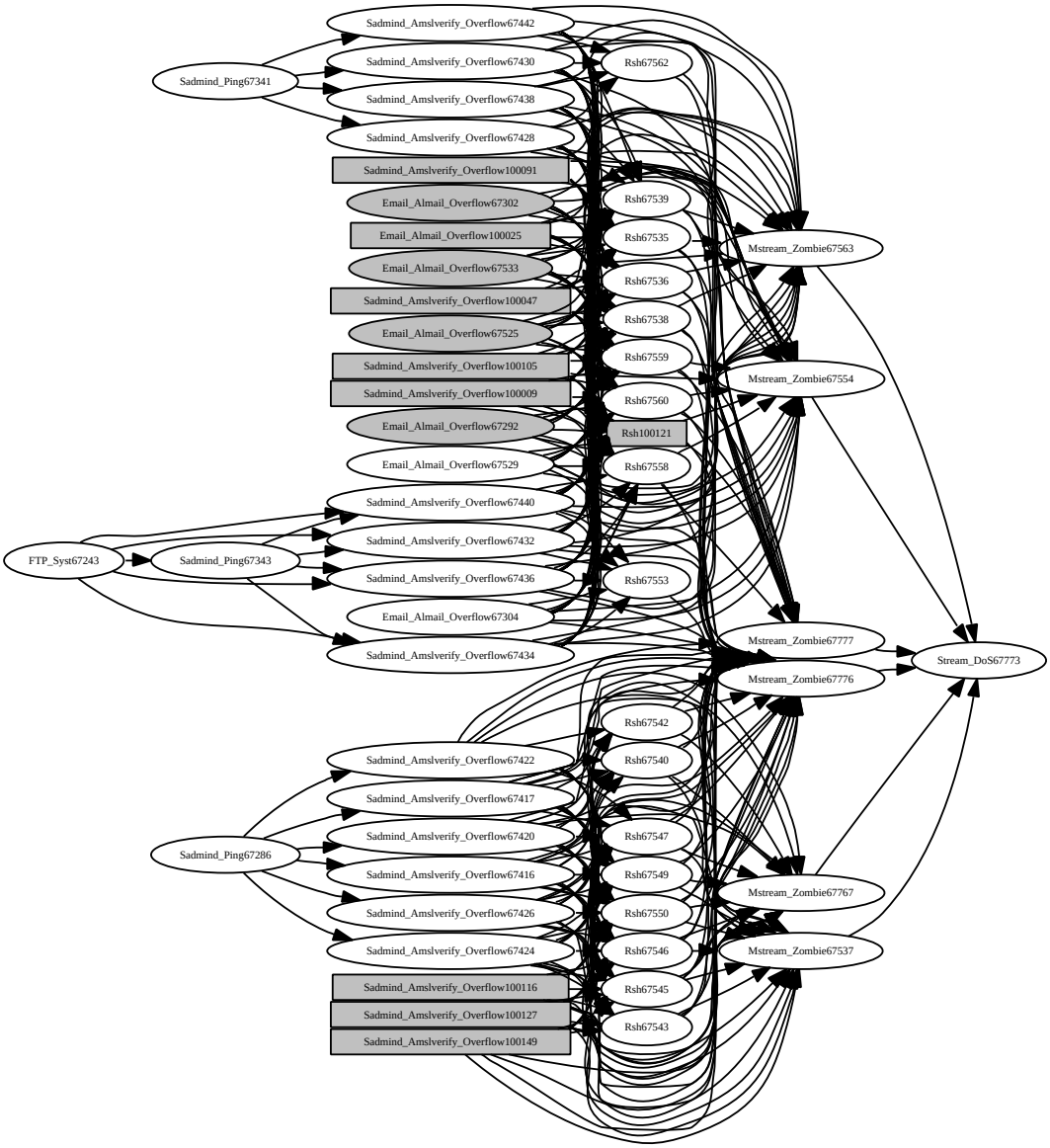


Figure 8.6: A correlation graph in LLDOS 1.0 Inside data set

In Figure 8.6, the string inside each node is the alert type followed by an alert ID. For comparison purpose, we also built the alert correlation graph from the corresponding original data set, where the nodes in this scenario are those without gray-color marking in Figure 8.6. For all those gray nodes, the ellipse nodes are in original data sets, while the rectangle nodes are artificial

Table 8.4: Recall and precision measures in our experiments

		LLDOS 1.0		LLDOS 2.0.2	
		Inside	DMZ	Inside	DMZ
RealSecure	# alerts	922	886	489	425
Correlation for original datasets	# alerts	44	57	13	5
Correlation for anonymized datasets	# original alerts	48	61	20	5
	# artificial alerts	9	3	2	0
RealSecure	Recall M_r	61.67%	57.30%	80.00%	57.14%
Correlation for original datasets	Recall M_r	60.00%	56.18%	66.67%	42.86%
Correlation for anonymized datasets	RecallM_r	60.00%	56.18%	66.67%	42.86%
RealSecure	Precision M_p	4.77%	6.43%	3.27%	1.41%
Correlation for original datasets	Precision M_p	93.18%	94.74%	76.92%	60.00%
Correlation for anonymized datasets	Precision M_p	77.19%	84.38%	45.45%	60.00%

alerts. The attack scenario in Figure 8.6 tells us that attackers probed the vulnerable service using *Sadmind_Ping*, compromised *Sadmind* services using *Sadmind_Amslverify_Overflow* attacks, used *Rsh* to start *mstream* master and zombies, let *mstream* master and zombies communicate with each other (*Mstream_Zombie*), and finally launched *Stream_DoS* attacks, which is consistent with the real attack scenario involved in this data set.

In Figure 8.6, we observed that artificial alerts (e.g., *Sadmind_Amslverify_Overflow100091*) and false alerts (e.g., *Email_Almail_Overflow67302*) may be involved in alert correlation graphs. To further measure the quality of alert correlation graphs, we computed *recall* M_r and *precision* M_p measures[3] for all four data sets, where we let $M_r = \frac{\text{\# detected attacks}}{\text{\# total attacks in the real scenario}}$, and $M_p = \frac{\text{\# true alerts}}{\text{\# alerts}}$. The results are shown in Table 8.4.

In Table 8.4, the number of alerts related to correlation methods are the number of alerts in alert correlation graphs. From Table 8.4, we observed that artificial alerts may or may not be included into the alert correlation graphs. This tells us that attackers cannot use alert correlation graphs to distinguish between original and artificial alerts, which is desirable for privacy protection. We also noticed that the correlation methods have slightly lower recall measures compared with RealSecure network sensor, but much higher precision measures. In addition, the measures related to the correlation method for anonymized data sets are lower than those related to the correlation method for original data sets. This is reasonable because original data sets are more precise than

[3]Recall and precision are basic measures in information-retrieval field.

anonymized data sets.

We also did experiments to polish the alert correlation graphs through examining the (lower-bound) probability of each edge. In our experiments, we set probability threshold to $\frac{1}{256}$. For demonstration purpose, here we discuss the result on polishing the alert correlation graph in Figure 8.6. After probability polishing, the number of nodes in the resulting graphs reduced from 57 to 45. We noticed that probability based polishing can help us remove false prepare-for relations, which may further filter out false alerts (e.g., *Email_Almail_Overflow67292*) and artificial alerts (e.g., *Sadmind_Amslverify_Overflow100009*). However, true prepare-for relations also have a chance to be ruled out (e.g., the prepare-for relation between *Rsh67542* and *Mstream_Zombie67777*), which is not desirable. So it is always helpful to examine both alert correlation graphs with and without probability based pruning to learn attackers' activities.

8.4 Summary

To protect the privacy of intrusion alert data sets, in this chapter we propose three perturbation based schemes to anonymize sensitive attributes of alerts. Our techniques include injecting artificial alerts into original data sets, applying attribute randomization, and partitioning data sets and then performing randomization. To examine the utility of anonymized alerts, we use probability based method to estimate attribute similarity and build attack scenarios. We also use various measures such as correct classification rate to measure the utility of anonymized data sets. Our experimental results demonstrated the usefulness of our techniques. Though our experiments mainly focused on 2000 DARPA intrusion detection scenario specific data sets and we used a simple attribute anonymization policy, we would expect some observations are also useful to other data sets, for examples, the attack scenarios constructed from our probability based approach (without pruning) are supergraphs of the ones constructed from original data sets, probability-based pruning may filter out both false and true prepare-for relations, and for similarity estimation, we cannot always expect 0% misclassification rate. We also notice that to apply our techniques, some expert knowledge is necessary, for example, deciding sensitive attributes, choosing the desirable number of peers ($\mathcal{L}$) or desirable entropy values, and designing concept hierarchies with the consideration of the desirable number of peers or entropy values.

There are several directions worth further investigation. One is additional techniques to perform alert anonymization. Our techniques on injecting artificial alerts may introduce additional

overhead on alert correlation analysis. Combining our techniques with other complementary techniques such as hash function based methods [69] is worth additional research. Other directions include, for example, additional alert correlation analysis techniques that can help us understand security threats based on anonymized data sets, and the performance of privacy-preserving alert correlation techniques.

Chapter 9

Conclusion and Future Work

9.1 Conclusion

To defend against various attacks, many security systems such as intrusion detection systems are deployed into hosts and networks to better protect digital assets. These systems flag alerts when suspicious events are monitored. However, there are well-known problems related to the current intrusion detection systems: (1) they may flag thousands of alerts per day, thus overwhelming the security officers, (2) among all the alerts, true positives are mixed with false positives, and it is usually difficult to differentiate between them, and (3) existing intrusion detection systems cannot detect all attacks. These challenges make manually analyzing the alerts from multiple security systems time-consuming and error-prone. To better understand security threats from various sources and take appropriate response, it is necessary to perform alert correlation.

My dissertation focuses on correlation analysis of intrusion alerts. In particular, I have studied the following problems.

Efficiency of Intrusion Alert Correlation. This is an extended work to our previous correlation method [82, 83]. The initial implementation of [83] is a Database Management System (DBMS) based toolkit, which have been shown to be effective through our experiments. However, our experience also shows relying entirely on DBMS introduces unacceptable performance penalty, especially for interactive analysis of intensive alerts.

To address the performance problem, we adapt main memory index structures (e.g., B

Trees, T Trees, Linear Hashing) and database query optimization techniques (e.g., nested loop join, sort join) to facilitate timely correlation of intensive alerts. By taking advantage of the characteristics of the alert correlation process, we present three techniques named *hyper-alert container*, *two-level index*, and *sort correlation*. The performance of these techniques is studied through a series of experiments. The experimental results demonstrate that (1) hyper-alert containers improve the efficiency of order-preserving index structures, with which an insertion operation involves search (e.g., Array Binary Search, T Trees), (2) two-level index improves the efficiency of all index structures, (3) a two-level index structure combining Chained Bucket Hashing and Linear Hashing is the most efficient for streamed alerts, (4) sort correlation with heap sort algorithm is the most efficient for alert correlation in batch, (5) two-level Linear Hashing is the most efficient for alert correlation when sliding window is used to cope with memory constraint.

Learning Attack Strategies. We notice that understanding the strategies of attacks is crucial for security applications such as computer and network forensics, intrusion response, and prevention of future attacks. We present techniques to automatically learn attack strategies from intrusion alerts. The essence of this approach is a model that represents an attack strategy as a graph of attacks with constraints on the attack attributes and the temporal order among these attacks. To learn the intrusion strategy is to extract such a graph from a sequence of intrusion alerts. To further facilitate the analysis of attack strategies, we present techniques to measure the similarity between attack strategies. The basic idea is to reduce the similarity measurement of attack strategies into error-tolerant graph isomorphism problem and measure the similarity between attack strategies in terms of the cost to transform one strategy into another.

Hypothesizing and Reasoning about Attacks Missed by Intrusion Detection Systems. Though many alert correlation methods have been proposed in recent years, we observe that all of these methods depend heavily on the underlying IDSs, and cannot deal with the attacks missed by IDSs. In order to reduce the impact of missed attacks, we present a series of techniques to hypothesize and reason about attacks possibly missed by the IDSs. In addition, we also discuss techniques to infer attribute values for hypothesized attacks, to validate hypothesized attacks through raw audit data, and to consolidate hypothesized attacks to generate concise attack scenarios.

Intrusion Alert Correlation Based on Triggering Events and Common Resources. We notice that complementary security systems are widely deployed in networks to better protect digital assets. To analyze the alerts from different systems, we propose a correlation approach based on triggering events and common resources. One of the key concepts in our approach is triggering events, which are the (low-level) events that trigger alerts. By grouping the alerts sharing “similar”

triggering events, a set of alerts can be partitioned into different clusters such that the alerts in the same cluster may correspond to the same attack. Our approach further examines whether the alerts in each cluster are *consistent* with relevant network and host configurations, which help analysts to partially identify the severity of alerts and clusters. The other key concept in our approach is input and output resources, where *input resources* are the necessary resources for an attack to succeed, and *output resources* are the resources that an attack supplies if successful. We propose to model each attack through specifying input and output resources. By identifying the "common" resources between output resources of one attack and input resources of another, it discovers causal relationships between alert clusters and builds attack scenarios.

Privacy-Privacy Alert Correlation through Generalization. With the increasing security threats from infrastructure attacks such as worms and distributed denial of service attacks, it is clear that the cooperation among different organizations is necessary to defend against these attacks. However, organizations' privacy concerns for the incident and security alert data require that sensitive data be sanitized before they are shared with other organizations. Such sanitization process usually has negative impacts on intrusion analysis (such as alert correlation). To balance the privacy requirements and the need for intrusion analysis, in Chapter 7 we propose a privacy-preserving alert correlation approach through generalization based on concept hierarchies. Our approach consists of two phases. The first phase is *entropy guided alert sanitization*, where sensitive alert attributes are generalized to high-level concepts to introduce uncertainty into the dataset with partial semantics. To balance the privacy and the usability of alert data, we propose to guide the alert sanitization process with the entropy or differential entropy of sanitized attributes. The second phase is *sanitized alert correlation*. We focus on defining similarity functions between sanitized attributes and building attack scenarios from sanitized alerts.

Privacy-Preserving Alert Correlation through Perturbation. Intrusion alert data sets are critical for security research such as alert correlation. However, privacy concerns about the data sets from different data owners may prevent data sharing and investigation. It is always desirable and sometimes mandatory to anonymize sensitive data in alert sets before they are shared and analyzed. To address privacy concerns, in Chapter 8 we propose three perturbation based schemes to flexibly perform alert anonymization. These schemes are closely related but can also be applied independently. In Scheme I, we generate artificial alerts and mix them with original alerts to help hide original attribute values. In Scheme II, we further map sensitive attributes to random values based on concept hierarchies. In Scheme III, we propose to partition an alert set into multiple subsets and apply Scheme II in each subset independently. To evaluate privacy protection and guide alert

anonymization, we define *local privacy* and *global privacy*, and use *entropy* to compute their values. Though we emphasize alert anonymization techniques in Chapter 8, to examine the utility of data sets, we further perform correlation analysis for anonymized data sets. Similar as Chapter 7, We focus on computing similarity values between anonymized attributes and building attack scenarios from anonymized data sets.

9.2 Future Work

Though we have addressed a few problems in intrusion alert correlation and intrusion detection, there are many problems that have not been fully addressed. In the following, we list two directions worth further investigation.

- *Distributed intrusion alert correlation*. To protect the cyber security of an enterprise, an organization, or an institution with large numbers of computers and networks, security systems such as intrusion detection systems are usually deployed into many different places. To effectively learn the security threats, security officers need to perform correlation analysis for the alerts from those different places. Thus, how do we deploy multiple different security systems? How do we perform alert correlation in order to learn the local as well as the global security threats in a timely and effective fashion? These problems are challenging and have not been fully addressed.

- *Additional techniques for privacy-preserving alert correlation*. To defend against the infrastructure attacks such as worms and distributed denial of service (DDoS) attacks, it is clear that the cooperation between different organizations is necessary. However, different organizations, companies and individuals are not willing to share attack related data unless the sensitive information in the data sets are anonymized. In Chapters 7 and 8, we propose two complementary approaches to anonymize sensitive alert attributes (based on concept hierarchies), where in Chapter 7, we propose to generalize sensitive attributes to high-level concepts, and in Chapter 8, we propose to use concept hierarchies to facilitate artificial alert generation and attribute randomization. These approaches can preserve the privacy of alert data. However, since generalized attribute values usually may take different formats compared with original values (they usually have different attribute domains), the approach in Chapter 7 may let malicious users realize that attributes in alert sets are sanitized, which

may infer organizations' privacy policy. Artificial alert injection in Chapter 8 may introduce more overhead on correlation analysis of anonymized alerts. Moreover, both approaches in Chapters 7 and 8 may introduce false causal relations. Thus, additional, complementary alert anonymization techniques that can not only protect the privacy of data sets, but also generate useful results after correlation analysis are worth future research.

Bibliography

[1] N. Adam and J. Wortmann. Security-control methods for statistical databases: A comparison study. *ACM Computing Surveys*, 21(4):515–556, 1989.

[2] D. Agrawal and C. Aggarwal. On the design and quantification of privacy-preserving data mining algorithms. In *Proceedings of the 20th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, May 2001.

[3] R. Agrawal and R. Srikant. Privacy-preserving data mining. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, May 2000.

[4] A. Aho, J. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[5] A. Ammann, M. Hanrahan, and R. Krishnamurthy. Design of a memory resident DBMS. In *Proceedings of IEEE COMPCON*, San Francisco, February 1985.

[6] P. Ammann, D. Wijesekera, and S. Kaushik. Scalable, graph-based network vulnerability analysis. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 217–224, November 2002.

[7] E. Amoroso. *Intrusion Detection: An Introduction to Internet Surveillance, Correlation, Trace Back, Traps, and Response*. Intrusion.Net Books, 1999.

[8] J. P. Anderson. Computer security threat monitoring and surveillance. Technical report, James P. Anderson Co., Fort Washington, PA, 1980.

[9] AT & T Research Labs. Graphviz - open source graph layout and drawing software. `http://www.research.att.com/sw/tools/graphviz/`.

[10] S. Axelsson. Research in intrusion-detection systems: A survey. Technical Report TR 98-17, Department of Computer Engineering, Chalmers University of Technology, Goteborg, Sweden, 1999.

[11] S. Axelsson. The base-rate fallacy and the difficulty of intrusion detection. *ACM Transactions on Information and System Security*, 3(3):186–205, August 2000.

[12] D. Barbará, Cuotuo J., S. Jajodia, and N. Wu. ADAM: A testbed for exploring the use of data mining in intrusion detection. *ACM SIGMOD Record*, 30(4):15–24, December 2001.

[13] D. Barbará, N. Wu, and S. Jajodia. Detecting novel network intrusion using bayes estimators. In *Proceedings of the First SIAM Conference on Data Mining*, April 2001.

[14] M. Bellare, R. Canetti, and H. Krawczyk. Message authentication using hash function - the HMAC construction. *RSA Laboratories' CryptoBytes*, 2(1):12–15, 1996.

[15] H. Bunke and K. Shearer. A graph distance metric based on the maximal common subgraph. *Pattern Recognition Letters*, 19(3-4):255–259, 1998.

[16] Brian Caswell and Marty Roesch. Snort: The open source network intrusion detection system. `http://www.snort.org`.

[17] CERT Coordination Center. CERT Coordination Center. `http://www.cert.org`.

[18] CERT Coordination Center. Cert advisory CA-2001-10 buffer overflow vulnerability in microsoft IIS 5.0. `http://www.cert.org/advisories/CA-2001-10.html`, 2001.

[19] CERT Coordination Center. Overview of attack trends. `http://www.cert.org/archive/pdf/attack_trends.pdf`, 2002.

[20] CERT Coordinate Center. Overview of attack trends. `http://www.cert.org/archive/pdf/attack_trends.pdf`, 2002.

[21] D. Comer. The ubiquitous B-Tree. *ACM Computeing Surveys*, 11(2):121–137, 1979.

[22] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1989.

[23] Microsoft Corporation. Microsoft security bulletin (ms00-029). http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/bulletin/MS00-029.asp, 2000.

[24] Microsoft Corporation. Microsoft security bulletin (ms00-078). http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/bulletin/MS00-078.asp, 2000.

[25] Microsoft Corporation. Microsoft security bulletin (ms01-023). http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/bulletin/MS01-023.asp, 2001.

[26] T. Cover and J. Thomas. *Elements of Information Theory*. John Wiley & Sons, Inc., 1991.

[27] Y. Cui. A toolkit for intrusion alerts correlation based on prerequisites and consequences of attacks. Master's thesis, North Carolina State University, December 2002. Available at http://www.lib.ncsu.edu/theses/available/etd-12052002-193803/.

[28] F. Cuppens. Managing alerts in a multi-intrusion detection environment. In *Proceedings of the 17th Annual Computer Security Applications Conference*, December 2001.

[29] F. Cuppens and A. Miege. Alert correlation in a cooperative intrusion detection framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, May 2002.

[30] F. Cuppens and R. Ortalo. LAMBDA: A language to model a database for detection of attacks. In *Proc. of Recent Advances in Intrusion Detection (RAID 2000)*, pages 197–216, September 2000.

[31] D. Curry and H. Debar. Intrusion detection message exchange format data model and extensible markup language (xml) document type definition. Internet Draft, draft-ietf-idwg-idmef-xml-03.txt, February 2001.

[32] W. Dai. Speed comparison of popular crypto algorithms. http://www.eskimo.com/~weidai/benchmarks.html.

[33] O. Dain and R.K. Cunningham. Building scenarios from a heterogeneous alert stream. In *Proceedings of the 2001 IEEE Workshop on Information Assurance and Security*, pages 231–235, June 2001.

[34] O. Dain and R.K. Cunningham. Fusing a heterogeneous alert stream into scenarios. In *Proceedings of the 2001 ACM Workshop on Data Mining for Security Applications*, pages 1–13, November 2001.

[35] DARPA Cyber Panel Program. DARPA cyber panel program grand challenge problem. `http://www.grandchallengeproblem.net/`, 2003.

[36] H. Debar and A. Wespi. Aggregation and correlation of intrusion-detection alerts. In *Recent Advances in Intrusion Detection*, LNCS 2212, pages 85 – 103, 2001.

[37] DEFCON. Def con capture the flag (CTF) contest. http://www.defcon.org/html/defcon-8-post.html, July 2000. Archive accessible at http://wi2600.org/mediawhore/mirrors/shmoo/.

[38] DEFCON. Def con capture the flag (CTF) contest. http://www.defcon.org/html/defcon-9/defcon-9-pre.html, July 2001.

[39] D. Eastlake and P. Jones. US secure hash algorithm 1 (sha1). Request for Comments: (RFC) 3174, September 2001.

[40] S.T. Eckmann, G. Vigna, and R.A. Kemmerer. STATL: An Attack Language for State-based Intrusion Detection. *Journal of Computer Security*, 10(1/2):71–104, 2002.

[41] H. Feng, J. Giffin, Y. Huang, S. Jha, W. Lee, and B. Miller. Formalizingsensitivity in static analysis for intrusion detection. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy (S&P'04)*, May 2004.

[42] H. Feng, O. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy (S&P'03)*, May 2003.

[43] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff. A sense of self for unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, May 1996.

[44] Fyodor. Nmap free security scanner. `http://www.insecure.org/nmap`, 2003.

[45] H. Garcia-Molina and J. Widom J. D. Ullman. *Database System Implementation*. Prentice Hall, 2000.

[46] M. R. Gary and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman and Company, 1979.

[47] J. Giffin, S. Jha, and B. Miller. Detecting manipulated remote call streams. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.

[48] J. Haines, D. Ryder, L. Tinnel, and S. Taylor. Validation of sensor alert correlators. *IEEE Security & Privacy Magazine*, 1(1):46–56, 2003.

[49] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, 2001.

[50] K. Ilgun. USTAT: A real-time intrusion detection system for UNIX. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 16–28, Oakland, CA, May 1993.

[51] RTI International. PREDICT - Protected Repository for the Defense of Infrastructure Against Cyber Threats. `http://www.predict.org`.

[52] Internet Security Systems. RealSecure intrusion detection system. `http://www.iss.net`.

[53] Internet Security Systems, Inc. REALSECURE signatures reference guide. `http://www.iss.net/`.

[54] D. A. Jackson, K. M. Somers, and H. H. Harvey. Similarity coefficients: Measures of co-occurence and association or simply measures of occurrence? *The American Naturalist*, 133(3):436–453, March 1989.

[55] A.K. Jain and R.C. Dubes. *Algorithms for Clustering Data*. Prentice Hall, 1988.

[56] H. S. Javitz and A. Valdes. The SRI IDES statistical anomaly detector. In *Proceedings IEEE Symposium on Security and Privacy*, pages 316–326, Oakland, CA, May 1991.

[57] H. S. Javitz and A. Valdes. The NIDES statistical component: Description and justification. Technical report, SRI International, March 1994.

[58] S. Jha, O. Sheyner, and J.M. Wing. Two formal analyses of attack graphs. In *Proceedings of the 15th Computer Security Foundation Workshop*, June 2002.

[59] K. Julisch. Dealing with false positives in intrusion detection. In *The 3th Workshop on Recent Advances in Intrusion Detection*, October 2000.

[60] K. Julisch. Mining alarm clusters to improve alarm handling efficiency. In *Proceedings of the 17th Annual Computer Security Applications Conference (ACSAC)*, pages 12–21, December 2001.

[61] K. Julisch. Clustering intrusion detection alarms to support root cause analysis. *ACM Transactions on Information and System Security*, 6(4):443–471, Nov 2003.

[62] K. Julisch and M. Dacier. Mining intrusion detection alarms for actionable knowledge. In *The 8th ACM International Conference on Knowledge Discovery and Data Mining*, July 2002.

[63] L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley and Sons, 1990.

[64] D. Knuth. *The Art of Computer Programming*. Addison-Wesley, 1973.

[65] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for message authentication. Internet Engineering Task Force, Request for Comments (RFC) 2104, February 1997.

[66] S. Kummar and E.H. Spafford. A software architecture to support misuse intrusion detection. In *Proceedings of the 18th National Information Systems Security Conference*, pages 194–204, 1995.

[67] T. J. Lehman and M. J. Carey. A study of index structure for main memory database management systems. In *Proceedings of the Twelfth International Conference on Very Large Databases*, pages 294–303, Kyoto, Japan, August 1986.

[68] C. Liew, U. Choi, and C. Liew. A data distortion by probability distribution. *ACM Transactions on Database Systems*, 10(3):395–411, September 1985.

[69] P. Lincoln, P. Porras, and V. Shmatikov. Privacy-preserving sharing and correlation of security alerts. In *Proceedings of 13th USENIX Security Symposium*, August 2004.

[70] W. Litwin. Linear hashing: A new tool for file and table addressing. In *Proceedings of the 6th Conference on Very Large Data Bases*, pages 212–223, Montreal, Canada, October 1980.

[71] Y. Lu. Concept hierarchy in data mining: Specification, generation and implementation. Master's thesis, School of Computing Science, Simon Fraser University, Canada, December 1997.

[72] A. Menezes, P. Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, October 1996.

[73] B. T. Messmer. *Efficient Graph Matching Algorithms for Preprocessed Model Graphs*. PhD thesis, University of Bern, Switzerland, November 1995.

[74] B. T. Messmer and H. Bunke. A new algorithm for error-tolerant subgraph isomorphism detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(5):493–504, 1998.

[75] B. T. Messmer and H. Bunke. Efficient subgraph isomorphism detection: A decomposition approach. *IEEE Transactions on Knowledge and Data Engineering*, 12(2):307–323, 2000.

[76] B.T. Messmer and H. Bunke. A decision tree approach to graph and subgraph isomorphism detection. *Pattern Recognition*, 32(12):1979–1998, 1999.

[77] MIT Lincoln Lab. 2000 DARPA intrusion detection scenario specific datasets. `http://www.ll.mit.edu/IST/ideval/data/2000/2000_data_index.html`, 2000.

[78] B. Morin and H. Debar. Correlation of intrusion symptoms: an application of chronicles. In *Proceedings of the 6th International Conference on Recent Advances in Intrusion Detection (RAID'03)*, September 2003.

[79] B. Morin, L. Mé, H. Debar, and M. Ducassé. M2D2: A formal data model for IDS alert correlation. In *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID 2002)*, pages 115–137, 2002.

[80] A. Mounji, B.L. Charlier, D. Zampuniéris, and N. Habra. Distributed audit trail analysis. In *Proceedings of the ISOC '95 Symposium on Network and Distributed System Security*, pages 102–112, 1995.

[81] National Institute of Standards and Technology (NIST). NIST brief comments on recent cryptanalytic attacks on secure hashing functions and the continued security provided by SHA-1. `http://csrc.nist.gov/hash_standards_comments.pdf`.

[82] P. Ning, Y. Cui, and D. S Reeves. Analyzing intensive intrusion alerts via correlation. In *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID 2002)*, pages 74–94, Zurich, Switzerland, October 2002.

[83] P. Ning, Y. Cui, and D. S Reeves. Constructing attack scenarios through correlation of intrusion alerts. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 245–254, Washington, D.C., November 2002.

[84] P. Ning, Y. Cui, D. S. Reeves, and D. Xu. Tools and techniques for analyzing intrusion alerts. *ACM Transactions on Information and System Security*, 7(2):273–318, May 2004.

[85] P. Ning and S. Jajodia. Intrusion detection techniques. In H. Bidgoli, editor, *Internet Encyclopedia*. John Wiley & Sons, 2003.

[86] P. Ning, D. Xu, C. Healey, and R. St. Amant. Building attack scenarios through integration of complementary alert correlation methods. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS '04)*, pages 97–111, February 2004.

[87] Packet storm. `http://packetstormsecurity.nl`. Accessed on April 30, 2003.

[88] R. Pang and V. Paxson. A high-level programming environment for packet trace anonymization and transformation. In *Proceedings of ACM SIGCOMM 2003*, August 2003.

[89] M. Peuhkuri. A method to compress and anonymize packet traces. In *Proceedings of ACM Internet Measurement Workshop 2001*, November 2001.

[90] P.A. Porras, M.W. Fong, and A. Valdes. A mission-impact-based approach to INFOSEC alarm correlation. In *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID 2002)*, pages 95–114, 2002.

[91] X. Qin and W. Lee. Statistical causality analysis of infosec alert data. In *Proceedings of The 6th International Symposium on Recent Advances in Intrusion Detection (RAID 2003)*, Pittsburgh, PA, September 2003.

[92] S. Reiss. Practical data-swapping: The first steps. *ACM Transactions on Database Systems*, 9(1):20–37, March 1984.

[93] R. Rivest. The MD5 Message-Digest algorithm. Request for Comments: (RFC) 1321, April 1992.

[94] M. Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the 1999 USENIX LISA conference*, 1999.

[95] P. Samarati and L. Sweeney. Protecting privacy when disclosing information: k-anonymity and its enforcement through generalization and suppression. Technical Report SRI-CSL-98-04, Computer Science Laboratory, SRI International, 1998.

[96] R. Sekar, M. Bendre, P. Bollineni, and D. Dhurjati. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, May 2001.

[97] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J.M. Wing. Automated generation and analysis of attack graphs. In *Proceedings of IEEE Symposium on Security and Privacy*, May 2002.

[98] S. Staniford, J.A. Hoagland, and J.M. McAlerney. Practical automated detection of stealthy portscans. *Journal of Computer Security*, 10(1/2):105–136, 2002.

[99] L. Sweeney. Achieving k-anonymity privacy protection using generalization and suppression. *International Journal on Uncertainty, Fuzziness and Knowledge-based Systems*, 10(5):571–588, October 2002.

[100] L. Sweeney. k-anonymity: A model for protecting privacy. *International Journal on Uncertainty, Fuzziness and Knowledge-based Systems*, 10(5):557–570, October 2002.

[101] Symantec Corporation. Symantec's norton antivirus. `http://www.symantec.com`.

[102] S. Templeton and K. Levitt. A requires/provides model for computer attacks. In *Proceedings of New Security Paradigms Workshop*, pages 31 – 38. ACM Press, September 2000.

[103] F. Traub, Y. Yemini, and H. Woźniakowski. The statistical security of a statistical database. *ACM Transactions on Database Systems*, 9(4):672–679, December 1984.

[104] Tripwire, Inc. Tripwire changing monitoring and reporting solutions. `http://www.tripwire.com`.

[105] J. D. Ullman. *Principles of database and knowledge-base systems*, volume 2. Computer Science Press, 1989.

[106] J. Ullrich. DShield - distributed intrusion detection system. `http://www.dshield.org`.

[107] U.S. DEPARTMENT OF COMMERCE/National Institute of Standards and Technology. Data encryption standard (DES). Federal Information Processing Standards Publication 46-3, October 1999.

[108] H. Vaccaro and G. Liepins. Detection of anomalous computer session activity. In *Proceedings of 1989 IEEE Symposium on Security and Privacy*, pages 280–289, Oakland, CA, May 1989.

[109] A. Valdes and K. Skinner. Probabilistic alert correlation. In *Proceedings of the 4th International Symposium on Recent Advances in Intrusion Detection (RAID 2001)*, pages 54–68, 2001.

[110] V. Verykios, E. Bertino, I Fovino, L. Provenza, Y. Saygin, and Y. Theodoridis. State-of-the-art in privacy preserving data mining. *ACM SIGMOD Record*, 33(1):50–57, March 2004.

[111] G. Vigna and R. A. Kemmerer. NetSTAT: A network-based intrusion detection system. *Journal of Computer Security*, 7(1):37–71, 1999.

[112] G. Vigna and R. A. Kermmerer. NetSTAT: A network-based intrusion detection approach. In *Proceedings of the 14th Annual Security Applications Conference*, December 1998.

[113] D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, May 2001.

[114] C. Warrander, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, May 1999.

[115] V. Yegneswaran, P. Barford, and S. Jha. Global intrusion detection in the domino overlay system. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS'04)*, February 2004.

[116] Y. Zhai, P. Ning, P. Iyer, and D.S. Reeves. Reasoning about complementary intrusion evidence. In *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC '04)*, December 2004.

[117] Zone Labs. Zonealarm pro. `http://www.zonelabs.com`.

Appendix

Appendix A

Additional Experimental Results Using TIAA

Our techniques on intrusion alert correlation result in a software package TIAA (A Toolkit for Intrusion Alert Analysis), which is available at our website `http://discovery.csc.ncsu.edu/software/correlator/`. To further evaluate the effectiveness of TIAA, we did more experiments using a data set collected on our campus network and DEF CON 9 CTF event data sets [38]. In this appendix, we show some correlation results.

The first data set was collected from a network in Computer Science Department. We used *Snort* to parse the packets and generate intrusion alerts. The alert collection was carried from June 24, 2005 to June 29, 2005, and we totally got $325,968$ alerts with 29 alert types. In the following, we list some interesting attack scenarios discovered in this data set. (Some correlation graphs are too big, so we may only show part of them.)

Figure A.1 is an MS SQL server related attack. In this scenario, attackers compromised the victim machine through MS SQL Server related vulnerability, and installed some rootkit to maintain access to the victim. This scenario is consistent with our forensic analysis to the compromised system.

We show some other attack scenarios in Figure A.2. Figures A.2(a) and A.2(b) are two scanning based attack scenarios, where attackers may first detect live IP addresses, then further

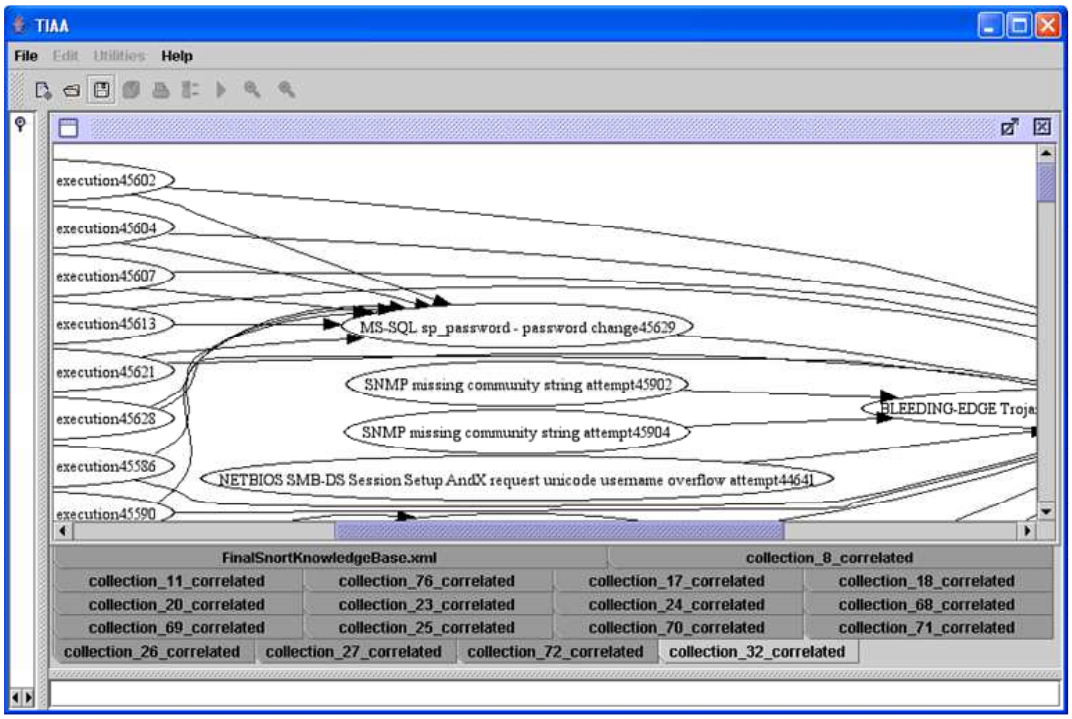
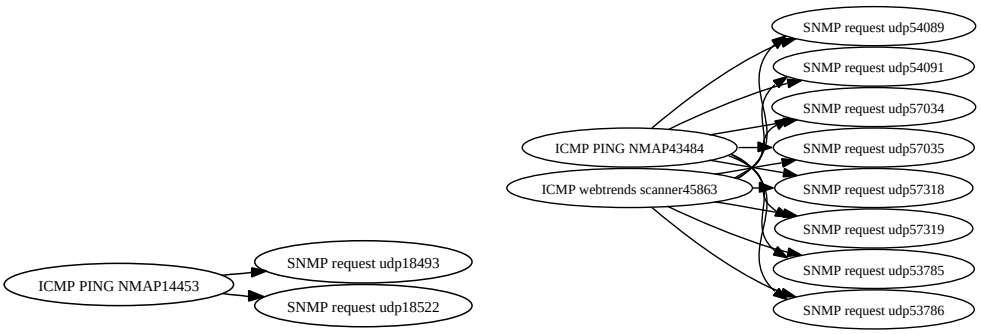


Figure A.1: An MS SQL server related attack scenario in campus collected data set

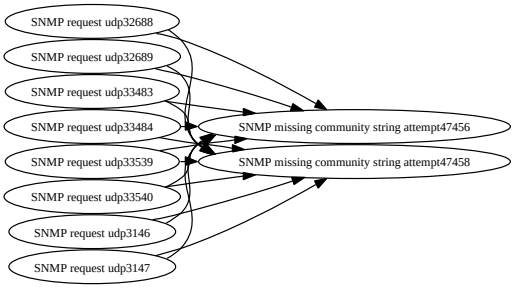
probe services running on live hosts. Figure A.2(c) is an SNMP related attack, where attackers may first detect the hosts that are running SNMP, then gain certain access to the hosts through the mis-configuration of the authentication in SNMP.

We also did experiments using DEF CON 9 CTF event data set. In the following, we list some attack scenarios discovered in the data set.

The first scenario we present is related to Figure A.3. Notice that there are 192 alerts and 12205 prepare-for relations involved in this scenario. To visualize it, we first performed aggregation to this scenario. The result is shown in Figure A.3(a). In Figure A.3(a), we observed that attackers may try different web based attacks to gain unauthorized access to victim hosts, and then ran commands to further attack victims. To further learn this attack scenario, we performed association analysis using TIAA. Specifically, we listed the attribute values that frequently occur in this scenario. The result is in Figure A.3(b). From Figure A.3(b), we know, for example, all alerts in this scenario have source IP address 10.255.10.34 and destination IP address 10.255.30.252, and 84.375% of the total alerts have destination port number 80, which is useful for us to learn attackers'



(a) A scanning based attack scenario (b) A scanning based attack scenario

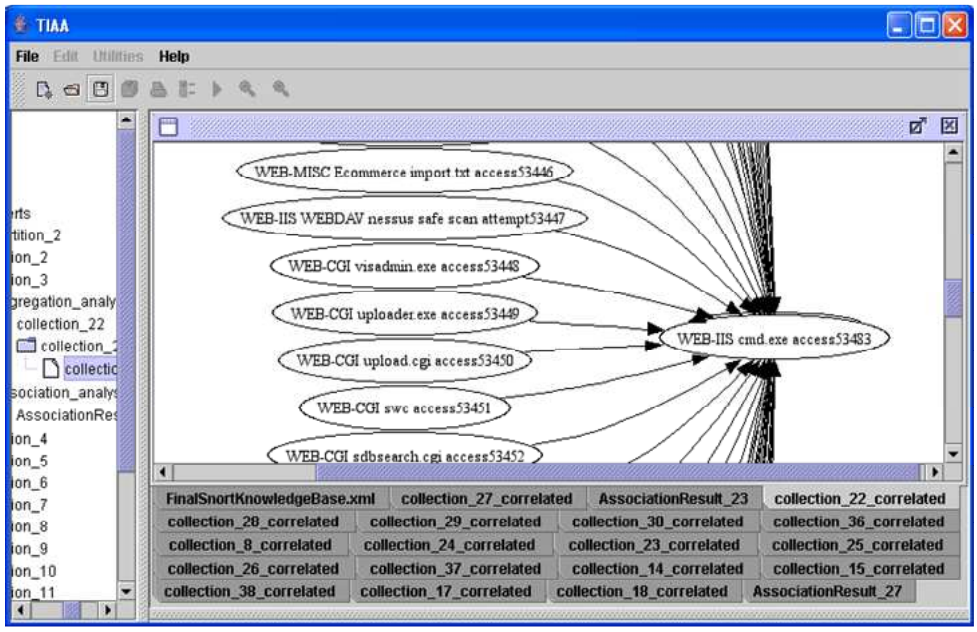


(c) An SNMP related attack scenario

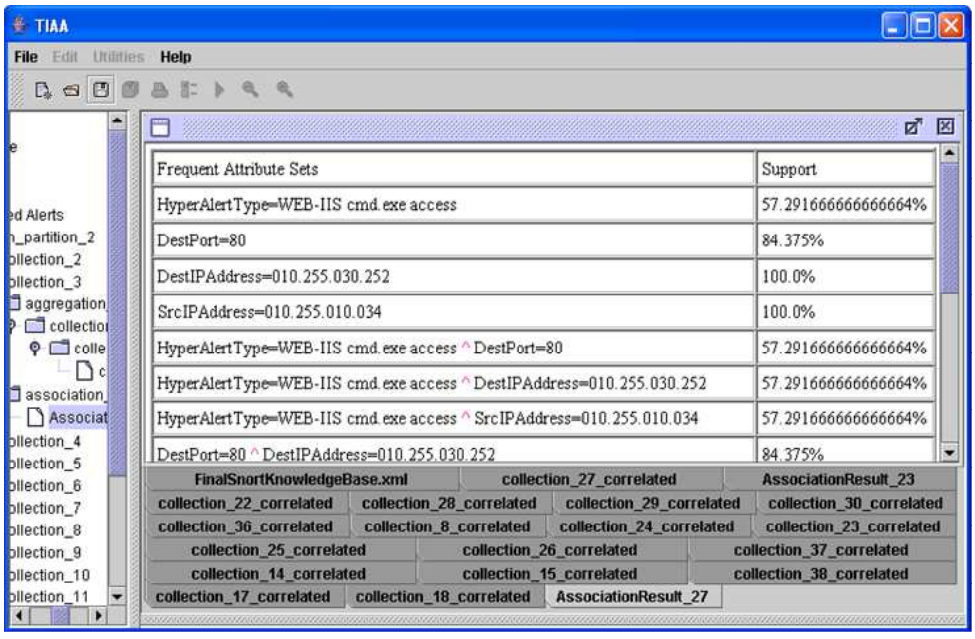
Figure A.2: Some attack scenarios in campus collected data set

activities.

The second scenario we present is related to Figure A.4. There are 6317 alerts and 185673 prepare-for relations in this scenario. Similar as done in Figure A.3, we also performed aggregation to this scenario. The aggregated attack scenario is shown in Figure A.4(a). From this scenario, we observed that attackers scanned live IP addresses, probed network services through various means (e.g., *SCAN FIN*), gained unauthorized access to victim hosts through various web based attacks, and ran some commands on the victims. This scenario is consistent with our intuition about how attackers may launch multi-phase attacks. Similar as in Figure A.3, we also performed association analysis in this scenario. The result is shown in Figure A.4(b). It tells us, for example, the major source IP addresses in this scenario is 10.255.0.213 and 10.255.0.253, the major destination IP address is 10.255.10.34, etc.

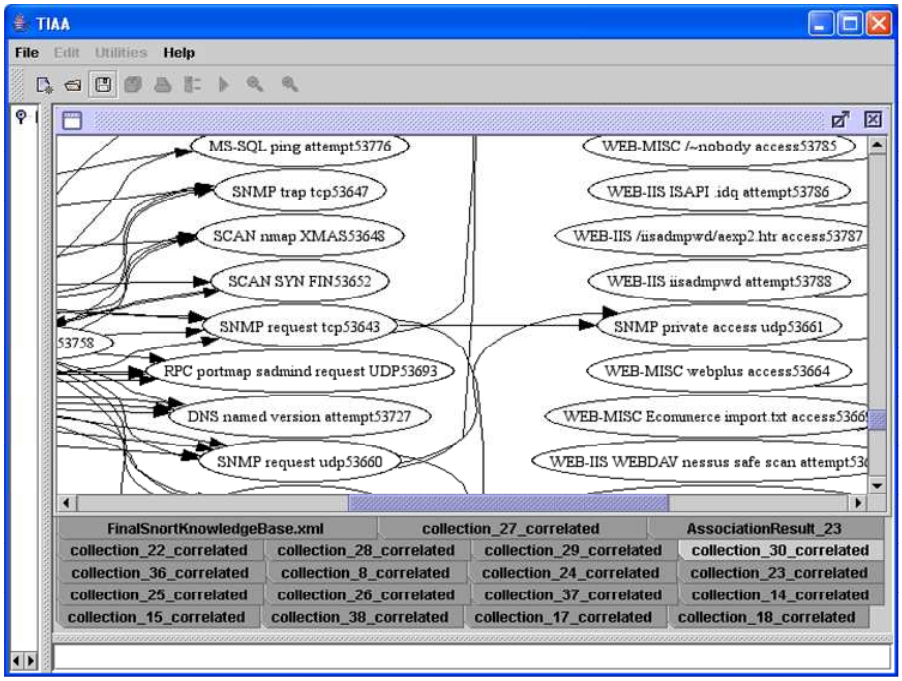


(a) An attack scenario (aggregated)

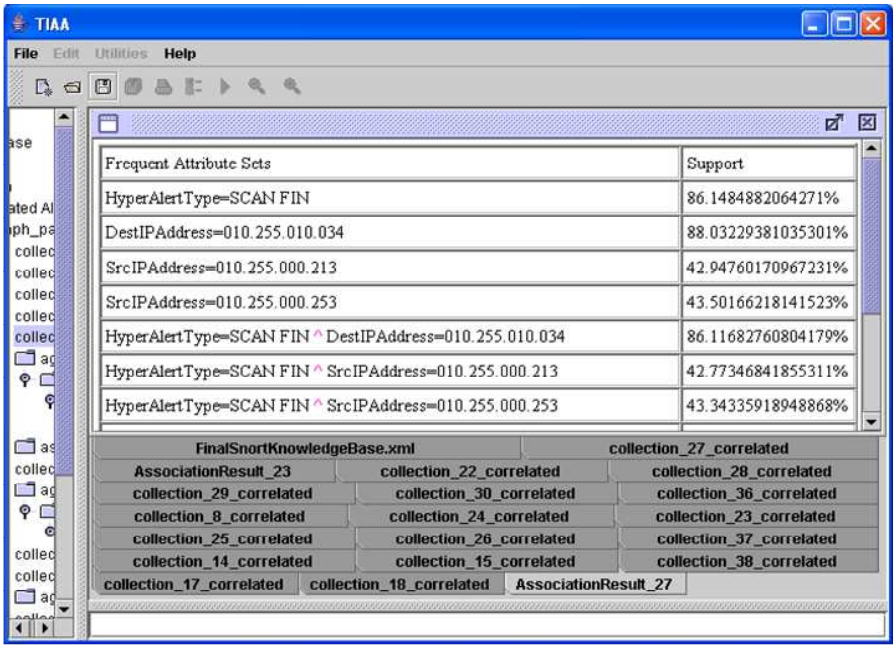


(b) Association analysis of this scenario

Figure A.3: One attack scenario in DEF CON 9 data set



(a) An attack scenario (aggregated)



(b) Association analysis of this scenario

Figure A.4: Another attack scenario in DEF CON 9 data set