

## Abstract

CUI, YUN. A Toolkit for Intrusion Alerts Correlation Based on Prerequisites and Consequences of Attacks. (Under the direction of Dr. Peng Ning.)

Intrusion Detection has been studied for about twenty years. Intrusion Detection Systems (IDSs) are usually considered the second line of defense to protect against malicious activities along with the prevention-based security mechanisms such as authentication and access control. However, traditional IDSs have two major weaknesses. First, they usually focus on low-level attacks or anomalies, and raise alerts independently, though there may be logical connections between them. Second, there are a lot of false alerts reported by traditional IDSs, which are mixed with true alerts. Thus, the intrusion analysts or the system administrators are often overwhelmed by the volume of alerts.

Motivated by this observation, we propose a technique to construct high-level attack scenarios by correlating low-level intrusion alerts using their *prerequisites* and *consequences*. The prerequisite of an alert specifies what must be true in order for the corresponding attack to be successful, and the consequence describes what is possibly true if the attack indeed succeeds. We conjecture that the alerts being correlated together have a higher possibility to be true alerts than the uncorrelated ones. If this is true, through this correlation, we can not only construct the high-level attack scenarios, but also differentiate between true alerts and false alerts.

In this thesis work, I implement an alert correlation tool based on this framework. It consists of the following components: a knowledge base, an alert preprocessor, an alert correlation engine and a graph output component. To further facilitate analysis of large amounts of intrusion alerts, I develop three utilities, namely adjustable graph reduction, focused analysis, and graph decomposition. I also perform a sequence of experiments to evaluate the aforementioned techniques using DARPA 2000 evaluation datasets [19] and DEFCON 8 CTF dataset [2]. The experimental results show that the proposed techniques are effective. First, we successfully construct attack scenarios behind the low-level alerts; second, the false alert rates are significantly reduced after the attention is focused on alerts that are correlated with others; third, the three utilities greatly reduce the complexity of the correlated alerts, while at the same time maintaining the structure of the correlated alerts.

A TOOLKIT FOR INTRUSION ALERTS CORRELATION  
BASED ON PREREQUISITES AND CONSEQUENCES OF  
ATTACKS

BY  
YUN CUI

A THESIS SUBMITTED TO THE GRADUATE FACULTY OF  
NORTH CAROLINA STATE UNIVERSITY  
IN PARTIAL FULFILLMENT OF THE  
REQUIREMENTS FOR THE DEGREE OF  
MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

RALEIGH  
DECEMBER 2002

APPROVED BY:

---

DR. PENG NING, CHAIR OF ADVISORY COMMITTEE

---

DR. DOUGLAS S. REEVES

---

DR. GREGORY T. BYRD

# Dedication

To my parents Liangcheng Cui, Meiqi Ma and my husband Deming Mao.

# Acknowledgments

I would like to express my sincere appreciation to my advisor, Dr. Peng Ning, for his patient guidance and constant support. His careful and critical comments significantly improves the content and presentations of this thesis. His knowledge and his dedication to research will be of life-long benefits for me.

I am grateful to my committee members, Dr. Douglas Reeves and Dr. Gregory Byrd, for their valuable comments, suggestions, and encouragements.

Thanks also to the colleagues in the Intrusion Detection Research Group: Pai Peng, Dingbang Xu, Kun Sun, Yan Zhai and Donggang Liu for all the help and support.

Finally, I am deeply grateful to my husband, Deming Mao, for his encouragement and support; I am indebted to my parents, Liangcheng Cui and Meiqi Ma who have been giving me constant support all my life.

This work is partially supported by the U.S. Army Research Office under grant DAAD19-02-1-0219, and by the National Science Foundation under grant CCR-0207297.

# Table of Contents

<b>List of Tables</b>	<b>vi</b>
<b>List of Figures</b>	<b>vii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Computer Security and Intrusion Detection System . . . . .	1
1.2 Summary of Contribution . . . . .	3
1.3 Organization of the Thesis . . . . .	3
<b>2 Related Work</b>	<b>4</b>
<b>3 A Framework for Alert Correlation</b>	<b>7</b>
3.1 Prerequisite and Consequence of Attacks . . . . .	7
3.2 Hyper-alert Type and Hyper-alert . . . . .	8
3.2.1 Temporal Constraints for Hyper-alerts . . . . .	12
3.3 Hyper-alert Correlation Graph . . . . .	12
3.4 Utilities . . . . .	15
3.4.1 Adjustable Graph Reduction . . . . .	15
3.4.2 Focused Analysis . . . . .	16
3.4.3 Graph Decomposition . . . . .	17
<b>4 Implementation</b>	<b>19</b>
4.1 Knowledge Base . . . . .	19
4.1.1 Database Structure . . . . .	19
4.1.2 Knowledge Base XML File . . . . .	21
4.2 Alert Preprocessor . . . . .	26
4.3 Correlation Engine . . . . .	27
4.4 Separating Correlation Graphs . . . . .	27
4.5 Transitive Edge Exclusion . . . . .	30
4.6 Adjustable Graph Reduction . . . . .	32
4.7 Focused Analysis . . . . .	33
4.8 Graph Decomposition . . . . .	34
4.9 Important Tables . . . . .	37
4.10 Property File . . . . .	38

<b>5</b>	<b>Experimental Results</b>	<b>39</b>
5.1	Experiment Setup . . . . .	39
5.2	Experiment on DARPA 2000 dataset . . . . .	40
5.2.1	Effectiveness of Alert Correlation . . . . .	40
5.2.2	Ability to Differentiate Alerts . . . . .	44
5.3	Experiment on Defcon 8 dataset . . . . .	45
5.3.1	Initial Attempt . . . . .	46
5.3.2	Adjustable Graph Reduction . . . . .	47
5.3.3	Focused Analysis . . . . .	50
5.3.4	Graph Decomposition . . . . .	50
<b>6</b>	<b>Conclusions and Future Work</b>	<b>55</b>
6.1	Conclusions . . . . .	55
6.2	Future Work . . . . .	56
6.2.1	Real-Time Correlator . . . . .	56
6.2.2	Input Alerts . . . . .	56
6.2.3	Identifying Missing Detections & Predicating Alerts . . . . .	56
6.2.4	Correlating Alerts from Different IDSs . . . . .	57
	<b>List of References</b>	<b>58</b>
<b>A</b>	<b>Knowledge Base Used In The Experiment</b>	<b>61</b>
A.1	XML Schema for Knowledge Base . . . . .	61
A.2	Knowledge Base Used for DARPA Dataset . . . . .	68
A.3	Knowledge Base Used for DEFCON 8 Dataset . . . . .	89
<b>B</b>	<b>JAVA Classes</b>	<b>96</b>
<b>C</b>	<b>User Installation &amp; Operation Manual</b>	<b>98</b>
C.1	Introduction . . . . .	98
C.2	Installation . . . . .	98
C.2.1	System Environment . . . . .	98
C.2.2	Checklist . . . . .	98
C.2.3	Download . . . . .	99
C.3	How to Use This Tool . . . . .	99
C.3.1	Knowledge Base . . . . .	99
C.3.2	Property File . . . . .	101
C.3.3	Execution . . . . .	103
C.4	Sample Execution Procedure . . . . .	105
C.5	Trouble Shooting . . . . .	105
	<b>List of References</b>	<b>106</b>

# List of Tables

5.1	Completeness and soundness of alert correlation. . . . .	44
5.2	Ability to differentiate true and false alerts . . . . .	45
5.3	General statistics of the initial analysis . . . . .	46
5.4	Statistics of top 10 uncorrelated hyper-alert types. . . . .	46
5.5	number of clusters and graphs after decomposing the largest hyper-alert correlation graph. . . . .	51
5.6	Detailed information of each cluster after decomposing the largest hyper-alert correlation graph. . . . .	51

# List of Figures

3.1	Hyper-alerts correlation graphs . . . . .	13
4.1	The architecture of the correlator. . . . .	20
4.2	Example tables for predicates and hyper-alert types in the knowledge base . . . . .	21
4.3	A sample “Predicates” section in a knowledge base XML file. . . . .	22
4.4	A sample “Implications” section in a knowledge base XML file. . . . .	23
4.5	A sample “HyperAlertTypes” section in a knowledge base XML file. . . . .	25
4.6	The algorithm used to generate a single graph. . . . .	29
4.7	Transitive edges. . . . .	30
4.8	The algorithm used to generate all the possible transitive edges. . . . .	31
4.9	The algorithm used to do the adjustable graph reduction. . . . .	33
4.10	The algorithm used to do the graph decomposition. . . . .	36
5.1	The (only) hyper-alert correlation graph discovered in the inside network traffic of LLDOS 1.0. . . . .	41
5.2	The hyper-alert correlation graphs discovered in the dmz network traffic of LLDOS1.0	42
5.3	The hyper-alert correlation graph discovered in the inside network traffic of LLDOS 2.0.2. . . . .	43
5.4	The hyper-alert correlation graph discovered in the dmz network traffic of LLDOS 2.0.2. . . . .	43
5.5	A small hyper-alert correlation discovered in initial analysis . . . . .	47
5.6	The fully reduced graph for the largest aggregated hyper-alert correlation graph. . .	48
5.7	Sizes of the reduced graphs w.r.t. the interval threshold for the largest hyper-alert correlation graph . . . . .	49
5.8	A fully reduced hyper-alert correlation graph resulting from graph decomposition with $C_{c1}$ . (Cluster ID = 1; DestIP = 010.020.001.010.) . . . . .	52
5.9	A fully reduced hyper-alert correlation graph resulting from graph decomposition with $C_{c2}$ . (Cluster ID = 10; SrcIP = 010.020.011.251; DestIP = 010.020.001.010.) . . . . .	53

# Chapter 1

## Introduction

### 1.1 Computer Security and Intrusion Detection System

With the Internet having grown enormously and globally, people care more and more about the security problems. Carnegie Mellon University's Computer Emergency Response Team's (CERT) [1] Coordination Center reported 3,734 attacks in 1998 and 52,658 attacks in 2001. Not only does the number of attacks increase quickly, the attacks are also more and more sophisticated. According to CERT's "Internet Security Overview", the sophistication of attacks is increasing and there are usually several stages involved in one attack. So, how to detect and respond to the attacks is becoming very important.

Intrusion detection has been used to protect information systems along with prevention-based mechanisms such as authentication and access control. Intrusion detection systems can be roughly classified as *anomaly detection* and *misuse detection*. Anomaly detection is based on the normal behavior of a subject (e.g., a user or a system). Any action that significantly deviates from the normal behavior is considered intrusive. Misuse detection is based on the characteristics of known attacks or system vulnerabilities, which are also called *signatures*. Any action that matches the signature is considered intrusive.

Both anomaly detection and misuse detection have their limitations. For anomaly detection, it is difficult to define what are normal behaviors and what are malicious behaviors. As to misuse detection, it detects attacks based on signatures of known attacks, and matches the traffic pattern with these signatures. If a match is found, then it is reported as an attack, otherwise it is not. So misuse detection cannot detect novel attacks.

Intrusion detection systems can also be classified as *network-based* and *host-based* according to the information source of the detection. Network-based IDS monitors the network traffic and looks for network-based attacks, while host-based IDS is installed on host and monitors the host audit

trail. Network-based IDS and host-based IDS have their own strength and limitations. Network-based IDS excels at detecting network-level abnormalities or abuses, but it cannot understand what is going on within the host. If the traffic is encrypted, network-based IDS can do nothing about it. The encrypted traffic can only be understood by host-based IDS after the traffic is decrypted on the host. But host-based IDS also has its weakness. It is hard to manage, easy to be attacked and disabled as part of the attack, does not suit for detecting network scans, can be disabled by denial-of-service attacks and inflicts a performance cost on the monitored systems [16].

Traditional intrusion detection systems usually focus on low-level attacks or anomalies, and raise alerts independently, though there may be logical connections between them. In situations where there are intensive intrusions, not only will actual alerts be mixed with false alerts, but the amount of alerts will also become unmanageable. As a result, it is difficult for human users or intrusion response systems to understand the intensive alerts and take appropriate actions.

Therefore, it is necessary to have techniques and tools to help improve the accuracy and quality of alerts. A good technique should aid the intrusion detection analyst or system administrator by pulling out “needles in the haystack” and bringing them to attention without the noise that generally follows.

Motivated by the above reasons, we have developed a technique [22, 23] to correlate the alerts generated by IDS. This technique is based on the *prerequisites* and *consequences* of attacks. The *prerequisite* of an attack specifies what must be true in order for the attack to be successful, and the *consequence* of the attack describes what is possibly true if the attack indeed succeeds. By reasoning about the *consequences* of earlier attacks and the *prerequisites* of later attacks, we can correlate the alerts together and construct attack scenarios. Moreover, we conjecture that the correlated alerts are more possible to be true alerts than uncorrelated ones. So, by correlation, we can potentially differentiate the false alerts from the real ones.

In this thesis work, I implement a set of tools based on the above technique. The basic tool for alert correlation is composed of a knowledge base, an alert preprocessor, an alert correlation engine and a graph output component. On the basis of the basic alert correlator, I also implement three utilities, namely adjustable graph reduction, focused analysis and graph decomposition, to facilitate analysis of intensive alerts.

The experiments in Chapter 5 show this technique is very effective. First, it uncovers the high-level attack scenario from a potentially large set of alerts. The DARPA dataset [19] has a detailed description about the attacks included in the dataset. The attack scenario generated by our tool matches this description very well. Second, the false alert rate is significantly dropped after we discard uncorrelated alerts. In one experiment, we reduce the false alert rate from 95.23% to 6.82% through correlation. This is significant as it filters out most of the false alerts so that the intrusion

detection analyst or system administrator can concentrate on the real attacks. To evaluate the effectiveness of our technique, we define two measurements: soundness and completeness. Soundness evaluates how correctly the alerts are correlated and completeness assesses how well we can correlate related alerts together. The values of soundness are all above 90%; however, the lowest value of completeness is only 62.5%. But further analysis reveals that all the alerts missed are those triggered by *telnet*. If this factor is considered, then the alerts correlated are highly accurate. Third, the three utilities are very useful in analyzing intensive dataset. It is demonstrated through the analysis of the DEFCON 8 CTF dataset [2] that the three utilities can greatly reduce the complexity of the correlated alerts, while at the same time keeping the structure of the attacks.

## 1.2 Summary of Contribution

The contribution of this thesis is twofold. The first is the development of an intrusion alert correlation tool based on the framework we proposed which uses the causal relationships among the alerts to correlate them. Second, some experiments are performed to evaluate the effectiveness of this tool. The experiments on DARPA 2000 evaluation datasets show that this method can not only uncover the attack scenario, but also filter out most of the false alerts. The experiments with DEFCON 8 CTF dataset further show the three utilities (adjustable graph reduction, focused analysis and graph decomposition) are very helpful in analyzing intensive alerts.

## 1.3 Organization of the Thesis

The remainder of this thesis is organized as follows. The next chapter discusses the related work on intrusion alert correlation. Chapter 3 presents the framework on which this tool is based. Chapter 4 describes the implementation details of the tool. Chapter 5 presents the experimental results of this tool. Chapter 6 concludes this thesis and points out some future work. Appendix A gives the knowledge base we use in this tool, Appendix B gives the summary of Java classes of the tool, and Appendix C is the user manual.

## Chapter 2

### Related Work

Intrusion detection has been studied for about twenty years. An excellent overview of intrusion detection techniques and related issues can be found in a recent book by Bace [7].

Several alert correlation methods have been proposed. These methods fall into three classes. The first class correlates alerts based on the similarities between alert attributes. Though they are effective for correlating some alerts (e.g., alerts with same source and destination IP addresses), they cannot fully discover the causal relationships between related alerts. The second class bases alert correlation on attack scenarios specified by human users or learned through training datasets. This is similar to misuse detection. Obviously, these methods are restricted to known attack scenarios. The third class is based on the preconditions and consequences of individual attacks.

Several alert correlation techniques have been proposed to facilitate analysis of intrusions. In [33], a probabilistic method was used to correlate alerts using similarity between their features. This belongs to the first category. As we discussed above, this method is not suitable for fully discovering causal relationships between alerts. In [30], a similar approach was applied to detect stealthy portscans along with several heuristics. Though some such heuristics (e.g., feature separation heuristics [30]) may be extended to general alert correlation problem, the approach cannot fully recover the causal relationships between alerts, either. The technique in [18] uses data mining to group all the similar alarms based on their root causes. Though they can reduce the false positive rate, they cannot provide the attack scenario.

Techniques for aggregating and correlating alerts have been proposed by others [11]. In particular, the correlation method in [11] uses a *consequence mechanism* to specify what types of alerts may follow a given alert type. This belongs to the second category. However, the consequence mechanism only uses alert types, the probes that generate the alerts, the severity level, and the time interval between the two alerts involved in a consequence definition, which do not provide sufficient information to correlate all possibly related alerts. Moreover, it is not easy to predict how an attacker may

arrange a sequence of attacks. In other words, developing a sufficient set of consequence definitions for alert correlation is not a solved problem.

Another approach, which also belongs to the second category, has been proposed to “learn” alert correlation models by applying machine learning techniques to training data sets embedded with known intrusion scenarios [10]. This approach can automatically build models for alert correlation; however, it requires training in every deployment, and the resulting models may overfit the training data, thereby missing attack scenarios not seen in the training data sets. The alert correlation techniques that we present in this paper address this same problem from a novel angle, overcoming the limitations of the above approaches.

JIGSAW [32] and our method fall into the third category. Both methods try to uncover attack scenarios based on specifications of individual attacks. However, our method also differs from JIGSAW. First, our method allows partial satisfaction of prerequisites (i.e., required capabilities in JIGSAW [32]), recognizing the possibility of undetected attacks and that of attackers gaining information through non-intrusive ways (e.g., talking to a friend working in the victim organization), while JIGSAW requires all required capabilities be satisfied. Second, our method allows aggregation of alerts, and thus can reduce the complexity involved in alert analysis, while JIGSAW currently does not have any similar mechanisms. Third, we develop a set of utilities for interactive analysis of correlated alerts, which is not provided by JIGSAW.

An alert correlation approach similar to ours was proposed recently in the MIRADOR project [8]. The MIRADOR approach also correlates alerts using partial match of prerequisites (pre-conditions) and consequences (post-conditions) of attacks. However, the MIRADOR approach uses a different formalism than ours. In particular, the MIRADOR approach treats alert aggregation as an individual stage before alert correlation, while our method allows alert aggregation during and after correlation. As we will see in the later sections, this difference leads to the three utilities for interactive alert analysis.

M-Correlator [26] first translates an alert to its internal incident report format. It then processes them using the vulnerability requirements of the incident type together with the knowledge of the protected network’s topology. It computes two parameters: one is the relevance score between the alert and the incident in its fact base; the other is the priority of the asset that the alert targets. Based on these two parameters, M-Correlator keeps the alerts that will cause the greatest risk to the monitored network.

M2D2 [21] introduces a formal information model for security information representation and correlation. The model includes four types of information: information system characteristics, vulnerabilities, security tools and events and alerts.

GrIDS [31] uses activity graphs to represent the causal structure of network activities and detect

propagation of large-scale attacks. Our method also uses graphs to represent correlated alerts. However, unlike GrIDS, in which nodes represent hosts or departments and edges represent network traffic between them, our method uses nodes to represent alerts, and edges the relationships between the alerts.

Several languages have been proposed to represent attacks, including STAT [34, 14], Colored-Petri Automata (CPA), LAMBDA [9], and MuSig [20] and its successor [25]. In particular, LAMBDA uses a logic-based method to specify the precondition and postcondition of attack scenarios, which is similar to our method. (See Chapter 3.) However, all these languages specify entire attack scenarios, which are limited to known scenarios. In contrast, our method (as well as JIGSAW) describes prerequisites and consequences of individual attacks, and correlate detected attacks (i.e., alerts) based on the relationship between these prerequisites and consequences. Thus, our method can potentially correlate alerts from unknown attack scenarios.

Alert correlation has been studied in the context of network management (e.g., [13], [27], and [12]). In theory, alert correlation methods for network management are applicable to intrusion alert correlation. However, intrusion alert correlation faces more challenges than its counterpart in network management: While alert correlation for network management deals with alerts about natural faults, which usually exhibit regular patterns, intrusion alert correlation has to cope with less predictable, malicious intruders.

## Chapter 3

# A Framework for Alert Correlation

This chapter describes the foundation of the toolkit, which has been proposed in [22] and [23].

In a series of attacks where the attackers launch earlier attacks to prepare for later ones, there are usually strong connections between the consequences of the earlier attacks and the prerequisites of the later ones. If an earlier attack is to prepare for a later attack, the consequence of the earlier attack should at least partly satisfy the prerequisite of the later attack.

Our method is to identify the prerequisites (e.g., existence of vulnerable services) and the consequences (e.g., discovery of vulnerable services) of each type of attack. These are then used to correlate alerts, which are attacks detected by IDSs, by matching the consequences of (the attacks corresponding to) some previous alerts and the prerequisites of (the attacks corresponding to) some later ones. For example, if we find a *Sadmin Ping* followed by a buffer overflow attack against the corresponding *Sadmin* service, we can correlate them to be parts of the same series of attacks. In other words, we model the knowledge (or state) of attackers in terms of individual attacks, and correlate alerts if they indicate the progress of attacks.

Note that an attacker does *not* have to perform early attacks to prepare for a later attack, even though the later attack has certain prerequisites. For example, an attacker may launch an individual buffer overflow attack against a service blindly, without knowing if the service exists. In other words, the prerequisite of an attack should not be mistaken for the necessary existence of an earlier attack. However, if the attacker does launch attacks with earlier ones preparing for later ones, our method can correlate them, provided that the attacks are detected by IDSs.

### 3.1 Prerequisite and Consequence of Attacks

Predicates are the basic constructs to represent prerequisites and consequences of attacks. For example, a scanning attack may discover UDP services vulnerable to a certain buffer overflow attack.

We can use the predicate  $UDPVulnerableToBOF (VictimIP, VictimPort)$  to represent the attacker's discovery. Similarly, if an attack requires a UDP service vulnerable to the buffer overflow attack, we can use the same predicate to represent the prerequisite.

Some attacks may require several conditions be satisfied at the same time in order to be successful. To represent such complex conditions, a logical combination of predicates is allowed in order to describe the prerequisite of an attack. For example, a network launched buffer overflow attack may require the target host have a vulnerable UDP service accessible to the attacker through the firewall. This prerequisite can be represented by  $UDPVulnerableToBOF (VictimIP, VictimPort) \wedge UDPAccessibleViaFirewall (VictimIP, VictimPort)$ . To simplify the discussion, we restrict the logical operators to  $\wedge$  (conjunction) and  $\vee$  (disjunction).

We use a set of predicates to represent the consequence of an attack. For example, an attack may result in compromise of the root privilege as well as modification of the *.rhost* file. Thus, we may use the following to represent the corresponding consequence:  $\{GainRootAccess (VictimIP), rhostModified (VictimIP)\}$ . Note that the set of predicates used to represent the consequence is essentially the conjunction of these predicates and can be represented by a single logical formula. However, representing the consequence as a set rather than a long formula is more convenient and will be used here.

The consequence of an attack is indeed the *possible* result of the attack. In other words, the attack may or may not generate the stated consequence, depending on whether the attack is successful or not. For example, after a buffer overflow attack against a service, an attacker may or may not gain the root access, depending on if the service is vulnerable to the attack.

We use *possible* consequences instead of *actual* consequences due to the following reasons. First, an IDS may not have enough information to decide if an attack is effective or not. For example, a network based IDS can detect certain buffer overflow attacks by matching the patterns of the attacks; however, it cannot decide whether the attempts succeed or not without more information from the related hosts. In contrast, the possible consequence of a type of attack can be analyzed and made available for IDS. Second, even if an attack fails to prepare for the follow-up attacks, the follow-up attacks may still occur simply because, for example, the attacker uses a script to launch a series of attacks. Using possible consequences of attacks will lead to better opportunity to correlate such attacks.

### 3.2 Hyper-alert Type and Hyper-alert

Using predicates as the basic construct, we introduce the notion of a *hyper-alert type* to represent the prerequisite and the consequence of each type of alert.

**Definition 1** A *hyper-alert type*  $T$  is a triple  $(fact, prerequisite, consequence)$ , where (1) *fact* is a set of attribute names, each with an associated domain of values, (2) *prerequisite* is a logical combination of predicates whose free variables are all in *fact*, and (3) *consequence* is a set of predicates such that all the free variables in *consequence* are in *fact*.

Each hyper-alert type encodes the knowledge about a type of attack. The component *fact* of a hyper-alert type tells what kind of information is reported along with the alert (i.e., detected attack), *prerequisite* specifies what must be true in order for the attack to be successful, and *consequence* describes what is true if the attack indeed succeeds. For the sake of brevity, we omit the domains associated with the attribute names when they are clear from the context.

**Example 1** Consider the buffer overflow attack against the *sadmind* remote administration tool. We may have a hyper-alert type  $SadmindBufferOverflow = (\{ VictimIP, VictimPort \}, ExistHost(VictimIP) \wedge VulnerableSadmind(VictimIP), \{ GainRootAccess(VictimIP) \})$  for such attacks. Intuitively, this hyper-alert type says that such an attack is against the host at IP address *VictimIP*. (We expect the actual values of *VictimIP* are reported by an IDS.) For the attack to be successful, there must exist a host at IP address *VictimIP*, and the corresponding *sadmind* service must be vulnerable to buffer overflow attacks. The attacker may gain root privilege as a result of the attack.

Given a hyper-alert type, a *hyper-alert instance* can be generated if the corresponding attack is detected and reported by an IDS. For example, we can generate a hyper-alert instance of type *SadmindBufferOverflow* from a corresponding alert. The notion of hyper-alert instance is formally defined as follows.

**Definition 2** Given a hyper-alert type  $T = (fact, prerequisite, consequence)$ , a *hyper-alert (instance)*  $h$  of type  $T$  is a finite set of tuples on *fact*, where each tuple is associated with an interval-based timestamp  $[begin\_time, end\_time]$ . The hyper-alert  $h$  implies that *prerequisite* must evaluate to True and all the predicates in *consequence* might evaluate to True for each of the tuples. (Notation-wise, for each tuple  $t$  in  $h$ , we use  $t.begin\_time$  and  $t.end\_time$  to refer to the timestamp associated with  $t$ .)

The *fact* component of a hyper-alert type is essentially a relation schema (as in relational databases), and a hyper-alert is a relation instance of this schema. One may point out that an alternative way is to represent a hyper-alert as a record, which is equivalent to a single tuple on *fact*. However, such an alternative cannot accommodate certain alerts possibly reported by an IDS. For example, an IDS may report an IPSweep attack along with multiple swept IP addresses, which cannot be represented as a single record. In addition, our current formalism allows aggregation of alerts of the same type, and is flexible in reasoning about alerts. Therefore, we believe the current notion of a hyper-alert is an appropriate choice.

A hyper-alert instantiates its *prerequisite* and *consequence* by replacing the free variables in

*prerequisite* and *consequence* with its specific values. Since all free variables in *prerequisite* and *consequence* must appear in *fact* in a hyper-alert type, the instantiated prerequisite and consequence will have no free variables. Note that *prerequisite* and *consequence* can be instantiated multiple times if *fact* consists of multiple tuples.

In the following, we treat timestamps implicitly and omit them if they are not necessary for our discussion.

**Example 2** Consider the hyper-alert type *SadminBufferOverflow* in example 1. A hyper-alert  $h_{SadminBOF}$  of this type could be:  $\{(VictimIP = 152.1.19.5, VictimPort = 1235), (VictimIP = 152.1.19.7, VictimPort = 1235)\}$ . This implies that if the attack is successful, the following two logical formulas must be True as the prerequisites of the attack:  $ExistHost(152.1.19.5) \wedge VulnerableSadmin(152.1.19.5)$ ,  $ExistHost(152.1.19.7) \wedge VulnerableSadmin(152.1.19.7)$ . Moreover, as possible consequences of the attack, the following might be True:  $GainRootAccess(152.1.19.5)$ ,  $GainRootAccess(152.1.19.7)$ . This hyper-alert says that there are buffer overflow attacks against *sadmin* at IP addresses 152.1.19.5 and 152.1.19.7, and the attacker may gain root access as a result of the attacks.

A hyper-alert may correspond to one or several related alerts. If an IDS reports one alert for a certain attack and the alert has all the information needed to instantiate a hyper-alert, a hyper-alert can be generated from the alert. However, some IDSs may report a series of alerts for a single attack. For example, EMERALD may report several alerts (within the same thread) related to an attack that spreads over a period of time. In this case, a hyper-alert may correspond to the aggregation of all the related alerts. Moreover, several alerts may be reported for the same type of attack in a short period of time. Our definition of hyper-alert allows them to be treated as one hyper-alert, and thus provides flexibility in the reasoning about alerts. Certain constraints are necessary to make sure the hyper-alerts are reasonable. However, since our hyper-alert correlation method does not depend on them directly, we will discuss them after introducing our method.

Ideally, we may correlate a set of hyper-alerts with a later hyper-alert together if the consequences of the former ones imply the prerequisite of the latter one. However, such an approach may not work in reality due to several reasons. First, the attacker may not always prepare for certain attacks by launching some other attacks. For example, the attacker may learn a vulnerable *sadmin* service by talking to people who work in the organization where the system is running. Second, the current IDSs may miss some attacks, and thus affect the alert correlation if the above approach is used. Third, due to the combinatorial nature of the aforementioned approach, it is computationally expensive to examine sets of alerts to find out whether their consequences satisfy (or more precisely, imply) the prerequisite of an alert.

Having considered these issues, we adopt an alternative approach. Instead of examining if several

hyper-alerts satisfy the prerequisite of a later one, we check if an earlier hyper-alert *contributes* to the prerequisite of a later one. Specifically, we decompose the prerequisite of a hyper-alert into pieces of predicates and test whether the consequence of an earlier hyper-alert makes some pieces of the prerequisite True (i.e., make the prerequisite easier to satisfy). If the result is yes, then we correlate the hyper-alerts together. This idea is specified formally through the following Definitions.

**Definition 3** Consider a hyper-alert type  $T = (fact, prerequisite, consequence)$ . The *prerequisite set* (or *consequence set*, resp.) of  $T$ , denoted  $P(T)$  (or  $C(T)$ , resp.), is the set of all predicates that appear in *prerequisite* (or *consequence*, resp.). Given a hyper-alert instance  $h$  of type  $T$ , the *prerequisite set* (or *consequence set*, resp.) of  $h$ , denoted  $P(h)$  (or  $C(h)$ , resp.), is the set of predicates in  $P(T)$  (or  $C(T)$ , resp.) whose arguments are replaced with the corresponding attribute values of each tuple in  $h$ . Each element in  $P(h)$  (or  $C(h)$ , resp.) is associated with the timestamp of the corresponding tuple in  $h$ . (Notation-wise, for each  $p \in P(h)$  (or  $C(h)$ , resp.), we use  $p.begin\_time$  and  $p.end\_time$  to refer to the timestamp associated with  $p$ .)

**Example 3** Consider the *Sadmin Ping* attack through which an attacker discovers possibly vulnerable *sadmin* services. The corresponding alerts can be represented by a hyper-alert type  $SadminPing = (\{VictimIP, VictimPort\}, ExistHost (VictimIP), \{VulnerableSadmin (VictimIP)\})$ .

Suppose a hyper-alert instance  $h_{SadminPing}$  of type  $SadminPing$  has the following tuples:  $\{(VictimIP = 152.1.19.5, VictimPort = 1235), (VictimIP = 152.1.19.7, VictimPort = 1235), (VictimIP = 152.1.19.9, VictimPort = 1235)\}$ . Then  $P(h_{SadminPing}) = \{ExistHost (152.1.19.5), ExistHost (152.1.19.7), ExistHost (152.1.19.9)\}$ , and  $C(h_{SadminPing}) = \{VulnerableSadmin (152.1.19.5), VulnerableSadmin (152.1.19.7), VulnerableSadmin (152.1.19.9)\}$ .

**Example 4** Consider  $h_{SadminBOF}$  in example 2. We have  $P(h_{SadminBOF}) = \{ExistHost (152.1.19.5), ExistHost (152.1.19.7), VulnerableSadmin (152.1.19.5), VulnerableSadmin (152.1.19.7)\}$  and  $C(h_{SadminBOF}) = \{GainRootAccess (152.1.19.5), GainRootAccess (152.1.19.7)\}$ .

**Definition 4** Hyper-alert  $h_1$  *prepares for* hyper-alert  $h_2$ , if there exist  $p \in P(h_2)$  and  $C \subseteq C(h_1)$  such that for all  $c \in C$ ,  $c.end\_time < p.begin\_time$  and the conjunction of all the predicates in  $C$  implies  $p$ .

The prepare-for relation is developed to capture the causal relationships between hyper-alerts. Intuitively,  $h_1$  prepares for  $h_2$  if some attacks represented by  $h_1$  make the attacks represented by  $h_2$  easier to succeed.

**Example 5** Let us continue examples 3 and 4. Assume that all tuples in  $h_{SadminPing}$  have timestamps earlier than every tuple in  $h_{SadminBOF}$ . By comparing  $C(h_{SadminPing})$  and  $P(h_{SadminBOF})$ , it is clear that  $VulnerableSadmin (152.1.19.5)$  (among others) in  $P(h_{SadminBOF})$  is also in  $C(h_{SadminPing})$ . Thus,  $h_{SadminPing}$  prepares for, and is correlated with  $h_{SadminBOF}$ .

Given a sequence  $S$  of hyper-alerts, a hyper-alert  $h$  in  $S$  is a *correlated hyper-alert*, if there exists another hyper-alert  $h'$  in  $S$  such that either  $h$  prepares for  $h'$  or  $h'$  prepares for  $h$ . If no such  $h'$  exists,  $h$  is called an *isolated hyper-alert*. Our goal is to discover all pairs of hyper-alerts  $h_1$  and  $h_2$  in  $S$  such that  $h_1$  prepares for  $h_2$ .

### 3.2.1 Temporal Constraints for Hyper-alerts

As discussed earlier, we allow multiple alerts to be aggregated into a hyper-alert, which gives some flexibility in reasoning about alerts. However, the definition of hyper-alert is overly flexible in some situations; it allows alerts that occur at arbitrary points in time to be treated as a single hyper-alert. Although some attackers usually spread their intrusive activities over time, aggregating alerts at arbitrary time points is still overly broad, and may affect the effectiveness of alert correlation.

In the following, we introduce two temporal constraints for hyper-alerts. The purpose of these temporal constraints is to restrict the alert aggregation to meaningful ones. We are particularly interested in hyper-alerts that satisfy at least one of the constraints. However, most of our discussion in this paper applies to general hyper-alerts. Thus, we will not specifically indicate the constraints if it is not necessary.

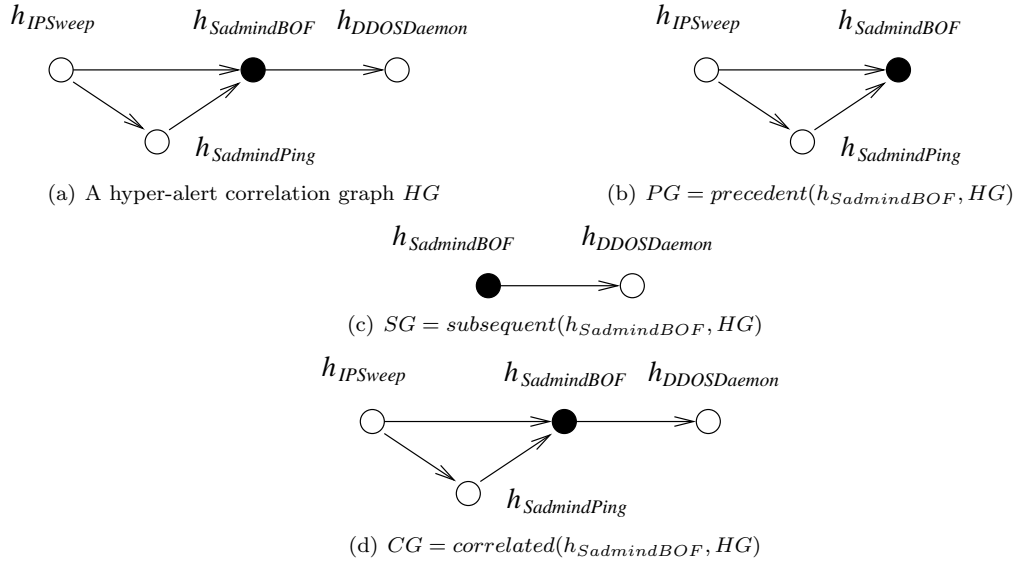
**Definition 5** Given a time duration  $D$ , a hyper-alert  $h$  satisfies *duration constraint of  $D$*  if  $Max\{t.end\_time|\forall t \in h\} - Min\{t.begin\_time|\forall t \in h\} < D$ .

**Definition 6** Given a time interval  $I$ , a hyper-alert  $h$  satisfies *interval constraint of  $I$*  if (1)  $h$  has only one tuple, or (2) for all  $t$  in  $h$ , there exist another  $t'$  in  $h$  such that there exist  $t.begin\_time < T < t.end\_time$ ,  $t'.begin\_time < T' < t'.end\_time$ , and  $|T - T'| < I$ .

The temporal constraints are introduced to prevent unreasonable aggregation of alerts. However, this does not imply that alerts have to be aggregated. Indeed, in our initial experiments, we treat each alert as an individual hyper-alert. In other words, aggregation of alerts is an option provided by our model, and temporal constraints are restrictions that make the aggregated hyper-alerts meaningful.

## 3.3 Hyper-alert Correlation Graph

The prepare-for relation between hyper-alerts provides a natural way to represent the causal relationship between correlated hyper-alerts. In the following, we introduce the notion of a *hyper-alert correlation graph* to represent attack scenarios on the basis of the prepare-for relation. As we will see, the hyper-alert correlation graph reflects the high-level strategies or logical steps behind a sequence of attacks.

**Figure 3.1:** Hyper-alerts correlation graphs

**Definition 7** A *hyper-alert correlation graph*  $HG = (N, E)$  is a connected DAG (directed acyclic graph), where the set  $N$  of nodes is a set of hyper-alerts, and for each pair of nodes  $n_1, n_2 \in N$ , there is an edge from  $n_1$  to  $n_2$  in  $E$  if and only if  $n_1$  prepares for  $n_2$ .

**Example 6** Suppose in a sequence of hyper-alerts we have the following ones:  $h_{IPSweep}$ ,  $h_{SadmindPing}$ ,  $h_{SadmindBOF}$ , and  $h_{DDOSDaemon}$ .  $h_{SadmindBOF}$  and  $h_{SadmindPing}$  have been explained in examples 2 and 3, respectively. Suppose  $h_{IPSweep}$  represents an IP Sweep attack, and  $h_{DDOSDaemon}$  represents the activity of a DDOS daemon program. Assume that  $h_{IPSweep}$  prepares for  $h_{SadmindPing}$  and  $h_{SadmindBOF}$ , respectively,  $h_{SadmindPing}$  prepares for  $h_{SadmindBOF}$ , and  $h_{SadmindBOF}$  prepares for  $h_{DDOSDaemon}$ . These are intuitively shown in a hyper-alert correlation graph in Figure 3.1(a).

The hyper-alert correlation graph provides an intuitive representation of correlated hyper-alerts. With this notion, the goal of alert correlation can be rephrased as the discovery of hyper-alert correlation graphs that have maximal number of nodes from a sequence of hyper-alerts.

In addition to getting all the correlated hyper-alerts, it is often desirable to discover those directly or indirectly correlated to one particular hyper-alert. For example, if an IDS detects a DDOS daemon running on a host, it would be helpful to inform the administrator how this happened, that is, report all the alerts that directly or indirectly prepare for the DDOS daemon. Therefore, we define the following operations on hyper-alert correlation graphs.

**Definition 8** Given a hyper-alert correlation graph  $HG = (N, E)$  and a hyper-alert  $n$  in  $N$ , *precedent* ( $n, HG$ ) is an operation that returns the maximal sub-graph  $PG = (N', E')$  of  $HG$  that

satisfies the following conditions: (1)  $n \in N'$ , (2) for each  $n' \in N'$  other than  $n$ , there is a directed path from  $n'$  to  $n$ , and (3) each edge  $e \in E'$  is in a path from a node  $n'$  in  $N'$  to  $n$ . The resulting graph  $PG$  is called the *precedent graph of  $n$  w.r.t.  $HG$* .

**Definition 9** Given a hyper-alert correlation graph  $HG = (N, E)$  and a hyper-alert  $n$  in  $N$ , *subsequent* ( $n, HG$ ) is an operation that returns the maximum sub-graph  $SG = (N', E')$  of  $HG$  that satisfies the following conditions: (1)  $n \in N'$ , (2) for each  $n' \in N'$  other than  $n$ , there is a directed path from  $n$  to  $n'$ , and (3) each edge  $e \in E'$  is in a path from  $n$  to a node  $n'$  in  $N'$ . The resulting graph  $SG$  is called the *subsequent graph of  $n$  w.r.t.  $HG$* .

**Definition 10** Given a hyper-alert correlation graph  $HG = (N, E)$  and a hyper-alert  $n$  in  $N$ , *correlated* ( $n, HG$ ) is an operation that returns the maximal sub-graph  $CG = (N', E')$  of  $HG$  that satisfies the following conditions: (1)  $n \in N'$ , (2) for each  $n' \in N'$  other than  $n$ , there is either a path from  $n$  to  $n'$ , or a path from  $n'$  to  $n$ , and (3) each edge  $e \in E'$  is either in a path from a node in  $N'$  to  $n$ , or in a path from  $n$  to a node in  $N'$ . The resulting graph  $CG$  is called the *correlated graph of  $n$  w.r.t.  $HG$* .

Intuitively, the precedent graph of  $n$  w.r.t.  $HG$  describes all the hyper-alerts in  $HG$  that prepare for  $n$  directly or indirectly, the subsequent graph of  $n$  w.r.t.  $HG$  describes all the hyper-alerts in  $HG$  for which  $n$  prepares directly or indirectly, and the correlated graph of  $n$  w.r.t.  $HG$  includes all the hyper-alerts in  $HG$  that are correlated to  $n$  directly or indirectly. It is easy to see that  $correlated(n, HG) = precedent(n, HG) \cup subsequent(n, HG)$ .

Assuming the black node  $h_{Sadmin}dBOF$  in figure 3.1(a) is the hyper-alert of concern, figures 3.1(b) to 3.1(d) display the precedent graph, subsequent graph, and correlated graph of  $h_{Sadmin}dBOF$  w.r.t. the hyper-alert correlation graph in figure 3.1(a), respectively. Note that figure 3.1(d) is the same as figure 3.1(a). This is because all the hyper-alerts in figure 3.1(a) are related to  $h_{Sadmin}dBOF$  via the prepare-for relation.

The hyper-alert correlation graph is not only an intuitive representation of attack scenarios constructed through alert correlation, but also reveals opportunities to improve intrusion detection. First, the hyper-alert correlation graph can potentially reveal the intrusion strategies behind the attacks, and lead to better understanding of the attacker's intention. Second, assuming some attackers exhibit patterns in their strategies, we can use the hyper-alert correlation graph to profile previous attacks and identify on-going attacks by matching to the profiles. A partial match to the profile may indicate attacks possibly missed by the IDSs, and lead to human investigation and improvement of the IDSs.

## 3.4 Utilities

Correlation is effective to correlate the alerts and uncover the attack strategy. However, only correlation itself is not effective enough in some cases especially in dealing with intensive datasets. So, three utilities are proposed to help analyze the huge intrusion datasets.

### 3.4.1 Adjustable Graph Reduction

This utility is aimed to reduce the complexity (i.e., the number of nodes and edges) of hyper-alert correlation graphs while keeping the structure of sequences of attacks. The graph reduction is adjustable in the sense that users are allowed to control the degree of reduction.

Aggregation is very useful to reduce the complexity of the attack scenario analysis. Especially when the dataset is very large and we need some graphic output. It can dramatically reduce the number of nodes and number of edges in the graph. In the experiment with the data from DEFCON Capture the Flag (CTF), aggregation technique shows its power.

A natural way to reduce the complexity of a hyper-alert correlation graph is to reduce the number of nodes and edges. However, to make the reduced graph useful, any reasonable reduction should maintain the structure of the corresponding attacks.

We propose to aggregate hyper-alerts of the same type to reduce the number of nodes in a hyper-alert correlation graph. Due to the flexible definition of hyper-alerts, the result of hyper-alert aggregation will remain valid hyper-alerts. For example, in Figure 5.1, hyper-alerts 67432, 67434, 67436, and 67440 are all instances of hyper-alert type *Sadmind\_Amslverify\_Overflow*. Thus, we may aggregate them into one hyper-alert. As another example, hyper-alerts 6755, 67558, 67559, and 67560 are all instances of *Rsh*, and can be aggregated into a single hyper-alert.

Edges are reduced along with the aggregation of hyper-alerts. In Figure 5.1, the edges between the *Rsh* hyper-alerts are subsumed into the aggregated hyper-alert, while the edges between the *Sadmind\_Ping* hyper-alert and the four *Sadmind\_Amslverify\_Overflow* hyper-alerts are merged into a single edge.

Reduction of a hyper-alert correlation graph may lose information contained in the original graph. Indeed, hyper-alerts that are of the same type but belong to different sequences of attacks may be aggregated and thus provide overly simplified results. Nevertheless, our goal is to lose as little information of the structure of attacks as possible.

Depending on the actual alerts, the reduction of a hyper-alert correlation graph may be less simplified, or over simplified. We would like to give a human user more control over the graph reduction process. In the following, we use a simple mechanism to control this process, based on the notion of an interval constraint.

**Definition 11** Given a time interval  $I$  (e.g., 10 seconds), a hyper-alert  $h$  satisfies *interval constraint of  $I$*  if (1)  $h$  has only one tuple, or (2) for all  $t$  in  $h$ , there exist another  $t'$  in  $h$  such that there exist  $t.begin\_time < T < t.end\_time$ ,  $t'.begin\_time < T' < t'.end\_time$ , and  $|T - T'| < I$ .

We allow hyper-alert aggregation only when the resulting hyper-alerts satisfy an interval constraint of a given threshold  $I$ . Intuitively, we allow hyper-alerts to be aggregated only when they are close to each other. The larger a threshold  $I$  is, the more a hyper-alert correlation graph can be reduced. By adjusting the interval threshold, a user can control the degree to which a hyper-alert correlation graph is reduced.

### 3.4.2 Focused Analysis

Focused analysis is implemented on the basis of focusing constraints. A *focusing constraint* is a logical combination of comparisons between attribute names and constants. (In our work, we restrict logical operations to AND ( $\wedge$ ), OR ( $\vee$ ), and NOT ( $\neg$ .) For example, we may have a focusing constraint  $SrcIP = 129.174.142.2 \vee DestIP = 129.174.142.2$ . We say a focusing constraint  $C_f$  is *enforceable w.r.t. a hyper-alert type  $T$*  if when we represent  $C_f$  in a disjunctive normal form, at least for one disjunct  $C_{fi}$ , all the attribute names in  $C_{fi}$  appear in  $T$ . For example, the above focusing constraint is enforceable w.r.t.  $T = (\{SrcIP, SrcPort\}, NULL, \emptyset)$ , but not w.r.t.  $T' = (\{VictimIP, VictimPort\}, NULL, \emptyset)$ . Intuitively, a focusing constraint is enforceable w.r.t.  $T$  if it can be evaluated using a hyper-alert instance of type  $T$ .

We may *evaluate* a focusing constraint  $C_f$  with a hyper-alert  $h$  if  $C_f$  is enforceable w.r.t. the type of  $h$ . A focusing constraint  $C_f$  evaluates to True for  $h$  if there exists a tuple  $t \in h$  such that  $C_f$  is True with the attribute names replaced with the values of the corresponding attributes of  $t$ ; otherwise,  $C_f$  evaluates to False. For example, consider the aforementioned focusing constraint  $C_f$ , which is  $SrcIP = 129.174.142.2 \vee DestIP = 129.174.142.2$ , and a hyper-alert  $h = \{(SrcIP = 129.174.142.2, SrcPort = 80)\}$ , we can easily have that  $C_f = \text{True}$  for  $h$ .

The idea of focused analysis is quite simple: we only analyze the hyper-alerts with which a focusing constraint evaluates to True. In other words, we would like to filter out irrelevant hyper-alerts, and concentrate on analyzing the remaining hyper-alerts. We are particularly interested in applying focusing constraints to *atomic hyper-alerts*, i.e., hyper-alerts with only one tuple. In our framework, atomic hyper-alerts correspond to the alerts reported by an IDS directly.

Focused analysis is particularly useful when we have certain knowledge of the alerts, the systems being protected, or the attacking computers. For example, if we are interested in the attacks against a critical server with IP address  $Server\_IP$ , we may perform a focused analysis using  $DestIPAddress = Server\_IP$ . However, focused analysis cannot take advantage of the intrinsic relationship among the hyper-alerts (e.g., hyper-alerts having the same IP address). In the following, we introduce the

third utility, graph decomposition, to fill in this gap.

### 3.4.3 Graph Decomposition

The purpose of graph decomposition is to use the inherent relationship between (the attributes of) hyper-alerts to decompose a hyper-alert correlation graph. Conceptually, we cluster the hyper-alerts in a large correlation graph based on the “common features” shared by hyper-alerts, and then decompose the original correlation graphs into subgraphs on the basis of the clusters. In other words, hyper-alerts should remain in the same graph only when they share certain common features.

We use a *clustering constraint* to specify the “common features” for clustering hyper-alerts. Given two sets of attribute names  $A_1$  and  $A_2$ , a *clustering constraint*  $C_c(A_1, A_2)$  is a logical combination of comparisons between constants and attribute names in  $A_1$  and  $A_2$ . (In our work, we restrict logical operations to AND ( $\wedge$ ), OR ( $\vee$ ), and NOT ( $\neg$ ).) A clustering constraint is a constraint for two hyper-alerts; the attribute sets  $A_1$  and  $A_2$  identify the attributes from the two hyper-alerts. For example, we may have two sets of attribute names  $A_1 = \{SrcIP, DestIP\}$  and  $A_2 = \{SrcIP, DestIP\}$ , and  $C_c(A_1, A_2) = (A_1.SrcIP = A_2.SrcIP) \wedge (A_1.DestIP = A_2.DestIP)$ . Intuitively, this is to say two hyper-alerts should remain in the same cluster if they have the same source and destination IP addresses.

A clustering constraint  $C_c(A_1, A_2)$  is *enforceable w.r.t. hyper-alert types  $T_1$  and  $T_2$*  if when we represent  $C_c(A_1, A_2)$  in a disjunctive normal form, at least for one disjunct  $C_{ci}$ , all the attribute names in  $A_1$  appear in  $T_1$  and all the attribute names in  $A_2$  appear in  $T_2$ . For example, the above clustering constraint is enforceable w.r.t.  $T_1$  and  $T_2$  if both of them have *SrcIP* and *DestIP* in the *fact* component. Intuitively, a focusing constraint is enforceable w.r.t.  $T$  if it can be evaluated using two hyper-alerts of types  $T_1$  and  $T_2$ , respectively.

If a clustering constraint  $C_c(A_1, A_2)$  is enforceable w.r.t.  $T_1$  and  $T_2$ , we can *evaluate* it with two hyper-alerts  $h_1$  and  $h_2$  that are of type  $T_1$  and  $T_2$ , respectively. A clustering constraint  $C_c(A_1, A_2)$  evaluates to True for  $h_1$  and  $h_2$  if there exists a tuple  $t_1 \in h_1$  and  $t_2 \in h_2$  such that  $C_c(A_1, A_2)$  is True with the attribute names in  $A_1$  and  $A_2$  replaced with the values of the corresponding attributes of  $t_1$  and  $t_2$ , respectively; otherwise,  $C_c(A_1, A_2)$  evaluates to False. For example, consider the clustering constraint  $C_c(A_1, A_2) : (A_1.SrcIP = A_2.SrcIP) \wedge (A_1.DestIP = A_2.DestIP)$ , and hyper-alerts  $h_1 = \{(SrcIP = 129.174.142.2, SrcPort = 1234, DestIP = 152.1.14.5, DestPort = 80)\}$ ,  $h_2 = \{(SrcIP = 129.174.142.2, SrcPort = 65333, DestIP = 152.1.14.5, DestPort = 23)\}$ , we can easily have that  $C_c(A_1, A_2) = \text{True}$  for  $h_1$  and  $h_2$ . For brevity, we write  $C_c(h_1, h_2) = \text{True}$  if  $C_c(A_1, A_2) = \text{True}$  for  $h_1$  and  $h_2$ .

Our clustering method is very simple with a user-specified clustering constraint  $C_c(A_1, A_2)$ . Two hyper-alerts  $h_1$  and  $h_2$  are in the same cluster if  $C_c(A_1, A_2)$  evaluates to True for  $h_1$  and  $h_2$  (or

$h_2$  and  $h_1$ ). Note that  $C_c(h_1, h_2)$  implies that  $h_1$  and  $h_2$  are in the same cluster, but  $h_1$  and  $h_2$  in the same cluster do not always imply  $C_c(h_1, h_2) = \text{True}$  or  $C_c(h_2, h_1) = \text{True}$ . This is because  $C_c(h_1, h_2) \wedge C_c(h_2, h_3)$  does not imply  $C_c(h_1, h_3)$ , nor  $C_c(h_3, h_1)$ .

## Chapter 4

# Implementation

In this chapter, we give an overview of the implementation of an off-line intrusion alerts correlator based on the frame work discussed before. Figure 4.1 shows the architecture of this tool. It consists of a knowledge base, an alert preprocessor, a correlation engine, utility processors, a graph generator and a visualization component. We adopt the GraphViz package [1] as the visualization component.

We assume that all the alerts generated by IDSs are stored in a relational database, so is the knowledge base. All these components except for the visualization component interact with a relational database management system (RDBMS), which provides persistent storage for the intermediate data as well as the correlated alerts. In other words, this tool interacts with the RDBMS to access the alerts generated by IDSs and store the results back into the database. Such a method not only takes advantage of the RDBMS's functionalities, but also allows easy integration with other RDBMS-based intrusion analysis tools.

Java is used to implement this tool and JDBC is used to communicate with the database.

### 4.1 Knowledge Base

The knowledge base contains the necessary information about hyper-alert types as well as relationships between predicates. To simplify the implementation, we assume that each hyper-alert type is uniquely identified by its name, and there is no negation in the prerequisite nor the consequence of any hyper-alert type.

#### 4.1.1 Database Structure

There are totally five tables in the knowledge database. Two tables (*Predicate* and *Implication*) contain the information about predicates, and the other three tables (*HATFact*, *HATPrereq* and *HATConseq*) have the information about hyper-alert types.

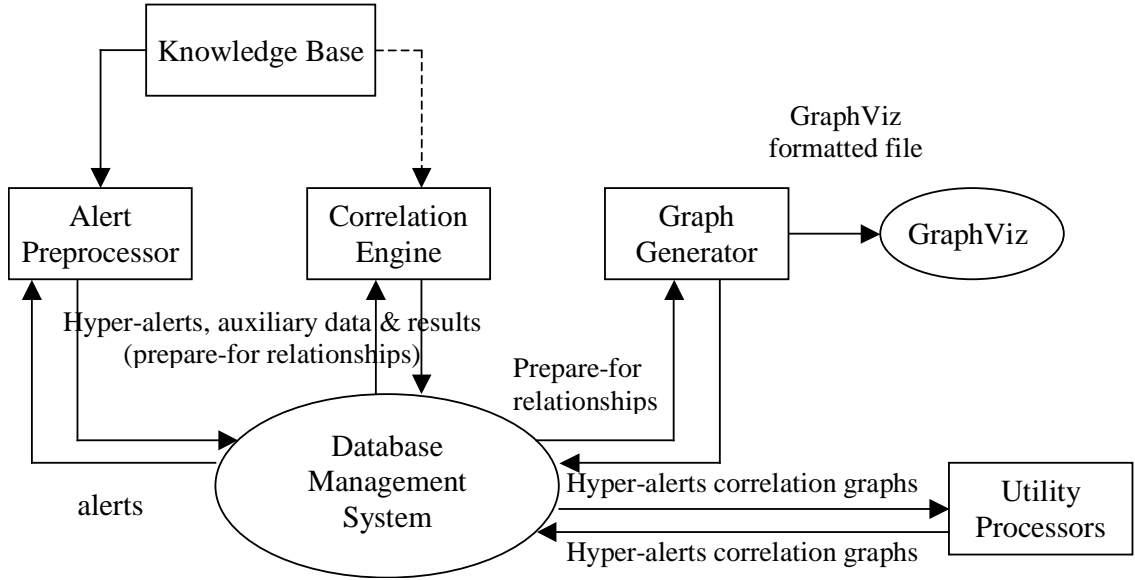


Figure 4.1: The architecture of the correlator.

Figure 4.2 shows the structures of the tables and some examples of the knowledge base.

Table *Predicate* (figure 4.2(a)) stores all the predicates that appear in the knowledge base. Their names, the number of arguments and arguments' data types are stored in the table. So, the *Predicate* table has three attributes: *Predicate*, *ArgNum*, and *ArgType*, to represent the predicate name, the position of an argument, and the data type of the argument, respectively. For example, the first tuple in figure 4.2(a) shows predicate *ExistHost* has one argument and this argument's data type is *varchar(15)*. The second and the third tuple in this figure represent predicate *ExistService*, which has two arguments. The first argument's data type is *varchar(15)* and the second argument's data type is *int*. All these data types and the data types we discuss in the following sections are all database data types.

Table *Implication* (figure 4.2(b)) keeps the implication relationships between predicates as well as the mapping between their arguments. It has four attributes: *Predicate*, *Implied*, *P\_Arg* and *L\_Arg*. *Predicate* attribute is to record the implying predicate name. *Implied* attribute is to record the implied predicate name. *P\_Arg* is the argument position in the implying predicate. *L\_Arg* is the argument position in the implied predicate. For example, the first tuple in figure 4.2(b) indicates that *ExistService* implies *ExistHost* and they both take their first argument.

Table *HATFact* (figure 4.2(c)) contains all the fact information of all hyper-alert types. It has

Predicate	ArgNum	ArgType
ExistHost	1	varchar (15)
ExistService	1	varchar (15)
ExistService	2	int
GainOSInfo	1	varchar (15)

(a) Table Predicate

Predicate	Implied	P_Arg	L_Arg
ExistService	ExistHost	1	1

(b) Table Implication

HyperAlertType	AttrName	AttrType
FTP_Syst	DestIPAddress	varchar (15)
FTP_Syst	DestPort	int

(c) Table HATFact

HyperAlertType	P_ID	Predicate	ArgPos	ArgName
FTP_Syst	1	ExistService	1	DestIPAddress
FTP_Syst	1	ExistService	2	DestPort

(d) Table HATPrereq

HyperAlertType	P_ID	Predicate	ArgPos	ArgName
FTP_Syst	1	GainOSInfo	1	DestIPAddress

(e) Table HATConseq

**Figure 4.2:** Example tables for predicates and hyper-alert types in the knowledge base

three attributes: *HyperAlertType*, *AttrName*, and *AttrType*. They represent the name of a hyper-alert type, the name of a fact, and the data type of a fact, respectively. For example, the two tuples in figure 4.2(c) show that there are two facts in the fact set of hyper-alert type *FTP\_Syst*. One is *DestIPAddress* with type *varchar(15)* and the other is *DestPort* with type *int*.

Table *HATPrereq* (figure 4.2(d)) stores the hyper-alert type's prerequisite information. It has five attributes: *HyperAlertType*, *P\_ID*, *Predicate*, *ArgPos*, and *ArgName*. They represent the name of a hyper-alert type, id of the prerequisite predicate of this hyper-alert type, predicate name, argument position, argument name, respectively. For example, the two tuples in figure 4.2(d) represent the first predicate of hyper-alert type *FTP\_Syst*'s prerequisites. It is *ExistService*, and this predicate has two arguments: one is *DestIPAddress* and the other is *DestPort*. We can get the data type of these arguments from one of these two tables: *Predicate* or *HATFact*.

Table *HATConseq* (figure 4.2(e)) is similar to *HATPrereq*. The only difference is this table contains the hyper-alert type's consequence information.

### 4.1.2 Knowledge Base XML File

An XML file is designed to provide a way to input the knowledge base and Xerces Java [3] is adopted as the XML parser. There are several advantages of using XML. First, it is easy to read and understand for human users; Second, it is easy to validate by using some existing tools. In a knowledge

```

<Predicates>
  <Predicate Name="ExistHost">
    <Arg id="1" Pos="1" Attr="varchar(15)"></Arg>
  </Predicate>
  <Predicate Name="ExistService">
    <Arg id="2" Pos="1" Attr="varchar(15)"></Arg>
    <Arg id="3" Pos="2" Attr="int"></Arg>
  </Predicate>
  <Predicate Name="GainOSInfo">
    <Arg id="4" Pos="1" Attr="varchar(15)"></Arg>
  </Predicate>
  <Predicate Name="OSUnix">
    <Arg id="5" Pos="1" Attr="varchar(15)"></Arg>
  </Predicate>
  <Predicate Name="VulnerableRPCService">
    <Arg id="6" Pos="1" Attr="varchar(15)"></Arg>
  </Predicate>
</Predicates>

```

**Figure 4.3:** A sample “Predicates” section in a knowledge base XML file.

base XML file, there are basically three sections: *Predicates*, *Implications* and *HyperAlertTypes*.

### Predicates

The *Predicates* section contains a set of *Predicates* which are used to represent the prerequisites and consequences of a hyper-alert type (See 3.1). In the XML knowledge base file, all predicates appeared in the *Implications* section or *HyperAlertTypes* section must be declared in the *Predicates* section. All the predicates defined in this section are saved in the *Predicate* table in the database.

Figure 4.3 shows an example of the *Predicates* section.

Each predicate is represented by the following parts: an attribute *Name*, and zero or more *Arg* elements. The attribute *Name* is used to represent a unique name which works as the key of the predicate. It is saved in the *Predicate* column of the *Predicate* table. Each *Arg* element represents one argument of this predicate. It has three attributes: *id*, *Pos* and *Attr*. In order to make the validation strict and easy, a unique argument *id* is used for each argument in all the predicates. The *Pos* attribute is to tell the position of this argument in the predicate’s argument list, and it goes to the *ArgNum* column of the *Predicate* table; The *Attr* attribute is to tell the data type of this argument, and it goes to the *ArgType* column of the *Predicate* table. The *id* attribute of each argument in the XML file is only used for validation, we don’t save it in database.

For example, the first tuple in figure 4.2(a) represents the predicate “ExistHost” defined in figure

```

<Implications>
  <Implication>
    <ImpliedName>ExistHost</ImpliedName>
    <ImpliedArg id="1"></ImpliedArg>
    <ArgMap>
      <ImpliedArg id="1"></ImpliedArg>
      <ImpliedArg id="2"></ImpliedArg>
    </ArgMap>
  </Implication>
  <Implication Phantom="Yes">
    <ImpliedName>OSUnix</ImpliedName>
    <ImpliedArg id="5"></ImpliedArg>
    <ArgMap>
      <ImpliedArg id="5"></ImpliedArg>
      <ImpliedArg id="4"></ImpliedArg>
    </ArgMap>
  </Implication>
</Implications>

```

**Figure 4.4:** A sample “Implications” section in a knowledge base XML file.

4.3. The second and the third tuple in figure 4.2(a) represent another predicate, “ExistService”, which is also defined in figure 4.3.

## Implications

The *Implications* section contains a set of *Implications*. Implication is used to represent the relationship between predicates. This part is saved in the *Implication* table in the database.

To simplify the implementation, we assume that for all implication of the form  $\forall x_1, x_2, \dots, x_n, y_1, y_2, \dots, y_m \ p_1(x_1, x_2, \dots, x_n) \rightarrow p_2(y_1, y_2, \dots, y_m), \{y_1, y_2, \dots, y_m\} \subseteq \{x_1, x_2, \dots, x_n\}$ . As a result, in order to keep the implication from  $p_1$  to  $p_2$ , we only need to record what arguments of  $p_1$  are mapped to the arguments of  $p_2$ . For example,  $ExistService(IP, port) \rightarrow ExistHost(IP)$  can be represented as the first tuple in figure 4.2(b), which states that the first argument of *ExistHost* is the first argument of *ExistService*.

Figure 4.4 shows a sample *Implications* section in a knowledge base XML file.

In the *Implications* section, there are two kinds of implications: normal and phantom. Normal implications are the implications can be derived by us, the correlators, directly. For example,  $ExistService(IP, port) \rightarrow ExistHost(IP)$  is a normal implication, because we are sure if  $ExistService(IP, port)$  is true,  $ExistHost(IP)$  must be true. This relationship can be derived by us, the correlators, directly. While, the phantom implications mean that the implied predicate is unclear to us, but

it is clear to the attacker. For example, the consequence of *FTP\_Syst* is *GainOSInfo*. Through *FTP\_Syst*, the attacker may get knowledge about what operating system is in the target machine, either *OSUnix* or *OSWindows*, etc. But we, the correlators, cannot get this detailed information. So, we call this kind of implication,  $GainOSInfo(IP) \rightarrow OSUnix(IP)$ , phantom implication. Implication relationships are not complete without the phantom implications, because we should stand on attackers' position and see what information they can get.

In the knowledge base XML file, we use an attribute *Phantom* to indicate whether this implication is a phantom implication or not. The default value is false. We regard an implication as a normal implication if the *Phantom* attribute is not specified. In the correlation process, both phantom implications and normal implications have the same effect and this information isn't stored in the database. We mark it just for the sake of clearness to users. In figure 4.4,  $ExistService(DestIPAddress)$  implies  $ExistHost(DestIPAddress)$  is a normal implication, while  $GainOSInfo(DestIPAddress)$  implies  $OSUnix(DestIPAddress)$  is a phantom implication.

An implication is represented by the following parts: one *ImplyingName*, one *ImpliedName*, and zero or more *ArgMap*. The usage of *ImplyingName* and *ImpliedName* is obvious, they are the names of the implying predicate and the implied predicate, respectively. One *ArgMap* represents the mapping between one pair of arguments. *ImplyingArg* and *ImpliedArg* represent the argument of implying predicate and implied predicate respectively. The attribute *id* of the *ImplyingArg* and *ImpliedArg* is used to refer to the unique *id* of the predicate's argument which is defined in the *Predicates* section. Take  $GainOSInfo$  implies  $OSUnix$  in figure 4.4 as an example. We want to represent  $GainOSInfo(DestIPAddress) \rightarrow OSUnix(DestIPAddress)$ . The argument *id* of  $GainOSInfo$ , which is defined in the *Predicates* section, is 4. And, the argument *id* of  $OSUnix$  is 5. So, the attribute *id* of the *ImplyingArg* and the *ImpliedArg* here are 4 and 5, respectively.

The validation check performed by XML schema is not enough for the implication part. Because what XML schema can check is whether the *ImplyingArg id* and *ImpliedArg id* exist in the *Predicates* section. But this check cannot assure that the implication mappings are valid. It is very important to make sure that the *ImplyingArg* and the *ImpliedArg* have the same data type. This validation check is performed by the tool. Before we write the implication mapping relationship into database, we will get their data types from the *Predicate* table. If they are the same data type, this implication mapping relationship will be written into the table *Implication*. Otherwise, the tool will quit the process.

## HyperAlertTypes

The *HyperAlertTypes* section contains a set of *HyperAlertTypes*. Figure 4.5 shows a sample *HyperAlertTypes* section in the knowledge base XML file.

```

<HyperAlertTypes>
  <HyperAlertType Name="FTP_Syst">
    <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
    <Fact FactName="DestPort" FactType="int"></Fact>
    <Prerequisite>
      <Predicate Name="ExistService">
        <Arg id="2" ArgName="DestIPAddress"></Arg>
        <Arg id="3" ArgName="DestPort"></Arg>
      </Predicate>
    </Prerequisite>
    <Consequence>
      <Predicate Name="GainOSInfo">
        <Arg id="4" ArgName="DestIPAddress"></Arg>
      </Predicate>
    </Consequence>
  </HyperAlertType>
</HyperAlertTypes>

```

**Figure 4.5:** A sample “HyperAlertTypes” section in a knowledge base XML file.

Each *HyperAlertType* is represented by the following parts: one *Fact* element, zero or one *Prerequisite* element, and zero or one *Consequence* element. All these information in the *HyperAlertType* will be stored in three tables: *HATFact* is to store the fact information, *HATPrereq* is to store the prerequisite information and *HATConseq* is to store the consequence information.

*Fact* has two attributes: *FactName* and *FactType*. *FactName* is stored in the *AttrName* column of the table *HATFact* and *FactType* is stored in the *AttrType* of *HATFact*. For example, the two tuples in figure 4.2(c) represent the fact information of the hyper-alert type *FTP\_Syst* defined in figure 4.5.

Each *Prerequisite* or *Consequence* may contain a set of *Predicates*. This *Predicate* has the similar structure with it in the *Predicates* section but small difference. The difference is in the argument part. Instead of *Pos* and *Attr* attributes in the *Predicates* section, it has *ArgName* here to specify the exact argument name. Also, it has an *id* attribute which is used to refer to the *Arg* defined in the *Predicates* section. Since we assume there is no negation and the hyper-alerts are reasoned using prerequisite and consequence sets, we only need to keep the predicates appearing in the prerequisite and the consequence in these two tables.

We take the first hyper-alert type (*FTP\_Syst*) defined in figure 4.5 as an example. Figure 4.2(d) says the hyper-alert type *FTP\_Syst* has one predicate *ExistService* in its prerequisite, and this predicate takes the fact attribute *DestIPAddress* as its first argument, *DestPort* as its second argument. Similarly, figure 4.2(e) says *FTP\_Syst* has one predicate *GainOSInfo* in its consequence, and it takes

the fact attribute *DestIPAddress* as its first (and only) argument.

One important part in parsing the hyper-alert types is to expand the consequence predicate set based on the implications. For example, if a hyper-alert type has only one predicate *GainRootAccess (DestIP)* in its consequence set, and according to the table *Implication*, *GainRootAccess (DestIP)* implies *GainAccess (DestIP)*, then we will add *GainAccess (DestIP)* into the hyper-alert type's consequence set. To take into account indirect implications, this process should be repeated iteratively until no more predicates can be added.

## 4.2 Alert Preprocessor

The alert preprocessor processes the alerts to generate hyper-alerts and instantiate the prerequisite and consequence sets of each hyper-alert.

There are at least two choices in reasoning about the instantiated predicates during the correlation phase. The first is to allow the correlation engine to reason about the related predicates (e.g., *GainRootAccess (IP\_A) → GainAccess (IP\_A)*) using the information stored in the knowledge base directly. Such an approach is probably necessary for real-time correlation of hyper-alerts. However, it requires more development effort and do not take full advantage of the RDBMS's query processing capability. The second approach is to instantiate the predicates specified in the prerequisite and consequence sets as well as those implied by these predicates, and then correlation of the hyper-alerts becomes simple matching of the instantiated predicates. This method requires more storage than the previous one; however, it is suitable for batch processing of the alerts and is able to use the RDBMS's query processing capability. Since we are developing an off-line alert correlation tool, we choose the second method to reduce the development cost.

The alert preprocessing phase generates the hyper-alert data as well as two auxiliary tables. Each hyper-alert is uniquely identified by a hyper-alert ID. Two tables are used to store the hyper-alerts: The table *HyperAlert* has attributes *HyperAlertID*, *HyperAlertType*, *begin\_time* and *end\_time*, which represent the ID, the type, and the interval-based timestamps of the hyper-alerts. The table *HyperAlertFact* has attributes *HyperAlertID* and the union of the fact attributes of all known hyper-alert types; it keeps the fact attribute values for all hyper-alerts, with inapplicable attributes set to NULL.

The two auxiliary tables, *PrereqSet* and *ConseqSet*, are generated to facilitate hyper-alert correlation. They are used to keep instantiated prerequisite and consequence sets of the hyper-alerts, respectively. Both tables have the same set of attributes: *HyperAlertID*, *EncodedPredicate*, *begin\_time*, and *end\_time*; however, *PrereqSet* saves the elements in the prerequisite sets, while *ConseqSet* keeps

the elements in the consequence sets of hyper-alerts. Note that *begin\_time* and *end\_time* are redundant, since this information can be derived from the table *HyperAlert*. However, we replicate them in both *PrereqSet* and *ConseqSet* for performance reasons.

To simplify the correlation process, we encode instantiated predicates as strings. Specifically, each instantiated predicate is encoded as the predicate name followed by the character “(”, followed by the sequence of arguments separated with the character “,”, and finally followed by the character “)”. It is assumed that predicates are uniquely identified by their names and the characters “(”, “)”, and “,” do not appear in predicate names and arguments. Thus, comparing instantiated predicates is equivalent to comparing the encoded strings. For example, if a hyper-alert has two instantiated predicates, *VulnerableSadmin* (152.142.1.19) and *VulnerableSadmin* (152.142.1.52), in its prerequisite set, then it will have two tuples in the table *PrereqSet* with encoded predicates “VulnerableSadmin(152.142.1.19)” and “VulnerableSadmin(152.142.1.52)”, respectively.

### 4.3 Correlation Engine

The basic task of the correlation engine is to find out all the prepare-for relationships between hyper-alerts instantiated by the alert preprocessor. The correlated hyper-alert information is stored in the table *CorrelatedAlerts*, which has only two attributes, *PreparingHyperAlertID* and *PreparedHyperAlertID*. As suggested by the attribute names, each tuple in *CorrelatedAlerts* indicates the hyper-alert with the ID *PreparingHyperAlertID* prepares for the one with the ID *PreparedHyperAlertID*.

With the two auxiliary tables *PrereqSet* and *ConseqSet*, it is trivial to discover the aforementioned prepare-for relations. We use the following SQL query to perform this task. The result of the SQL query is inserted into the *CorrelatedAlerts* table. It is not difficult to verify that the SQL statement discovers all and only the hyper-alert pairs such that the first one of the pair prepares for the second one.

Query used to generate all the prepare-for relationships:

```
SELECT DISTINCT c.HyperAlertID, p.HyperAlertID
FROM PrereqSet p, ConseqSet c
WHERE p.EncodedPredicate = c.EncodedPredicate
AND c.end_time < p.begin_time;
```

### 4.4 Separating Correlation Graphs

After we have all the prepare-for relationships, we need to output them as hyper-alerts correlation graphs. The first thing we should do is to separate all the prepare-for relationships into several

graphs. We use nodes to represent hyper-alerts and edges to represent prepare-for relationships. All the graphs satisfy the following two conditions. First, in any single graph, all nodes are connected with each other. Second, any two graphs don't have any common nodes or common edges. We will refer a prepare-for relationship as an edge in the following sections.

The table *GraphEdges* is the output of this process, which contains prepare-for relationships and their graph id. It has three attributes: *PreparingHyperAlertID*, *PreparedHyperAlertID* and *graphID*. We use the *graphID* as the indicator of different graphs. All the prepare-for relationships with the same graph id are in the same graph.

Two auxiliary tables are used in the process. Table *ProcessedNodes* is used to save all the nodes we have processed so far for this graph. Table *NewlyAddedNodes* saves all the nodes which are added into *ProcessedNodes* in the previous round. Both of these two tables only contain one attribute, which is *HyperAlertID*.

We set 0 to *graphID* for all prepare-for relationships initially. From the *graphID*, we know whether this edge has been assigned a graph or not. If it is 0, it hasn't been processed yet. If it is greater than 0, it has already been assigned.

When we start a new graph, clear the table *ProcessedNodes* first. Then, select any unassigned edge whose *graphID*=0 and assign a new graph id  $g$  ( $g=1,2,3,\dots$ ) to this edge. Insert the two nodes of this edge into table *ProcessedNodes*.

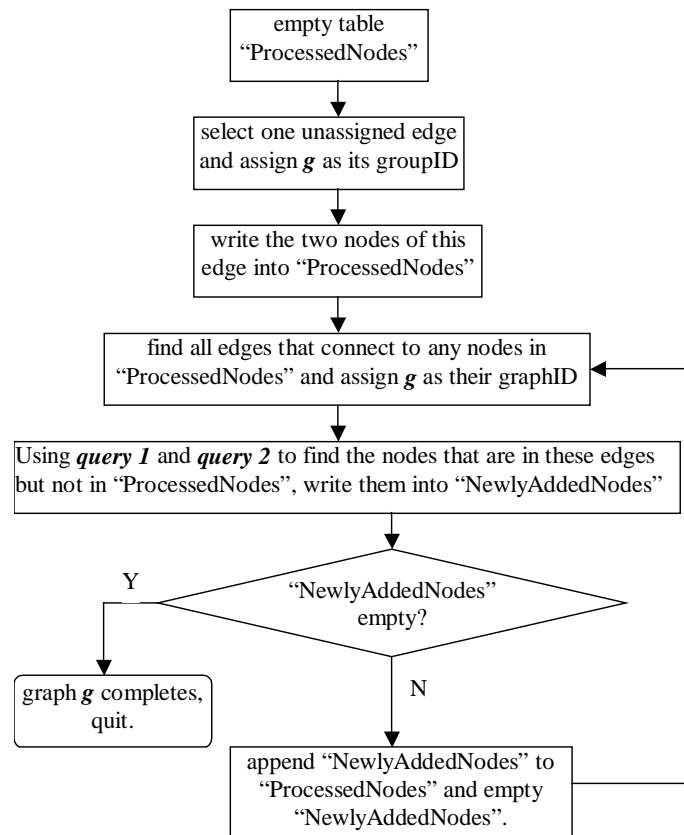
We will repeat the following process to find out all the unassigned edges that connect to each other. First, find all the unassigned edges that their *PreparingHyperAlertIDs* are in the table *ProcessedNodes* and set their *graphID* to  $g$ ; Also, find all the unassigned edges that their *PreparedHyperAlertIDs* are in the table *ProcessedNodes* and set *graphID* to  $g$ . Up to now, we have found all the edges that connect with the nodes in *ProcessedNodes*. We need update table *ProcessedNodes* and *NewlyAddedNodes*. So, select the nodes whose *graphID* is  $g$  but the nodes themselves are not in the set *ProcessedNodes*. That is, one node of an edge isn't in *ProcessedNodes*, but the other node is already in it. Replace *NewlyAddedNodes* with these nodes. Append the nodes in *NewlyAddedNodes* into *ProcessedNodes*. Repeat the above process until that *NewlyAddedNodes* is null. Thus, one single graph  $g$  completes.

Figure 4.6 is the algorithm we use to generate a single graph with graph id is  $g$ .

Here are two queries used in figure 4.6.

Query1 (in figure 4.6):

```
INSERT INTO NewlyAddedNodes
SELECT DISTINCT PreparingHyperAlertID
FROM GraphEdges
WHERE PreparedHyperAlertID IN (SELECT * FROM ProcessedNodes)
```



**Figure 4.6:** The algorithm used to generate a single graph.

```
AND PreparingHyperAlertID NOT IN (SELECT * FROM ProcessedNodes);
```

Query2 (in figure 4.6):

```
INSERT INTO NewlyAddedNodes
SELECT DISTINCT PreparedHyperAlertID
FROM GraphEdges
WHERE PreparingHyperAlertID IN (SELECT * FROM ProcessedNodes)
AND PreparedHyperAlertID NOT IN (SELECT * FROM ProcessedNodes);
```

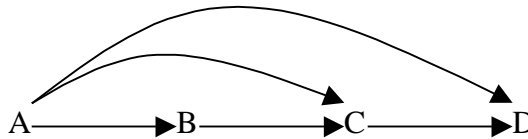
## 4.5 Transitive Edge Exclusion

In the prepare-for relationship, there may exist such condition that A prepares for B, B prepares for C and A prepares for C also (see figure 4.7). If we think the relationship can propagate, then A prepares for C can be got from A prepares for B and B prepares for C. If there exists a lot of such relationships, it will make the correlation graph hard to read and understand. It is better that we take these out from the prepare-for relationships. If we think it in the graph, A prepares for C in the above example is a transitive edge. So, we call this process *Transitive Edge Exclusion*. We call the edge  $A \rightarrow C$  is a 2-hop edge and  $A \rightarrow D$  is a 3-hop edge in figure 4.7.

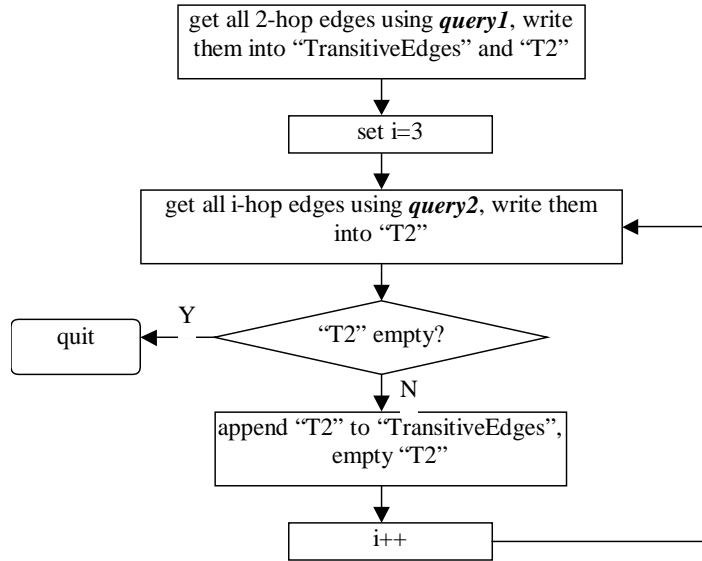
Here is the way we do the *Transitive Edge Exclusion*. We will use the notation  $A \rightarrow B$  to represent that A prepares for B. The basic idea is we enumerate all the possible transitive edges which may exist in this graph. For example, if figure 4.7 represents all the prepare-for relationships, then all the possible transitive edges in figure 4.7 are:  $A \rightarrow C$ ,  $A \rightarrow D$  and  $B \rightarrow D$ . We save this result to an auxiliary table *TransitiveEdges*. It is used to keep all the possible transitive edges. This table has two attributes: *PreparingHyperAlertID* and *PreparedHyperAlertID*. Then, delete the edges which exist in both prepare-for relationships and the table *TransitiveEdges*. So,  $A \rightarrow C$  and  $A \rightarrow D$  will be deleted in the above example.

To get all the possible transitive edges, we first find out all the possible 2-hop edges  $A \rightarrow C$  from the prepare-for relationships. These edges match the condition that there exists a node B that  $A \rightarrow B$  and  $B \rightarrow C$  are both existed in the prepare-for relationships. Store this result into *TransitiveEdges* and *T2*. *T2* is another auxiliary table which keeps the result generated by the previous round. It has the same attributes of table *TransitiveEdges*. Then, we will repeat a same process to find out all the 3-hop transitive edges, 4-hop transitive edges, and so on, until there is no more new transitive edges.

Suppose we want to find all the  $n$ -hop edges ( $n=3, 4, \dots$ ). Because *T2* contains all the possible  $(n-1)$ -hop edges, all the edges  $A \rightarrow C$  are  $n$ -hop edges if  $A \rightarrow B$  is existed in *T2* and  $B \rightarrow C$  is existed in the prepare-for relationship.



**Figure 4.7:** Transitive edges.



**Figure 4.8:** The algorithm used to generate all the possible transitive edges.

Replace  $T2$  with this set of results and append it to  $TransitiveEdges$ . We will stop here if  $T2$  is null. Otherwise, we will continue to find all the  $(n+1)$ -hop edges.

After we have the table  $TransitiveEdges$  which contains all the transitive edges that may exist in the prepare-for relationships, it is very easy to delete them from the prepare-for relationships.

Figure 4.8 is the algorithm we use to get all the possible transitive edges.

Query1 (generate all the 2-hop transitive edges in figure 4.8):

```

SELECT c1.PreparingHyperAlertID, c2.PreparedHyperAlertID
FROM prepare-for-relationships c1, prepare-for-relationships c2
WHERE c1.PreparedHyperAlertID = c2.PreparingHyperAlertID
AND c1.PreparingHyperAlertID != c2.PreparedHyperAlertID;
  
```

Query2 (generate all the n-hop transitive edges in figure 4.8 where n is greater than 2):

```

SELECT T.PreparingHyperAlertID, R.PreparedHyperAlertID
FROM T2 T, prepare-for-relationships R
WHERE T.PreparedHyperAlertID=R.PreparingHyperAlertID
AND T.PreparingHyperAlertID != R.PreparedHyperAlertID
AND NOT EXISTS (SELECT * FROM TransitiveEdges TE
  
```

```
WHERE T.PreparingHyperAlertID = TE.PreparingHyperAlertID
AND R.PreparedHyperAlertID = TE.PreparedHyperAlertID);
```

## 4.6 Adjustable Graph Reduction

As we discussed before, graph reduction is used to reduce the complexity of a graph. This utility not only can be applied on graphs generated by original prepare-for relationships, but also it can be applied on the results generated by focused analysis and graph decomposition.

In this implementation, we use the interval constraint to do the graph reduction. That is, if the time interval between two alerts are less than the threshold user provides, these two alerts are aggregated together.

Prepare-for relationships are browsed according to their graph id. For each graph, it will aggregate the same type of hyper-alerts based on their time interval. First, the hyper-alerts are clustered by different hyper-alert types. For each type of hyper-alerts (one cluster), they are sorted by their timestamps. Among each cluster, if the interval between the two adjacent alerts is less than the threshold, then these two adjacent alerts can be aggregated into one alert. One auxiliary table *IDMapping* is used to keep the mapping relationship between the original alert id and the new aggregated alert id.

Once finishing aggregation, the next step is to build up the aggregated prepare-for relationship. There will be an aggregated edge between two aggregated alerts *A1* and *A2*, if there is an original edge between two original alerts *O1* and *O2*, such that *O1* has been aggregated into *A1* and *O2* has been aggregated into *A2*.

Figure 4.9 is the algorithm we use to do the adjustable graph reduction.

It can have different input and output when this utility is applied in different circumstances. The input of this utility could be either the table *GraphEdges*, *GraphEdgesFocus* or *GraphEdgesDecomposed* depending on where this utility is used. *GraphEdges* is the original prepare-for relationship generated by the correlation engine, *GraphEdgesFocus* is the result generated by the focused analysis and *GraphEdgesDecomposed* is the result generated by the graph decomposition.

The output of this utility could be either the table *SimplifiedGraphEdges*, *SimplifiedFocus*, or *SimplifiedDecomposed*. All these three tables have the same attributes with table *GraphEdges*. The attributes are: *PreparingHyperAlertID*, *PreparedHyperAlertID* and *graphID*. The difference is that the alert id in *GraphEdges* is the original hyper-alert id, while the alert id in the other three tables is the new aggregated id.

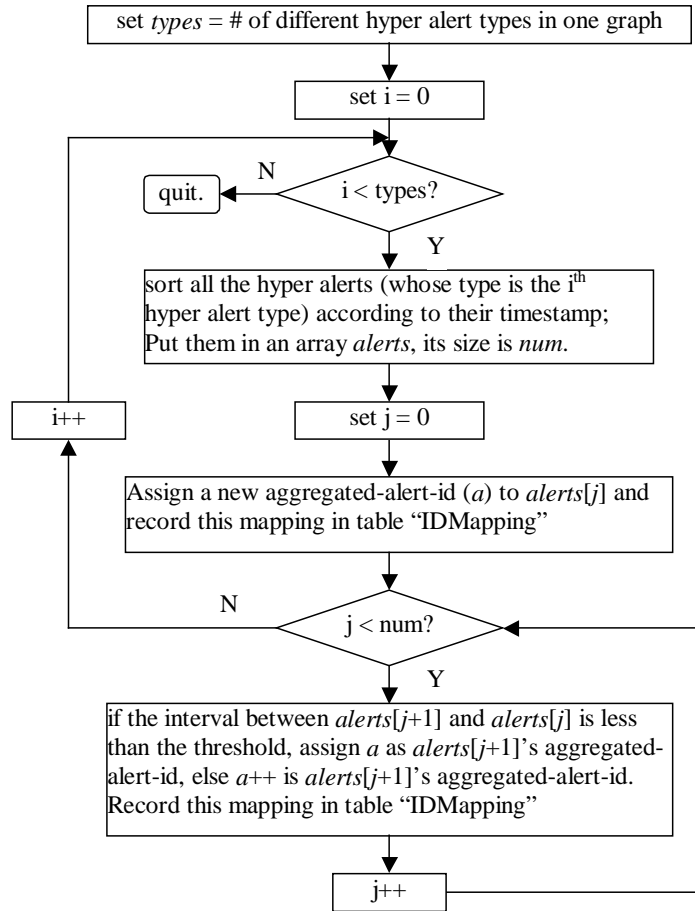


Figure 4.9: The algorithm used to do the adjustable graph reduction.

## 4.7 Focused Analysis

After correlation, it is still possible that the graph is too large to understand if the dataset is huge. If this happens and you have some rough knowledge of the graphs, you can use this utility to facilitate your analysis. The user is requested to input the focusing constraint for this utility. For example, there are critical services running on the computer with IP “152.1.2.3”, the user wants to analyze the attacks targeted at this machine. So,  $DestIPAddress = '152.1.2.3'$  would be the focusing constraint. This focusing constraint must follow the SQL syntax and the attribute (e.g.,  $DestIPAddress$ ) must be a valid one in the knowledge base.

The tool takes input from the table  $GraphEdges$  which contains all the prepare-for relationships

and graph information, applies the focusing constraint onto it, and output the result into the table *GraphEdgesFocus*. Table *GraphEdgesFocus* has the same attributes with table *GraphEdges*.

The implementation of this utility is kind of easy. We just plug in the focusing constraint and the graph id provided by the user into the following SQL query statement.

Query:

```
INSERT INTO GraphEdgesFocus
SELECT * FROM GraphEdges
WHERE graphID = inputGraphID
AND (PreparingHyperAlertID IN
(SELECT HyperAlertID FROM HyperAlertFact WHERE inputFocusingConstraint)
AND PreparedHyperAlertID IN
(SELECT HyperAlertID FROM HyperAlertFact WHERE inputFocusingConstraint));
```

## 4.8 Graph Decomposition

As explained in the previous chapter, graph decomposition is to decompose a graph according to the common features shared by hyper-alerts. That is, we will cluster the alerts into clusters based on the clustering constraint. The constraint could be the same destination IP address, same source IP address, etc. If any two alerts match the constraint, they fall into the same cluster. Each cluster could be viewed as a sub-graph of the original graph.

Since we will plug in the clustering constraint into an SQL statement, the user is requested to provide it in an SQL valid presentation. Not only it must be SQL valid, it must also be a relationship between *h1* and *h2* because we use *h1* and *h2* as two alerts in the query. For example, if user wants to decompose a graph based on alerts' source IP addresses, then,  $h1.SrcIPAddress = h2.SrcIPAddress$  would be the clustering constraint; If user wants to decompose a graph based on both source IP addresses and destination IP addresses, then,  $h1.SrcIPAddress = h2.SrcIPAddress$  and  $h1.DestIPAddress = h2.DestIPAddress$  is the clustering constraint. Also, these features (*SrcIPAddress*, *DestIPAddress*) must be the valid attributes in the knowledge base.

We use a linked list to keep all the alerts that belong to the same cluster and use a dynamic array to keep all the linked lists. Each item in the array contains the head of a linked list. We record each alert's alert id in the linked list. So the size of the dynamic array is the number of sub-graphs after the graph decomposition, and the size of each linked list is the number of alerts existed in each sub-graph.

First, we generate all the distinct hyper-alerts in the graph by using the following query:

```

SELECT DISTINCT PreparingHyperAlertID AS HyperAlertID
FROM GraphEdges WHERE graphID = user-input-graphID
UNION
SELECT DISTINCT PreparedHyperAlertID AS HyperAlertID
FROM GraphEdges WHERE graphID = user-input-graphID;

```

The result of this query is saved in an auxiliary table *GraphAlerts* which has one attribute *HyperAlertID*.

We name two alerts an *alert pair* if the two alerts match the graph clustering constraint and can be put into the same cluster. Then, we find out all such *alert pairs* existed in the graph by using the following query:

```

SELECT h1.HyperAlertID, h2.HyperAlertID
FROM GraphAlerts a, HyperAlertFact h1, HyperAlertFact h2
WHERE h1.HyperAlertID IN (SELECT * FROM a)
AND h2.HyperAlertID IN (SELECT * FROM a)
AND h1.HyperAlertID NOT = h2.HyperAlertID
AND user-input-constraint;

```

An auxiliary table *AlertPairs* is used to keep all these pairs. It has three attributes: *alert1*, *alert2* and *assigned*. The *alert1* and *alert2* are used to keep the alert pair information. The attribute *assigned* is used as an indicator of whether this alert pair has been assigned to a sub-graph. We set it to 0 initially, which means this alert pair hasn't been assigned into cluster yet. When it is assigned, we set it to 1.

The algorithm in figure 4.10 is then used to cluster the alerts into different clusters. The auxiliary table *NewlyAddedAlerts* has one attribute *HyperAlertID*.

Query1 in figure 4.10:

```

SELECT alert1, alert2 FROM AlertPairs
WHERE assigned = 0
AND (alert1 IN (SELECT * FROM NewlyAddedAlerts)
OR alert2 IN (SELECT * FROM NewlyAddedAlerts));

```

Query2 in figure 4.10:

```

SELECT DISTINCT alert1 FROM AlertPairs
WHERE assigned = 0
AND alert1 NOT IN (SELECT * FROM NewlyAddedAlerts)

```

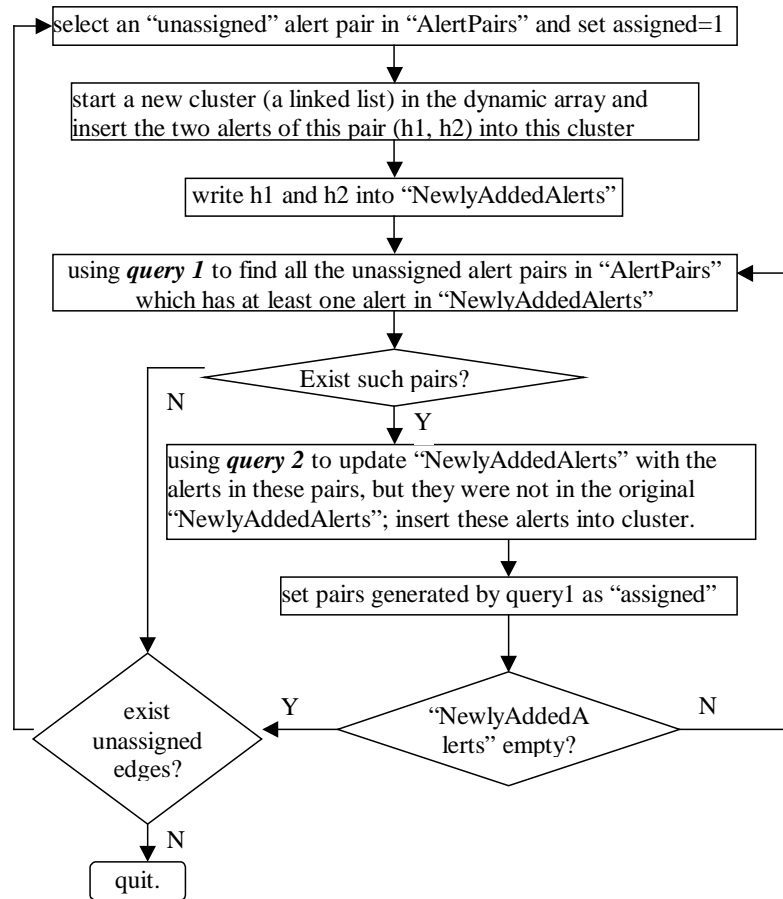


Figure 4.10: The algorithm used to do the graph decomposition.

```

AND alert2 IN (SELECT * FROM NewlyAddedAlerts)
UNION
SELECT DISTINCT alert2 FROM AlertPairs
WHERE assigned = 0
AND alert2 NOT IN (SELECT * FROM NewlyAddedAlerts)
AND alert1 IN (SELECT * FROM NewlyAddedAlerts);

```

After we set all the alerts into clusters, we can assign sub-graph ID to each prepare-for relationship. Each cluster has a unique cluster ID, we can use this cluster ID as the sub-graph ID. One thing we need to keep in mind is that it is possible that two alerts of a prepare-for relationship belong to

different clusters (sub-graphs). An auxiliary table *partGraphEdges* is used in assigning sub-graph id. It has four attributes: *PreparingHyperAlertID*, *PreparedHyperAlertID*, *PreparingGraphID* and *PreparedGraphID*. We use *PreparingGraphID* to represent the sub-graph id of the preparing hyper-alert and use *PreparedGraphID* to represent the sub-graph id of the prepared hyper-alert.

First, we copy the prepare-for relationships from table *GraphEdges* to table *partGraphEdges*. We don't copy all the prepare-for relationships, but the prepare-for relationships of one graph which is specified by the user. We set both *PreparingGraphID* and *PreparedGraphID* to 0 initially. Then, set *PreparingGraphID* and *PreparedGraphID* separately based on which cluster that the preparing hyper-alert and the prepared hyper-alert belongs to, respectively.

Only the prepare-for relationships which have the same *PreparingGraphID* and *PreparedGraphID* are kept. We save them into the table *GraphEdgeDecomposed* which has three attributes: *PreparingHyperAlertID*, *PreparedHyperAlertID* and *graphID*. The other prepare-for relationships which has different *PreparingGraphID* and *PreparedGraphID* are broken into different sub-graphs. They are actually some isolated nodes in the sub-graphs.

## 4.9 Important Tables

Here is the summarization of the important tables generated by the tool.

*CorrelatedAlerts*: It keeps the original prepare-for relationships, which are the results from the alert correlation engine.

*GraphEdges*: Basically, it has one more column based on *CorrelatedAlerts*, which is the graph id information. This table has the information that which graph does each prepare-for relationship belong to.

*GraphEdgesFocus*: It is the result generated by focused analysis. Actually, it is a subset of *GraphEdges*.

*GraphEdgesDecomposed*: Similar with *GraphEdgesFocus*, it is a subset of *GraphEdges* and is generated by graph decomposition.

*SimplifiedGraphEdges*: This table keeps the results when we apply the adjustable graph reduction utility upon the original hyper-alerts prepare-for relationships. Its structure is the same with *GraphEdges*. The difference between them is, *GraphEdges* is the graph information based on original prepare-for relationships but *SimplifiedGraphEdges* is the graph information based on aggregated prepare-for relationship. That is, *GraphEdges* has the original hyper-alert id, while *SimplifiedGraphEdges* has the aggregated hyper-alert id.

*SimplifiedFocus*: This table keeps the results when we apply the adjustable graph reduction utility upon the focused analysis result.

*SimplifiedDecomposed*: This table keeps the results when we apply the adjustable graph reduction utility upon the graph decomposition result.

Also, there are three tables which contain the mapping relationships between original hyper-alert ID and the aggregated hyper-alert ID. *IDMappingA* is used when the graph reduction utility is applied upon the original prepare-for relationships; *IDMappingF* is used when the graph reduction utility is applied upon the focused analysis result, and *IDMappingD* is used when it is applied upon the graph decomposition.

So, pure graph reduction utility uses the following tables: *GraphEdges*, *IDMappingA*, and *SimplifiedGraphEdges*; When graph reduction is applied upon focused analysis, it uses: *GraphEdgesFocus*, *IDMappingF*, and *SimplifiedFocus*; When graph reduction is applied upon graph decomposition, it uses: *GraphEdgesDecomposed*, *IDMappingD*, and *SimplifiedDecomposed*.

## 4.10 Property File

To make this tool easy to use, easy to integrate with other tools and easy to upgrade, we use a property file to specify the required parameters used in each functionality.

Basically, the property file is divided into 7 sections. Section 1 contains the database related parameters. Because this tool is implemented with Java, JDBC is used as the bridge to communicate with database. User can change the JDBC driver and database URL based on the environment. Section 2 contains the knowledge base related parameters. User can specify the path of the knowledge base XML file in this section so that the tool will parse this XML file and save the content as the knowledge base. Section 3 contains the parameters for the correlation engine. User needs to tell the tool where the original alerts are (what is the table name of the alerts generated by IDS), the column names the tool cares about, and so on. Section 4, 5 and 6 contain the parameters for the three utilities: adjustable graph reduction, focused analysis and graph decomposition, respectively. User can specify which graph needs to be analyzed, what constraint will be used, and so on. Section 7 is an optional one. We assume that the hyper-alerts have the same name with the original alerts if not specified. But we also provide this section as an alternative way to accommodate different names between hyper-alerts and original alerts.

Detailed information about the property file can be found in the appendix.

## Chapter 5

# Experimental Results

To evaluate the effectiveness of our method in constructing attack scenarios and its ability to differentiate true and false alerts, some experiments on different datasets are performed. The results are very good. One dataset is DARPA evaluation dataset 2000 [19], the other one is DEFCON CTF 8 [2]. For the DARPA dataset, we not only uncover the attack scenario, we also reduce the false alarm rate. For the DEFCON dataset, there are around 65,000 alerts generated by IDS, it is unhandlable by any system administrator. By applying our utilities, we can help the administrator to deal with the large number of alerts and discover the attack scenario behind it. I will discuss these two results in detail in the following sections.

### 5.1 Experiment Setup

In our experiments, we used NetPoke<sup>1</sup> to replay the network traffic in an isolated network monitored by a RealSecure Network Sensor 6.0 [2]. In all the experiments, 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. Our alert correlator was then used to process the alerts to discover the hyper-alert correlation graphs.

In these experiments, we mapped each alert type reported by the RealSecure Network Sensor to a hyper-alert type (with the same name). The prerequisite and consequence of each hyper-alert type were specified according to the descriptions of the attack signatures provided with the RealSecure Network Sensor 6.0. The hyper-alert types (as well as the implication relationships between predicates) can be found in the appendix.

---

<sup>1</sup>NetPoke is a utility to replay packets to a live network that were previously captured with the tcpdump program. [http://www.ll.mit.edu/IST/ideval/tools/tools\\\_index.html](http://www.ll.mit.edu/IST/ideval/tools/tools\_index.html)

## 5.2 Experiment on DARPA 2000 dataset

There are two separate datasets in DARPA 2000 evaluation data, LLDOS 1.0 and LLDOS 2.0.2. LLDOS 1.0 contains a series of attacks in which an attacker probes, breaks-in, installs the components necessary to launch a Distributed Denial of Service (DDOS) attack, and actually launches a DDOS attack against an off-site server. LLDOS 2.0.2 includes a similar sequence of attacks run by an attacker who is a bit more sophisticated than the first one.

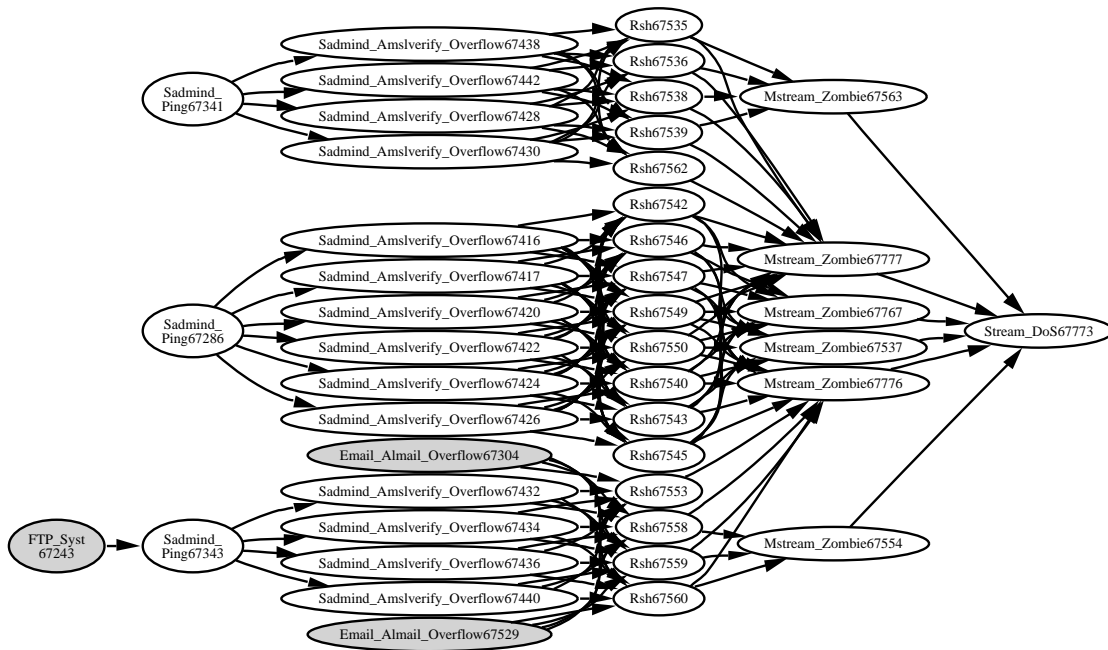
Each dataset includes the network traffic collected from both the DMZ and the inside part of the evaluation network. We have performed four sets of experiments, each with either the DMZ or the inside network traffic of one dataset.

Let's review the scenarios briefly. In both scenarios, the attacker tries to use the vulnerability of *Sadmind* RPC service and launches buffer overflow attacks against the vulnerable hosts. The attacker installs the *mstream* distributed DOS software after he breaks into the hosts successfully. And finally, the attacker launches DDOS attacks from the victims. The differences between these two scenarios lay in two aspects: First, the attacker uses *IPSweep* and *Sadmind\_Ping* to find out the vulnerable hosts in LLDOS1.0 while *DNS\_HInfo* is used in LLDOS2.0.2; second, the attacker attacks each host individually in LLDOS1.0, while in LLDOS2.0.2, the attacker breaks into one host first and then fans out from it.

### 5.2.1 Effectiveness of Alert Correlation

Our first goal of these experiments is to evaluate the effectiveness of our method in constructing attack scenarios from alerts. Figures 5.1, 5.2, 5.3 and 5.4 are the hyper-alert correlation graphs we have generated for this experiment. Each node in Figure 5.1 represents a hyper-alert. The text inside the node is the name of the hyper-alert type followed by the hyper-alert ID.

Let's take figure 5.1 as an example. It shows the (only) hyper-alert correlation graph discovered from the inside network traffic in LLDOS 1.0. There are 44 hyper-alerts in this graph, including 3 false alerts, which are shown in gray. The true hyper-alerts can be divided into five stages horizontally. The first stage consists of three *Sadmind\_Ping* alerts, which the attacker used to find out the vulnerable *Sadmind* services. The three alerts are from the source IP address 202.077.162.213, and to the destination IP addresses 172.016.112.010, 172.016.115.020, and 172.016.112.050, respectively. The second stage consists of fourteen *Sadmind\_Amslverify\_Overflow* alerts. According to the description of the attack scenario, the attacker tried three different stack pointers and two commands in *Sadmind\_Amslverify\_Overflow* attacks for each victim host until one attempt succeeded. All the above three hosts were successfully broken into. The third stage consists of some *Rsh* alerts, with which the attacker installed and started the *mstream* daemon and master programs. The fourth



**Figure 5.1:** The (only) hyper-alert correlation graph discovered in the inside network traffic of LLDOS 1.0.

stage consists of alerts corresponding to the communications between the DDOS master and daemon programs. Finally, the last stage consists of the DDOS attack.

We can see clearly that the hyper-alert correlation graph reveals the structure as well as the high-level strategy of the sequence of attacks.

This hyper-alert correlation graph is still not perfect. First, the two *Email\_Almail\_Overflow* hyper-alerts (shown in gray in Figure 5.1) are false alerts, and are mis-correlated with the *Rsh* alerts, though it is possible that an attacker uses these attacks to gain access to the victim system and then copy the DDOS program with *Rsh*. Second, the *FTP\_Syst* hyper-alert is also a false one; it is correlated with one of the *Sadmind\_Pings*, because an attacker may use *FTP\_Syst* to gain the OS information and then launch an *Sadmind\_Ping* attack.

Third, the attacker used a series of *telnet* as a part of the sequence attacks, and RealSecure detects one of them. But this graph does not include the corresponding hyper alert. The reason we miss the *telnet* alert is because we cannot tell whether the *telnet* is a normal user behavior or an attacker behavior. Fourth, because our correlator depends on the underlying IDS for alert information, if the underlying IDS misses some alerts, we may also miss them; If the underlying IDS reports more false alerts, we also have a chance to correlate more. In this example, RealSecure

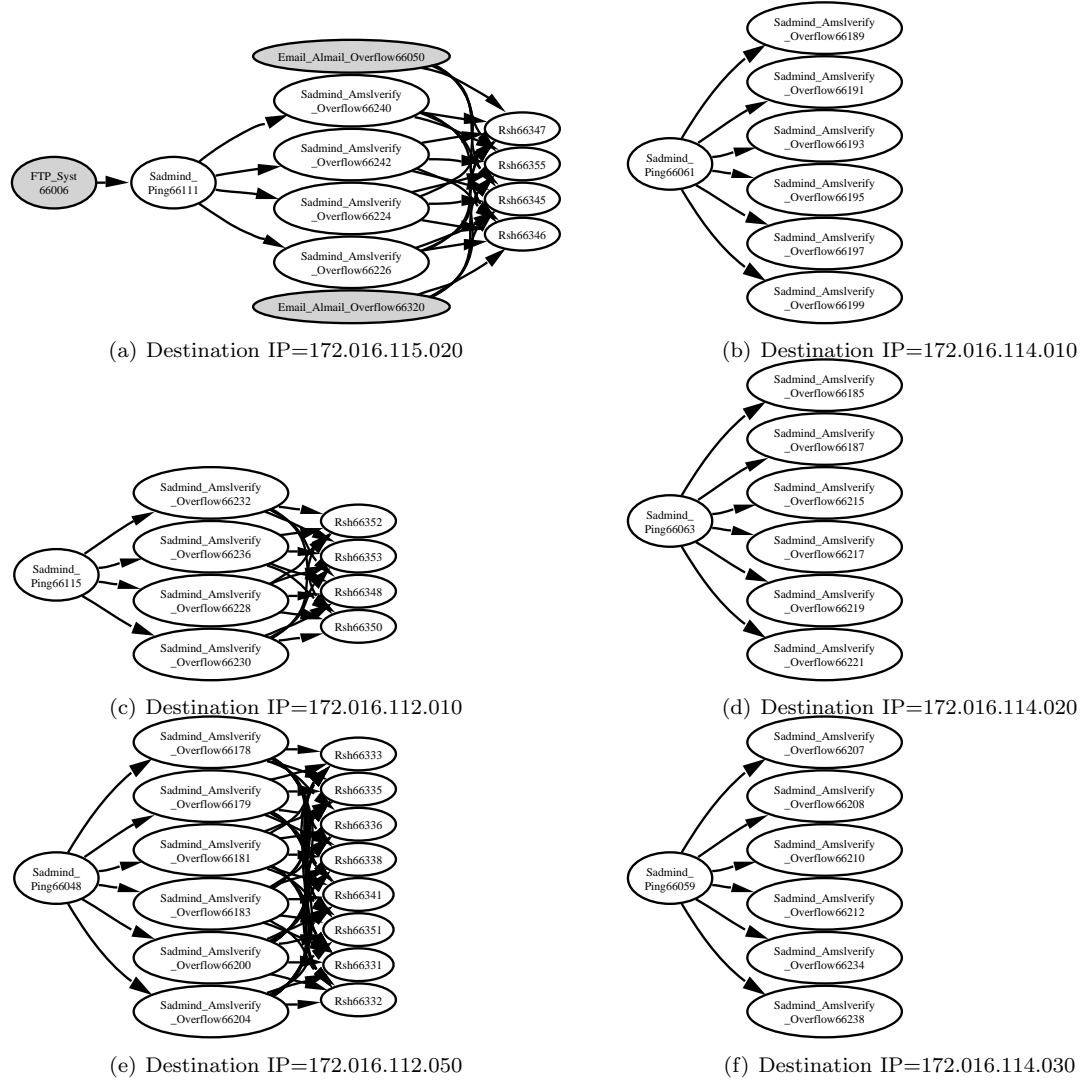
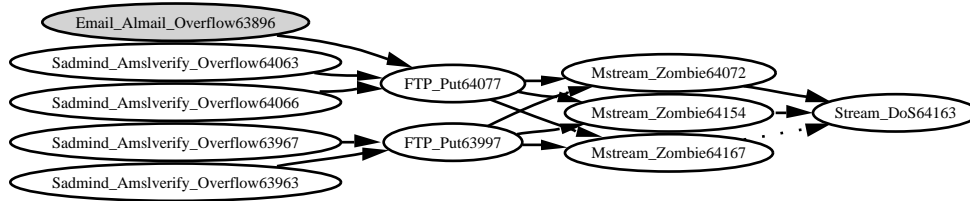
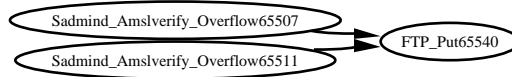


Figure 5.2: The hyper-alert correlation graphs discovered in the dmz network traffic of LLDOS1.0



**Figure 5.3:** The hyper-alert correlation graph discovered in the inside network traffic of LLDOS 2.0.2.



**Figure 5.4:** The hyper-alert correlation graph discovered in the dmz network traffic of LLDOS 2.0.2.

misses the IPSweep, so do us. RealSecure generates several duplicated Rsh alerts, so do us. But we can eliminate this duplication. In this experiment, we haven't used the time constraint, we just simply map one alert to one hyper-alert. If we use the time constraint, for example, we set the time interval constraint  $t=0$ , we can map the alerts with same timestamps to one hyper-alert and remove the duplicated alerts. In this way, we can decrease the false alarm rate more. Finally, not all of the *Mstream-Zombie* alerts prepare for the *Stream-DoS* alert. Only two of them with alert ID 67776 and 67777 prepare for *Stream-DoS* directly. What the other four *Mstream-Zombie* alerts do is to let the handler record them in the known agents list. They are part of the attack, but they don't lead to *Stream-DoS* alert directly. The command sent from handler to agent has the default destination port number 7983, while the command from agent to handler uses 9325 as its default destination port number. If we consider this factor in our experiment, we can distinguish these two kinds of *Mstream-Zombie* alerts.

To better understand the effectiveness of our method, we examine the *completeness* and the *soundness* of alert correlation. The completeness of alert correlation assesses how well we can correlate related alerts together, while the soundness evaluates how correctly the alerts are correlated. We introduce two simple measures,  $R_c$  and  $R_s$ , to quantitatively evaluate completeness and soundness, respectively:

$$R_c = \frac{\#correctly\ correlated\ alerts}{\#related\ alerts},$$

**Table 5.1:** Completeness and soundness of alert correlation.

	LLDOS 1.0		LLDOS 2.0.2	
	DMZ	Inside	DMZ	Inside
# correctly correlated alerts	54	41	5	12
# related alerts	57	44	8	18
# correlated alerts	57	44	5	13
completeness measure $R_c$	94.74%	93.18%	62.5%	66.7%
soundness measure $R_s$	94.74%	93.18%	100%	92.3%

and

$$R_s = \frac{\# \text{correctly correlated alerts}}{\# \text{correlated alerts}}.$$

Counting the numbers in  $R_c$  and  $R_s$  is easy, given the description of the attacks in the DARPA datasets. However, RealSecure generated duplicate alerts for several attacks. In our experiments, we counted the duplicate alerts as different ones. False alerts are counted (as incorrectly correlated alerts) so long as they are correlated. Though non-intrusive alerts (e.g., the above *Email\_Ehlo* and *Email\_Turn*) are not attacks, if they are related activities, we counted them as correctly correlated ones.

Table 5.1 shows the results about completeness and soundness of the alert correlation for the two datasets. As shown by the values of  $R_s$ , most of the hyper-alerts are correlated correctly. The completeness measures ( $R_c$ ) are satisfactory for LLDOS 1.0; however, they are only 62.5% and 66.7% for the DMZ and inside traffic in LLDOS 2.0.2. Our further analysis reveals that all the hyper-alerts missed are those triggered by the *telnets* that the attacker used to access a victim host. Each *telnet* triggered three alerts, *TelnetEnvAll*, *TelnetXDisplay* and *TelnetTerminalType*. According to RealSecure’s description, these alerts are about attacks that are launched using environmental variables (*TelnetEnvAll*) in a telnet session, including XDisplay (*TelnetXDisplay*) and TerminalType (*TelnetTerminalType*). However, according to the description of the datasets, the attacker did not launch these attacks, though he did telnet to one victim host after gaining access to it. Nevertheless, to be conservative, we consider them as related alerts in our evaluation. Considering these facts, we can conclude that our method is effective for these datasets.

### 5.2.2 Ability to Differentiate Alerts

Our second goal of these experiments is to see how well alert correlation can be used to differentiate false alerts and true alerts. As we conjectured in Chapter 3, false alerts, which do not correspond to any real attacks, tend to be more random than the actual alerts, and are less likely to be correlated to others. If this conjecture is true, we can divert more resources to deal with correlated alerts, and thus improve the effectiveness of intrusion response.

**Table 5.2:** Ability to differentiate true and false alerts

Dataset		# observable attacks	Tool	# alerts	# detected attacks	Detection rate	# true alerts	False alert rate
LLDOS 1.0	DMZ	89	RealSecure	891	51	57.30%	57	93.60%
			Our method	57	50	56.18%	54	5.26%
	Inside	60	RealSecure	922	37	61.67%	44	95.23%
			Our method	44	36	60%	41	6.82%
LLDOS 2.0.2	DMZ	7	RealSecure	425	4	57.14%	6	98.59%
			Our method	5	3	42.86%	3	40%
	Inside	15	RealSecure	489	12	80.00%	16	96.73%
			Our method	13	10	66.67%	10	23.08%

To understand this issue, we deliberately drop the uncorrelated alerts and then compare the resulting detection rate and false alert rate with the original ones of RealSecure.

We counted the number of actual attacks and false alerts according to the description included in the datasets. The initial phase of the attacks involved an IP Sweep attack. Though many packets were involved, we counted them as a single attack. Similarly, the final phase had a DDOS attack, which generated many packets but was also counted as one attack. For the rest of the attacks, we counted each action (e.g., *telnet*, *Sadmin\_Ping*) initiated by the attacker as one attack. The numbers of attacks observable in these datasets are shown in Table 5.2. Note that some activities such as *telnet* are not usually considered as attacks; however, we counted them if the attacker used them as part of the attacks.

RealSecure Network Sensor 6.0 generated duplicate alerts for certain attacks. For example, the same *rsh* connection that the attacker used to access the compromised host triggered two alerts. As a result, the number of true alerts (i.e., the alerts corresponding to actual attacks) is greater than the number of detected attacks. The detection rates were calculated as  $\frac{\# \text{detected attacks}}{\# \text{observable attacks}}$ , while the false alert rates were computed as  $(1 - \frac{\# \text{true alerts}}{\# \text{alerts}})$ .

Table 5.2 summarizes the results of these experiments. These results show that discarding uncorrelated alerts reduces the false alert rates greatly without sacrificing the detection rate too much. Thus, it is reasonable to treat correlated alerts more seriously than uncorrelated ones. However, simply discarding uncorrelated alerts is dangerous, since some of them may be true alerts, which correspond to individual attacks or attacks our method fails to correlate.

### 5.3 Experiment on Defcon 8 dataset

As demonstrated in the experiment with DARPA dataset, the alert correlation method is effective in analyzing small amount of alerts. However, our experience with intrusion intensive datasets (e.g., the DEF CON 8 CTF dataset) has revealed several problems. So, we applied the three utilities we

**Table 5.3:** General statistics of the initial analysis

# total hyper-alert types	115	# total hyper-alerts	65054
# correlated hyper-alert types	95	# correlated	9744
# uncorrelated hyper-alert types	20	# uncorrelated	55310
# partially correlated hyper-alert types	51	% correlated	15%

**Table 5.4:** Statistics of top 10 uncorrelated hyper-alert types.

Hyper-alert type	# uncorrelated alerts	# correlated alerts	Hyper-alert type	# uncorrelated alerts	# correlated alerts
IPHalfScan	33745	958	Windows_Access_Error	11657	0
HTTP_Cookie	2119	0	SYNFlood	1306	406
IPDuplicate	1063	0	PingFlood	1009	495
SSH_Detected	731	0	Port_Scan	698	725
ServiceScan	667	2156	Satan	593	280

mentioned earlier. It turned out that they are very useful.

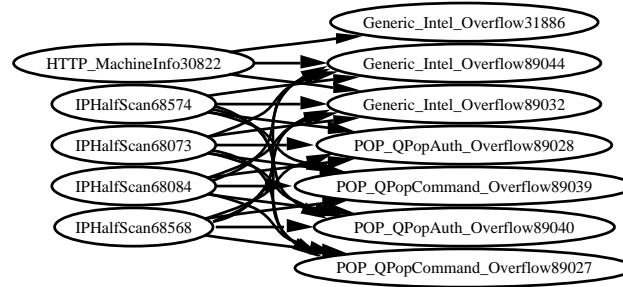
It would be helpful for the evaluation of our method if we could identify false alerts, alerts for sequences of attacks, and alerts for isolated attacks. Unfortunately, due to the nature of the dataset, we are unable to obtain any of them. Thus, in these experiments, we focus on the analysis of the attack strategies reflected by hyper-alert correlation graphs, but only discuss the uncorrelated alerts briefly.

### 5.3.1 Initial Attempt

In our initial analysis of the DEF CON 8 CTF dataset, we tried to correlate the hyper-alerts without reducing the complexity of any hyper-alert correlation graphs. The statistics of the initial analysis are shown in Table 5.3.

Table 5.3 shows that only 15% alerts generated by RealSecure are correlated. In addition, 20 out of 115 hyper-alert types that appear in this data set do not have any instances correlated. Among the remaining 95 hyper-alert types, 51 types have both correlated and uncorrelated instances.

Table 5.4 shows the statistics of the top 10 uncorrelated hyper-alert types (in terms of the number of uncorrelated hyper-alerts). Among these hyper-alert types, uncorrelated *IPHalfScan* counted 61% of all uncorrelated hyper-alerts. *Windows\_Access\_Error* counted 21% of all uncorrelated alerts. According to the description provided by RealSecure, a *Windows\_Access\_Error* represents an unsuccessful file sharing connection to a Windows or Samba server, which usually results from an attempt to brute-force a login under another account’s privileges. It is easy to see that the corresponding attacks could hardly prepare for any other attacks (since they failed). The third largest hyper-alert type *HTTP\_Cookie* counted for 3.3% of the total alerts. Though such alerts have certain privacy implications, we do not treat them as attacks, considering the nature of the DEF



**Figure 5.5:** A small hyper-alert correlation discovered in initial analysis

CON CTF events. These three hyper-alert types counted for 74.5% of all the alerts. We omit the discussion of the other uncorrelated hyper-alerts.

Figure 5.5 shows one of the small hyper-alert correlation graphs. The text in each node is the type followed by the ID of the hyper-alert. All the hyper-alerts in this figure were destined to the host at 010.020.001.024. All the *IPHalfScan* attacks were from source IP 010.020.011.240 at source port 55533 or 55534, and destined to port 110 at the victim host. After these attacks, all the attacks in the second stage except for 31886 were from 010.020.012.093 and targeted at port 110 of the victim host. The only two hyper-alerts that were not targeted at port 110 are hyper-alert 30882, which was destined to port 80 of the victim host, and hyper-alert 31886, which was destined to port 53. Thus, it is very possible that all the hyper-alerts except for 30882 and 31886 were related.

Not all of the hyper-alert correlation graphs are as small and comprehensible as Figure 5.5. In particular, the largest graph (in terms of the number of nodes) has 2,940 nodes and 25,321 edges, and on average, each graph has 21.75 nodes and 310.56 edges. Obviously, most of the hyper-alert correlation graphs are too big to understand for a human user.

### 5.3.2 Adjustable Graph Reduction

The largest hyper-alert correlation graph is taken as an example to analyze the results.

We first applied graph reduction utility to the hyper-alert correlation graphs. Figure 5.6 shows the fully reduced graph. Compared with the original graph, which has 2,940 nodes and 25,321 edges, the fully reduced graph has 77 nodes and 347 edges (including transitive edges).

The fully reduced graph in Figure 5.6 shows 7 stages of attacks. The layout of this graph was generated by GraphViz [1], which tries to reduce the number of cross edges and make the graph more balanced. As a result, the graph does not reflect the actual stages of attacks. Nevertheless, Figure 5.6 provides a much clearer outline of the attacks.

The hyper-alerts in stage 1 and about half of those in stage 2 correspond to scanning attacks

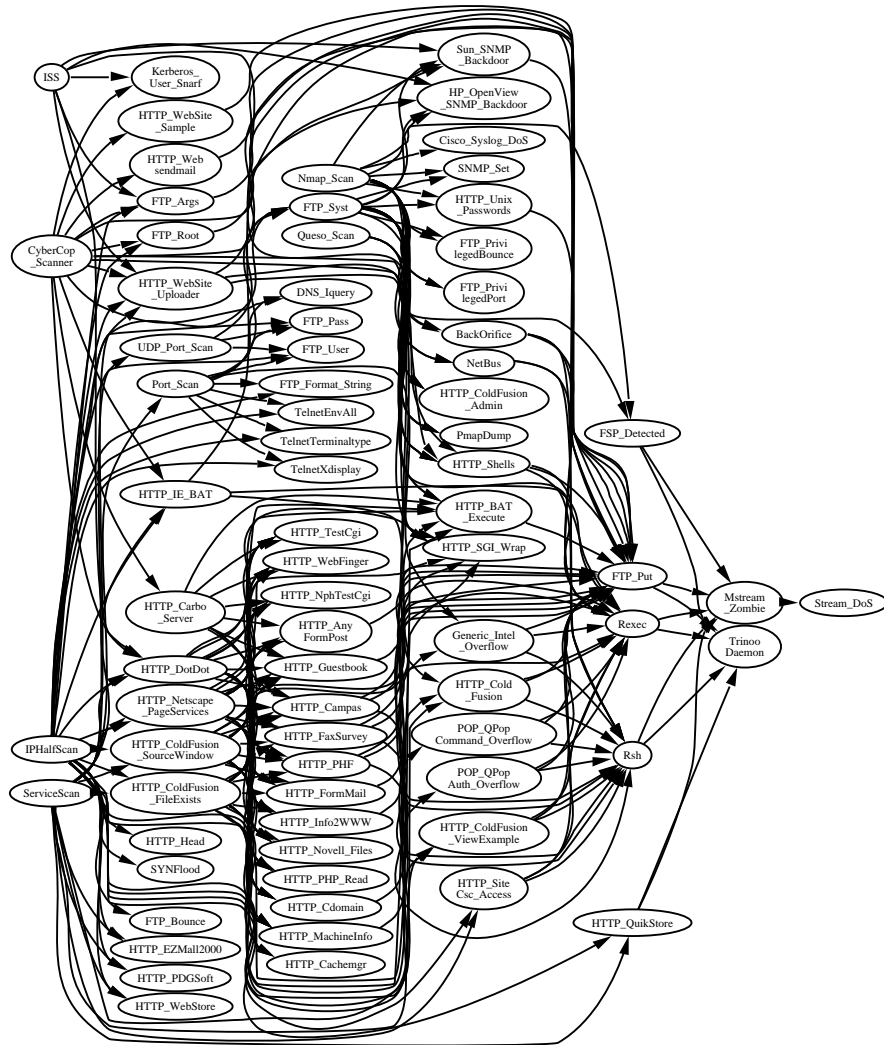
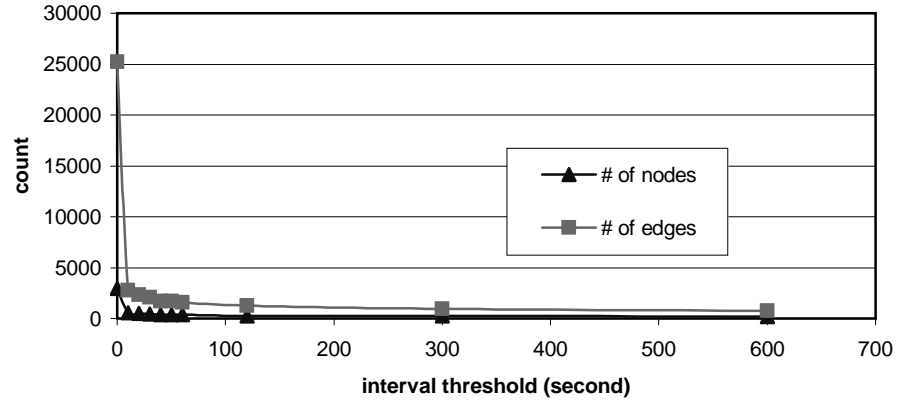


Figure 5.6: The fully reduced graph for the largest aggregated hyper-alert correlation graph.



**Figure 5.7:** Sizes of the reduced graphs w.r.t. the interval threshold for the largest hyper-alert correlation graph

or attacks to gain information of the target systems (e.g., *ISS*, *Port\_Scan*). The upper part of stage 2 include attacks that may lead to execution of arbitrary code on a target system (e.g., *HTTP\_WebSite\_Sample*). Indeed, these hyper-alerts directly prepare for some hyper-alerts in stage 5, but GraphViz arranged them in stage 2, possibly to balance the graph. Stages 3 consists of a mix of scanning attacks (e.g., *Nmap\_Scan*), attacks that reveal system information (e.g., *HTTP\_PHP\_Read*), and attacks that may lead to execution of arbitrary code (e.g., *HTTP\_Campas*). Stage 4 mainly consists of buffer overflow attacks (e.g., *POP\_QPopCommand\_Overflow*), detection of backdoor programs (e.g., *BackOrifice*), and attacks that may lead to execution of arbitrary code. The next 3 stages are much cleaner. Stage 5 consists of attacks that may be used to copy programs to target hosts, stage 6 consists of detection of two types of DDOS (Distributed Denial of Service) daemon programs, and finally, stage 7 consists of the detection of an actual DDOS attack.

Note that the fully reduce graph in Figure 5.6 is an approximation to the strategies used by the attackers. Hyper-alerts for different, independent sequences of attacks may be aggregated together in such a graph. For example, if two individual attackers use the sequence of attacks (e.g., using the same script downloaded from a website) to attack the same target, the corresponding hyper-alerts may be correlated and aggregated in the same fully reduced graph. Nevertheless, a fully reduced graph can clearly outline the attack strategies, and help a user understand the overall situation of attacks.

As we discussed earlier, the reduction of hyper-alert correlation graphs can be controlled with interval constraints. Figure 5.7 shows the numbers of nodes and edges of the reduced graphs for different interval sizes. The shapes of the two curves in Figure 5.7 indicate that most of the hyper-alerts that are of the same type occurred close to each other in time. Thus, the numbers of nodes and edges have a deep drop for small interval thresholds and a flat tail for large ones. A reasonable

guess is that some attackers tried the same type of attacks several times before they succeeded or gave up. Due to space reasons, we do not show these reduced graphs.

### 5.3.3 Focused Analysis

Focused analysis can help filter out the interesting parts of a large hyper-alert correlation graph. It is particularly useful when a user knows the systems being protected or the potential on-going attacks. For example, a user may perform a focused analysis with focusing constraint  $DestIP = ServerIP$ , where  $ServerIP$  is the IP address of a critical server, to find out attacks targeted at the server. As another example, he/she may use  $SrcIP = ServerIP \vee DestIP = ServerIP$  to find out attacks targeted at or originated from the server, suspecting that the server may have been compromised.

In our experiments, we tried a number of focusing constraints after we learned some information about the systems involved in the CTF event. Among these focusing constraints are (1)  $C_{f1} : (DestIP = 010.020.001.010)$  and (2)  $C_{f2} : (SrcIP = 010.020.011.251 \wedge DestIP = 010.020.001.010)$ . We applied both focusing constraints to the largest hyper-alert correlation graph. The results consist of 2154 nodes and 19423 edges for  $C_{f1}$ , and 51 nodes and 28 edges for  $C_{f2}$ . The corresponding fully reduced graphs are shown in Fig. 5.8 and Fig. 5.9, respectively. (Isolated nodes are shown in gray.) These two graphs also appear in the results of graph decomposition (Section 5.3.4). We defer the discussion of these two graphs to the next subsection.

Focused analysis is an attempt to approximate a sequence of attacks that satisfy the focusing constraint. Its success depends on the closeness of focusing constraints to the invariants of the sequences of attacks. A cunning attacker would try to avoid being correlated by launching attacks from different sources (or stepping stones) and introducing delays in between attacks. Thus, this utility should be used with caution.

### 5.3.4 Graph Decomposition

We applied three clustering constraints to decompose the largest hyper-alert correlation graph discovered in Section 5.3.1. In all these clustering constraints, we let  $A_1 = A_2 = \{SrcIP, DestIP\}$ .

1.  $C_{c1}(A_1, A_2): A_1.DestIP = A_2.DestIP$ . This is to cluster all hyper-alerts that share the same destination IP addresses. Since most of attacks are targeted at the hosts at the destination IP addresses, this is to cluster hyper-alerts in terms of the victim systems.
2.  $C_{c2}(A_1, A_2): A_1.SrcIP = A_2.SrcIP \wedge A_1.DestIP = A_2.DestIP$ . This is to cluster all the hyper-alerts that share the same source and destination IP addresses.

**Table 5.5:** number of clusters and graphs after decomposing the largest hyper-alert correlation graph.

	# clusters	# graphs
$C_{c1}$	12	10
$C_{c2}$	185	27
$C_{c3}$	2	1

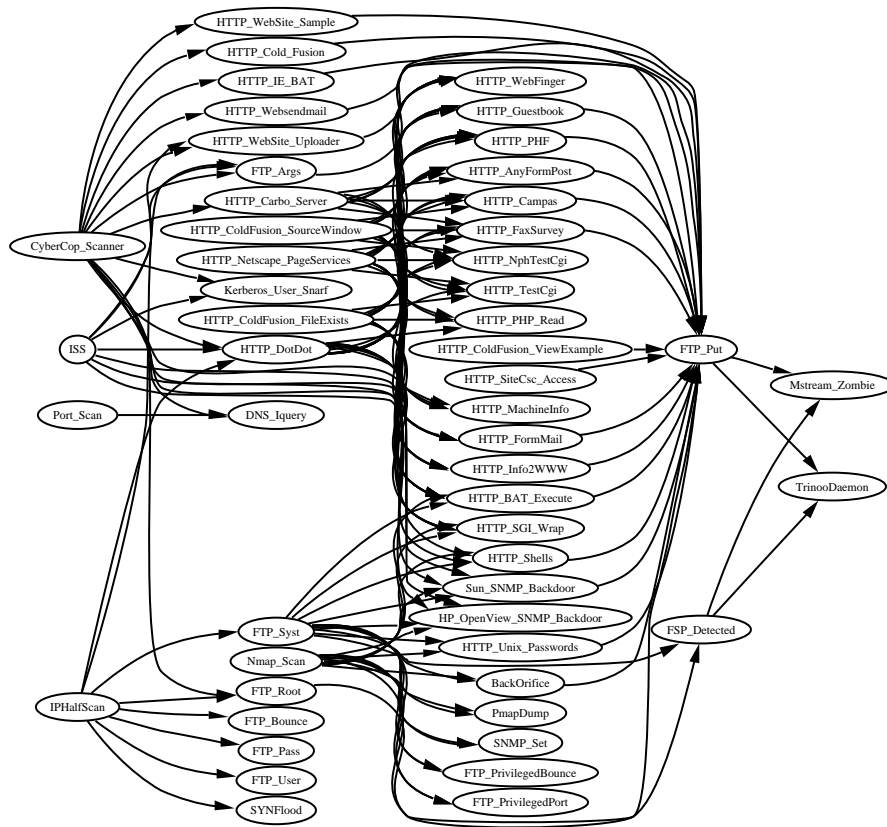
**Table 5.6:** Detailed information of each cluster after decomposing the largest hyper-alert correlation graph.

	cluster ID	1	2	3	4	5	6	7	8	9	10	11	12
$C_{c1}$	# connected nodes	2154	244	105	227	83	11	54	28	0	23	6	0
	# edges	19423	1966	388	2741	412	30	251	51	0	26	5	0
	# isolated nodes	0	0	0	0	0	0	0	0	0	1	0	4
$C_{c2}$	# correlated nodes	1970	17	0	12	0	0	0	3	0	29	0	0
	# edges	2240	66	0	10	0	0	0	2	0	28	0	0
	# isolated nodes	3	0	21	17	35	26	15	12	4	22	13	26
$C_{c3}$	# connected nodes	2935	0	-	-	-	-	-	-	-	-	-	-
	# edges	25293	0	-	-	-	-	-	-	-	-	-	-
	# isolated nodes	4	1	-	-	-	-	-	-	-	-	-	-

3.  $C_{c3}(A_1, A_2)$ :  $A_1.SrcIP = A_2.SrcIP \vee A_1.DestIP = A_2.DestIP \vee A_1.SrcIP = A_2.DestIP \vee A_1.DestIP = A_2.SrcIP$ . This is to cluster all the hyper-alerts that are connected via common IP addresses. Note that with this constraint, hyper-alerts in the same cluster may not share the same IP address directly, but they may connect to each other via other hyper-alerts.

Table 5.5 and table 5.6 show the statistics of the decomposed graphs.  $C_{c1}$  resulted in 12 clusters, among which 10 clusters contain edges.  $C_{c2}$  resulted in 185 clusters, among which 37 contain edges. Due to space reasons, we only show the first 12 clusters for  $C_{c2}$ .  $C_{c3}$  in effect removes one hyper-alert from the original graph. This hyper-alert is *Stream\_DoS*, which does not share the same IP address with any other hyper-alerts. This is because the source IPs of *Stream\_DoS* are spoofed and the destination IP is the target of this attack. This result shows that all the hyper-alerts except for *Stream\_DoS* share a common IP address with some others.

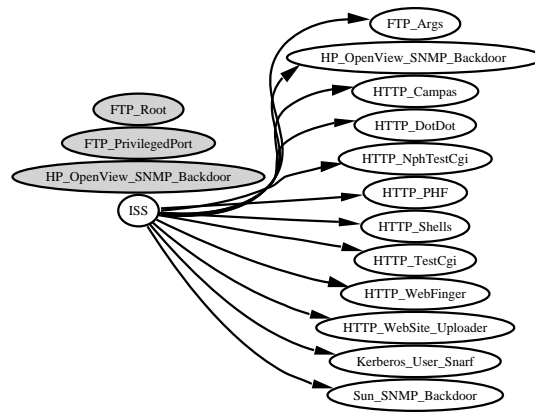
The isolated nodes in the resulting graphs are the hyper-alerts that prepare for or are prepared for by those that do not satisfy the same clustering constraints. Note that having isolated hyper-alerts in a decomposed graph does not imply that the isolated hyper-alerts are correlated incorrectly. For example, an attacker may hack into a host with a buffer overflow attack, install a DDOS daemon, and start the daemon program, which then tries to contact its master program. The corresponding alerts (i.e., the detection of the buffer overflow attack and the daemon’s message) will certainly not have the same destination IP address, though they are related.



**Figure 5.8:** A fully reduced hyper-alert correlation graph resulting from graph decomposition with  $C_{c1}$ . (Cluster ID = 1; DestIP = 010.020.001.010.)

Figures 5.8 and 5.9 show a decomposed graph for  $C_{c1}$  and  $C_{c2}$ , respectively. Both graphs are fully reduced to save space. All the hyper-alerts in Figure 5.8 are destined to 010.020.001.010. Figure 5.8 shows several possible attack strategies. The most obvious ones are those that lead to the *Mstream\_Zoombie* and *TrinooDaemon*. However, there are multiple paths that lead to these two hyper-alerts. Considering the fact that multiple attackers participated in the DEF CON 8 CTF event, we cannot conclude which path caused the installation of these daemon programs. Indeed, it is possible that none of them is the actual way, since the IDS may have missed some attacks.

Figure 5.8 involves 75 source IP addresses, including IP address 216.136.173.152, which does not belong to the CTF subnet. We believe that these attacks belong to different sequences of attacks, since there were intensive attacks from multiple attackers who participated in the CTF event.



**Figure 5.9:** A fully reduced hyper-alert correlation graph resulting from graph decomposition with  $C_{c2}$ . (Cluster ID = 10; SrcIP = 010.020.011.251; DestIP = 010.020.001.010.)

Figure 5.9 is related to Figure 5.8, since they both are about destination IP address 010.020.001.010. Indeed, Figure 5.9 is a part of Figure 5.8, though in Figure 5.8, *ISS* prepares for *HTTP\_Campas* through *HTTP\_DotDot*. Since all the hyper-alerts in Figure 5.9 have the same source and destination IP addresses, it is very possible that the correlated ones belong to the same sequence of attacks. Note that *HP\_OpenView\_SNMP\_Backdoor* appears as both connected and isolated node. This is because some instances are correlated, while the others are isolated.

We analyzed the correlated hyper-alerts using the three utilities and discovered several strategies used by the attackers. We first restricted us to the hyper-alert correlation graphs that satisfies the clustering constraint  $C_{c2}$ . One common strategy reflected by these graphs is to use scanning attacks followed by attacks that may lead to execution of arbitrary code. For example, the attacker(s) at 010.020.011.099 scanned host 010.020.001.010 with *CyberCop\_Scanner*, *IPHalfScan*, *Nmap\_Scan*, and *Port\_Scan* and then launched a sequence of HTTP-based attacks (e.g., *HTTP\_DotDot*) and FTP based attacks (e.g., *FTP\_Root*). The attacker(s) at 010.020.011.093 and 010.020.011.227 also used a similar sequence of attacks against the host 010.020.001.008.

As another strategy, the attacker(s) at 010.020.011.240 used a concise sequence of attacks against the host at 010.020.001.013: *Nmap\_Scan* followed by *PmapDump* and then *ToolTalk\_Overflow*. Obviously, they used *Nmap\_Scan* to find the *portmap* service, then used *PmapDump* to list the RPC services, and finally launched a *ToolTalk\_Overflow* attack against the *ToolTalk* service. Indeed, the sequence *Nmap\_Scan* followed by *PmapDump* with the same source and destination IP address appeared many times in this dataset.

The attacker(s) at 010.020.011.074 used the same sequence of HTTP-based attacks (e.g., *HTTP\_DotDot* and *HTTP\_TestCgi*) against multiple web servers (e.g., servers at 010.020.001.014, 010.020.001.015, 010.020.001.019, etc.). Our hyper-alert correlation graphs shows that *HTTP\_DotDot* prepares for the following HTTP-based attacks. However, our further analysis of the dataset shows that this may be an incorrect correlation. Though it is possible that the attacker used *HTTP\_DotDot* to collect necessary information for the later attacks, the timestamps of these alerts indicate that the attacker(s) used a script to launch all these attacks. Thus, it is possible that the attacker(s) simply launch all the attacks, hoping one of them would succeed. Though these alerts are indeed related, these prepare-for relations reveal that our method is aggressive in correlating alerts. Indeed, alert correlation is to recover the relationships between the attacks behind alerts; any alert correlation method may make mistakes when there is not enough information.

One interesting observation is that with clustering constraint  $C_{c2}$ , there is not many hyper-alert correlation graphs with more than 3 stages. Considering the fact that there are many alerts about *BackOrifice* and *NetBus* (which are tools to remotely manage hosts), we suspect that many attackers used multiple machines during their attacks. Thus, their strategies cannot be reflected by the restricted hyper-alert correlation graphs.

When relax the restriction to allow hyper-alert correlation graphs involving different source IP but still with the same destination IP addresses (i.e., with clustering constraint  $C_{c1}$ ), we have graphs with more stages. Figure 5.8 is one such fully reduced hyper-alert correlation graph. However, due to the amount of alerts and source IP addresses involved in this graph, it is difficult to conclude which hyper-alerts belong to the same sequences of attacks.

## Chapter 6

# Conclusions and Future Work

### 6.1 Conclusions

In this thesis, I implemented a toolkit based on the framework discussed in Chapter 3. The basic tool consists of a knowledge base, an alert preprocessor, an alert correlation engine and a graph output component. In order to help analyze intensive alerts, I also implemented three utilities (adjustable graph reduction, focused analysis and graph decomposition) on the basis of the basic tool.

I have performed a sequence of experiments to evaluate the tools developed in my thesis. The experimental results showed that the developed tools were very helpful in analyzing intrusion alerts. First, they can be used to construct high-level attack scenarios from the low-level alerts reported by IDS. In the experiments with DARPA 2000 intrusion detection scenario specific datasets, we have successfully constructed attack scenarios from the alerts, and the constructed scenarios matched the description of the data set very well. Second, they can be potentially used to differentiate true alerts from false alerts with a high accuracy. In the experiments, we reduced the false alert rate from 93.6% to 5.26% in the best case, and from 98.59% to 40% even in the worst case. At the same time, the soundness measures of our technique, that is, the correctness of the alerts correlated are above 90% for all experiments. The experiments with the DEFCON CTF 8 dataset also showed the three utilities could be used to reduce the complexity of the correlated alerts while keeping the structure of the attacks.

However, the experiments also showed that we still need to improve our techniques. The alerts correlated are not 100% right. Some of the true alerts are missed and some false alerts are correlated, although it is only a small portion compared with the truly correlated alerts.

## 6.2 Future Work

### 6.2.1 Real-Time Correlator

In this thesis work, I developed an offline toolkit to correlate intrusion alerts. Can we use this technique to correlate the alerts in real-time? The most important factor for real-time processing is the efficiency. In the offline toolkit implementation, DBMS is used to process the preprocessed data. The database operations do make it easier to develop this tool, but it is not an appropriate choice for real-time processing because of its performance penalty. For example, in the experiments of DEFCON, it takes me 45 minutes to preprocess 65,000 alerts if JDBC-ODBC bridge is used, and it drops to 5 minutes if JDBC bridge is used though. It is still unbearable for real-time processing.

One option to improve the efficiency is to adapt some existing in-memory query optimization techniques, such as Array Binary Search, AVL Trees, B Trees, Chained Bucket Hashing, Linear Hashing, and T Trees. Some initial results about this study can be found in [24].

### 6.2.2 Input Alerts

In the current implementation, we assume that we can get all the information about the original alerts from a single table in database. However, it is not true for some IDSs (e.g., Snort [3]).

An alternative way is provided in the property file to solve this problem partially. If the user can use one SQL statement to write all the needed information into a table in database, then he can specify this SQL statement in the property file, and we will process the SQL statement before any other operations.

But, it is not so easy sometimes. Also, it is not very user friendly. It would be much better if we have a specific component to deal with this problem. It gets all the information we need from different sources and writes them into a single table in database.

### 6.2.3 Identifying Missing Detections & Predicating Alerts

In our experiment of the DARPA dataset, there is a small decrease in the detection rate compared with RealSecure. Can we reduce the false alert rate without hurting the detection rate? It would be even better if we can not only reduce the false alert rate, but also improve the detection rate of the underlying IDS.

In other words, our correlator can help the underlying IDS figure out what alerts are probably missed by it, and what alerts are going to happen next based on the attack strategy we construct and previously seen attack scenarios.

#### 6.2.4 Correlating Alerts from Different IDSs

In our experiment, we only used the alerts generated by one IDS. As we mentioned in the introduction, different types of IDSs have their own strength and weaknesses. It would be more effective in detecting attacks if we could utilize the strength of each of them. But this leads to another problem, that is, how to correlate the alerts generated from different IDSs.

Although our framework supports it in theory, can our tool support it? How to synchronize the timestamps from different IDSs? How to map the original alerts generated by different IDSs to hyper-alerts? How to recognize the alerts generated by different IDSs, which are actually triggered by the same event? These are all the research problems that need to be addressed.

## List of References

- [1] <http://www.cert.org>
- [2] <http://www.shmoo.org/cctf>
- [3] <http://www.snort.org>
- [4] <http://xml.apache.org>
- [5] Anderson, J. P. Computer security threat monitoring and surveillance. Technical report, James P. Anderson Co., Fort Washington, PA, 1980.
- [6] AT & T Research Labs. GraphViz - open source graph layout and drawing software. <http://www.research.att.com/sw/tools/graphviz/>.
- [7] Bace, R. *Intrusion Detection*. Macmillan Technology Publishing, 2000.
- [8] Cuppens, F. and Mieke, A. Alert correlation in a cooperative intrusion detection framework. In *Proc. of the 2002 IEEE Symposium on Security and Privacy*, May 2002.
- [9] Cuppens, F. and Ortalo, R. 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.
- [10] Dain, O. and Cunningham, R. Fusing a heterogeneous alert stream into scenarios. In *Proc. of the 2001 ACM Workshop on Data Mining for Security Applications*, pages 1–13, Nov. 2001.
- [11] Debar, H. and Wespi, A. Aggregation and correlation of intrusion-detection alerts. In *Recent Advances in Intrusion Detection*, LNCS 2212, pages 85 – 103, 2001.
- [12] Gardner, R. and Harle, D. Pattern discovery and specification translation for alarm correlation. In *Proc. of Network Operations and Management Symposium (NOMS'98)*, pages 713–722, 1998.

- [13] Gruschke, B. Integrated event management: Event correlation using dependency graphs. In *Proc. of the 9th IFIP/IEEE International Workshop on Distributed Systems: Operations & Management*, 1998.
- [14] Ilgun, K., Kemmerer, R.A. and Porras, P.A. State transition analysis: A rule-based intrusion detection approach. In *IEEE Transaction on Software Engineering* 21, pages 181–199, 1995.
- [15] ISS, Inc. RealSecure intrusion detection system. <http://www.iss.net>.
- [16] Bace, R. and Mell, P. NIST Special Publication on Intrusion Detection System. NIST(National Institute of Standards and Technology) Special Publication 800-31, August 2001.
- [17] Jha, S., Sheyner, O. and Wing, J. Two formal analyses of attack graphs. In *Proc. of the 15th Computer Security Foundation Workshop*, June 2002.
- [18] Julisch, K. Mining Alarm Clusters to Improve Alarm Handling Efficiency. In *Proc. of the 17th Annual Computer Security Application Conference*, December 2001.
- [19] MIT Lincoln Lab. 2000 DARPA intrusion detection scenario specific datasets. [http://www.ll.mit.edu/IST/ideval/data/2000/2000\\_data\\_index.html](http://www.ll.mit.edu/IST/ideval/data/2000/2000_data_index.html), 2000.
- [20] Lin, J., Wang, X.S. and Jajodia, S. Abstraction-based misuse detection: High-level specifications and adaptable strategies. In *Proc. of the 11th Computer Security Foundations Workshop*, pages 190–201, 1998.
- [21] Morin, Benjamin, Me, Ludovic, Debar, Herve and Ducasse, Mireille M2D2: A Formal Data Model for IDS Alert Correlation In *Proc. of the 5th Int'l Symposium on Recent Advances in Intrusion Detection (RAID 2002)*, October 2002.
- [22] Ning, P., Cui, Y. and Reeves, D. S. Analyzing intensive intrusion alerts via correlation. In *Proc. of the 5th Int'l Symposium on Recent Advances in Intrusion Detection (RAID 2002)*, October 2002.
- [23] Ning, P., Cui, Y. and Reeves, D. S. Constructing attack scenarios through correlation of intrusion alerts. To appear in the 9th ACM Conference on Computer & Communications Security, November 2002.
- [24] Ning, P. and Xu, D. Adapting query optimization techniques for efficient intrusion alert correlation. Technical Report TR-2002-14, North Carolina State University, Department of Computer Science, September 2002.

- [25] Ning, P., Jajodia, S. and Wang, X.S. Abstraction-based intrusion detection in distributed environments. In *ACM Transactions on Information and System Security* 4, pages 407–452, 2001.
- [26] Porras, A. Phillip, Fong, W. Martin and Valdes, Alfonso A Mission-Impact-Based Approach to INFOSEC Alarm Correlation In *Proc. of the 5th Int'l Symposium on Recent Advances in Intrusion Detection (RAID 2002)*, October 2002.
- [27] Ricciulli, L. and Shacham, N. Modeling correlated alarms in network management system. In *Western Simulation Multiconference*, 1997.
- [28] Ritchey, R. and Ammann, P. Using model checking to analyze network vulnerabilities. In *Proc. of IEEE Symposium on Security and Privacy*, pages 156–165, May 2000.
- [29] Sheyner, O., Haines, J., Jha, S., Lippmann, R. and Wing, J. Automated generation and analysis of attack graphs. In *Proc. of IEEE Symposium on Security and Privacy*, May 2002.
- [30] Staniford, S., Hoagland, J. and McAlerney, J. Practical automated detection of stealthy portscans. To appear in *Journal of Computer Security*, 2002.
- [31] Staniford-Chen, S., Cheung, S., Crawford, R., Dilger, M., Frank, J., Hoagland, J., Levitt, K., Wee, C., Yip, R. and Zerkle, D. GrIDS - a graph based intrusion detection system for large network. In *Proc. of the 19th National Information System Security Conference*, Volume 1, pages 361–370, 1996.
- [32] Templeton, S. and Levit, K. A requires/provides model for computer attacks. In *Proc. of New Security Paradigms Workshop*, pages 31 – 38. September 2000.
- [33] Valdes, A. and Skinner, K. Probabilistic alert correlation. In *Proc. of the 4th Int'l Symposium on Recent Advances in Intrusion Detection (RAID 2001)*, pages 54–68, 2001.
- [34] Vigna, G. and Kemmerer, R.A. NetSTAT: A network-based intrusion detection system. In *Journal of Computer Security* 7, pages 37–71, 1999.

## Appendix A

# Knowledge Base Used In The Experiment

## A.1 XML Schema for Knowledge Base

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns="ID@NCSSU"
            targetNamespace="ID@NCSSU"
            elementFormDefault="qualified">

  <!-- Big Picture of KnowledgeBase -->

  <xsd:element name="KnowledgeBase">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="Predicates"/>
        <xsd:element ref="Implications"/>
        <xsd:element ref="HyperAlertTypes"/>
      </xsd:sequence>
    </xsd:complexType>

    <xsd:unique name="unique1">
      <xsd:selector xpath="HyperAlertTypes/HyperAlertType"/>
      <xsd:field xpath="@Name"/>
    </xsd:unique>
  </xsd:element>
</xsd:schema>
```

```

<!-- set the name of the predicate as the key -->

<xsd:key name="PredicateNameKey">
  <xsd:selector xpath="Predicates/Predicate"/>
  <xsd:field xpath="@Name"/>
</xsd:key>

<xsd:keyref name="keyref1" refer="PredicateNameKey">
  <xsd:selector xpath="Implications/Implication/Implying"/>
  <xsd:field xpath="ImplyingName"/>
</xsd:keyref>

<xsd:keyref name="keyref2" refer="PredicateNameKey">
  <xsd:selector xpath="Implications/Implication/Implied"/>
  <xsd:field xpath="ImpliedName"/>
</xsd:keyref>

<xsd:keyref name="keyref3" refer="PredicateNameKey">
  <xsd:selector xpath="HyperAlertTypes/HyperAlertType/Prerequisite/Predicate"/>
  <xsd:field xpath="@Name"/>
</xsd:keyref>

<xsd:keyref name="keyref4" refer="PredicateNameKey">
  <xsd:selector xpath="HyperAlertTypes/HyperAlertType/Consequence/Predicate"/>
  <xsd:field xpath="@Name"/>
</xsd:keyref>

<!-- set the argument ID as the key -->

<xsd:unique name="ArgumentKey">
  <xsd:selector xpath="Predicates/Predicate/Arg"/>
  <xsd:field xpath="@id"/>
</xsd:unique>

<xsd:keyref name="keyref5" refer="ArgumentKey">

```

```

    <xsd:selector xpath="Implications/Implication/ArgMap/ImpliedArg"/>
    <xsd:field xpath="@id"/>
  </xsd:keyref>

  <xsd:keyref name="keyref6" refer="ArgumentKey">
    <xsd:selector xpath="Implications/Implication/ArgMap/ImpliedArg"/>
    <xsd:field xpath="@id"/>
  </xsd:keyref>

  <xsd:keyref name="keyref7" refer="ArgumentKey">
    <xsd:selector xpath="HyperAlertTypes/HyperAlertType/Prerequisite/Predicate/Arg"/>
    <xsd:field xpath="@id"/>
  </xsd:keyref>

  <xsd:keyref name="keyref8" refer="ArgumentKey">
    <xsd:selector xpath="HyperAlertTypes/HyperAlertType/Consequence/Predicate/Arg"/>
    <xsd:field xpath="@id"/>
  </xsd:keyref>

</xsd:element>

<!-- ===== -->
<!-- Predicates schema -->

<xsd:element name="Predicates">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="Predicate" minOccurs="0" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="Arg" minOccurs="0" maxOccurs="unbounded">
              <xsd:complexType>
                <xsd:attribute name="id" type="xsd:positiveInteger">
              </xsd:attribute>
            <xsd:attribute name="Pos" type="xsd:positiveInteger">

```

```

        </xsd:attribute>
        <xsd:attribute name="Attr">
            <xsd:simpleType>
                <xsd:restriction base="xsd:string">
                    <xsd:enumeration value="int"/>
                    <xsd:enumeration value="varchar(15)"/>
                </xsd:restriction>
            </xsd:simpleType>
        </xsd:attribute>
    </xsd:complexType>
</xsd:element>
<!-- end of element Arg -->
</xsd:sequence>
    <xsd:attribute name="Name" type="xsd:string" use="required">
</xsd:attribute>
</xsd:complexType>
</xsd:element>
</xsd:sequence>
</xsd:complexType>
</xsd:element>

<!-- ===== -->
<!-- Implication schema -->
<xsd:element name="Implications">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="Implication" minOccurs='0' maxOccurs='unbounded' />
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>

<xsd:element name="Implication">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="ImplyingName" type="xsd:string"/>

```

```

<xsd:element name="ImpliedName" type="xsd:string"/>
<xsd:element name="ArgMap" minOccurs='0' maxOccurs='unbounded'>
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="ImplyingArg">
        <xsd:complexType>
          <xsd:attribute name="id" type="xsd:positiveInteger">
          </xsd:attribute>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="ImpliedArg">
        <xsd:complexType>
          <xsd:attribute name="id" type="xsd:positiveInteger">
          </xsd:attribute>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:attribute name="Phantom" default="No">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="Yes"/>
      <xsd:enumeration value="No"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:attribute>

</xsd:complexType>
</xsd:element>

<!-- ===== -->
<!-- HyperAlertTypes schema -->

```

```

<xsd:element name="HyperAlertTypes">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="HyperAlertType" minOccurs='1' maxOccurs='unbounded' />
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="HyperAlertType">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="Fact" minOccurs='1' maxOccurs='unbounded' />
      <xsd:element ref="Prerequisite" minOccurs='0' maxOccurs='unbounded' />
      <xsd:element ref="Consequence" minOccurs='0' maxOccurs='unbounded' />
    </xsd:sequence>
    <xsd:attribute name="Name" type="xsd:string" use="required">
    </xsd:attribute>
  </xsd:complexType>

<!-- set the Fact as the key for each hyper alert type -->
<xsd:key name="FactKey">
  <xsd:selector xpath="Fact" />
  <xsd:field xpath="@FactName" />
</xsd:key>

<xsd:keyref name="keyref9" refer="FactKey">
  <xsd:selector xpath="Prerequisite/Predicate/Arg" />
  <xsd:field xpath="@ArgName" />
</xsd:keyref>

<xsd:keyref name="keyref10" refer="FactKey">
  <xsd:selector xpath="Consequence/Predicate/Arg" />
  <xsd:field xpath="@ArgName" />
</xsd:keyref>

```

```
</xsd:element>
```

```
<xsd:element name="Fact">  
  <xsd:complexType>  
    <xsd:attribute name="FactName" type="xsd:string">  
  </xsd:attribute>  
    <xsd:attribute name="FactType">  
      <xsd:simpleType>  
        <xsd:restriction base="xsd:string">  
          <xsd:enumeration value="int"/>  
          <xsd:enumeration value="varchar(15)"/>  
        </xsd:restriction>  
      </xsd:simpleType>  
    </xsd:attribute>  
  </xsd:complexType>  
</xsd:element>
```

```
<xsd:element name="Prerequisite">  
  <xsd:complexType>  
    <xsd:sequence>  
      <xsd:element ref="Predicate" minOccurs='0' maxOccurs='unbounded' />  
    </xsd:sequence>  
  </xsd:complexType>  
</xsd:element>
```

```
<xsd:element name="Consequence">  
  <xsd:complexType>  
    <xsd:sequence>  
      <xsd:element ref="Predicate" minOccurs='0' maxOccurs='unbounded' />  
    </xsd:sequence>  
  </xsd:complexType>  
</xsd:element>
```

```
<xsd:element name="Predicate">
```

```

<xsd:complexType>
  <xsd:sequence>
    <xsd:element name="Arg" minOccurs='0' maxOccurs='unbounded'>
      <xsd:complexType>
        <xsd:attribute name="id" type="xsd:positiveInteger" use="required">
        </xsd:attribute>
        <xsd:attribute name="ArgName" type="xsd:string" use="required">
        </xsd:attribute>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
  <xsd:attribute name="Name" type="xsd:string" use='required'>
  </xsd:attribute>
</xsd:complexType>
</xsd:element>

</xsd:schema>

```

## A.2 Knowledge Base Used for DARPA Dataset

```

<?xml version="1.0" encoding="UTF-8"?>
<KnowledgeBase xmlns="ID@NCSU"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="ID@NCSU
  newKnowledgeBase.xsd">

<!-- based on .5 -->
<!-- add implication between ehlo and turn -->

<!-- ===== Predicates ===== -->
<Predicates>
<Predicate Name="CiscoCatalyst3500XL">
  <Arg id="1" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
<Predicate Name="ActiveXEnabledBrowser">

```

```

    <Arg id="2" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
<Predicate Name="ExistFTPService">
    <Arg id="3" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
<Predicate Name="GainAccess">
    <Arg id="4" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
<Predicate Name="GainAdmindInfo">
    <Arg id="5" Pos="1" Attr="varchar(15)"></Arg>
    <Arg id="6" Pos="2" Attr="int"></Arg>
</Predicate>
<Predicate Name="GainInformation">
    <Arg id="7" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
<Predicate Name="GainOSInfo">
    <Arg id="8" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
<Predicate Name="GainServiceInfo">
    <Arg id="9" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
<!--
<Predicate Name="GainSMTPInfo">
    <Arg id="10" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
-->
<Predicate Name="GainTerminalType">
    <Arg id="11" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
<Predicate Name="JavaEnabledBrowser">
    <Arg id="12" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
<Predicate Name="MailLeakage">
    <Arg id="13" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>

```

```
<Predicate Name="ReadyToLaunchDDOSAttack"></Predicate>
<Predicate Name="SendMailInDebugMode">
  <Arg id="14" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
<Predicate Name="SMTPSupportEhlo">
  <Arg id="15" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
<!--
<Predicate Name="SMTPSupportTurn">
  <Arg id="16" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
-->
<Predicate Name="SystemAttacked">
  <Arg id="17" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
<Predicate Name="SystemCompromised">
  <Arg id="18" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
<Predicate Name="VulnerableCGIBin">
  <Arg id="19" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
<Predicate Name="VulnerableAlMailPOP3Server">
  <Arg id="20" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
<Predicate Name="VulnerableSadmin">
  <Arg id="21" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
<Predicate Name="ExistService">
  <Arg id="22" Pos="1" Attr="varchar(15)"></Arg>
  <Arg id="23" Pos="2" Attr="int"></Arg>
</Predicate>
<Predicate Name="DNS_HInfo">
  <Arg id="24" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
<Predicate Name="OSSolaris">
```

```

    <Arg id="25" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
<Predicate Name="OSUNIX">
    <Arg id="26" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
<Predicate Name="DDOSAgainst">
    <Arg id="27" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
<Predicate Name="GainSMTPInfo">
    <Arg id="28" Pos="1" Attr="varchar(15)"></Arg>
    <Arg id="29" Pos="2" Attr="varchar(15)"></Arg>
</Predicate>
<Predicate Name="SMTPSupportTurn">
    <Arg id="30" Pos="1" Attr="varchar(15)"></Arg>
    <Arg id="31" Pos="2" Attr="varchar(15)"></Arg>
</Predicate>
</Predicates>

<!-- ===== Implications ===== -->
<Implications>
<!--
<Implication>
    <ImpliedName>GainAdmindInfo</ImpliedName>
    <ImpliedName>GainInformation</ImpliedName>
    <ArgMap>
        <ImpliedArg id="5"></ImpliedArg>
        <ImpliedArg id="7"></ImpliedArg>
    </ArgMap>
</Implication>
<Implication>
    <ImpliedName>GainOSInfo</ImpliedName>
    <ImpliedName>GainInformation</ImpliedName>
    <ArgMap>
        <ImpliedArg id="8"></ImpliedArg>
        <ImpliedArg id="7"></ImpliedArg>
    </ArgMap>
</Implication>

```

```

    </ArgMap>
</Implication>
<Implication>
  <ImpliedName>GainServiceInfo</ImpliedName>
  <ImpliedName>GainInformation</ImpliedName>
  <ArgMap>
    <ImpliedArg id="9"></ImpliedArg>
    <ImpliedArg id="7"></ImpliedArg>
  </ArgMap>
</Implication>
<Implication>
  <ImpliedName>GainSMTPInfo</ImpliedName>
  <ImpliedName>GainInformation</ImpliedName>
  <ArgMap>
    <ImpliedArg id="10"></ImpliedArg>
    <ImpliedArg id="7"></ImpliedArg>
  </ArgMap>
</Implication>
<Implication>
  <ImpliedName>GainTerminalType</ImpliedName>
  <ImpliedName>GainInformation</ImpliedName>
  <ArgMap>
    <ImpliedArg id="11"></ImpliedArg>
    <ImpliedArg id="7"></ImpliedArg>
  </ArgMap>
</Implication>
-->
<Implication>
  <ImpliedName>GainAccess</ImpliedName>
  <ImpliedName>SystemCompromised</ImpliedName>
  <ArgMap>
    <ImpliedArg id="4"></ImpliedArg>
    <ImpliedArg id="18"></ImpliedArg>
  </ArgMap>
</Implication>

```

```

<Implication>
  <ImplyingName>SystemCompromised</ImplyingName>
  <ImpliedName>SystemAttacked</ImpliedName>
  <ArgMap>
    <ImplyingArg id="18"></ImplyingArg>
    <ImpliedArg id="17"></ImpliedArg>
  </ArgMap>
</Implication>
<Implication Phantom="Yes">
  <ImplyingName>GainOSInfo</ImplyingName>
  <ImpliedName>OSSolaris</ImpliedName>
  <ArgMap>
    <ImplyingArg id="8"></ImplyingArg>
    <ImpliedArg id="25"></ImpliedArg>
  </ArgMap>
</Implication>
<Implication Phantom="Yes">
  <ImplyingName>GainOSInfo</ImplyingName>
  <ImpliedName>OSUNIX</ImpliedName>
  <ArgMap>
    <ImplyingArg id="8"></ImplyingArg>
    <ImpliedArg id="26"></ImpliedArg>
  </ArgMap>
</Implication>
<Implication Phantom="Yes">
  <ImplyingName>GainSMTPInfo</ImplyingName>
  <ImpliedName>SMTPSupportTurn</ImpliedName>
  <ArgMap>
    <ImplyingArg id="28"></ImplyingArg>
    <ImpliedArg id="30"></ImpliedArg>
  </ArgMap>
  <ArgMap>
    <ImplyingArg id="29"></ImplyingArg>
    <ImpliedArg id="31"></ImpliedArg>
  </ArgMap>
</Implication>

```

```

</Implication>
</Implications>

<!-- ===== Hyper Alert Types ===== -->
<HyperAlertTypes>

<HyperAlertType Name="Admind">
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <!-- <Prerequisite>
    <Predicate Name="ExistService">
      <Arg id="22" ArgName="DestIPAddress"></Arg>
      <Arg id="23" ArgName="DestPort"></Arg>
    </Predicate>
    <Predicate Name="OSSolaris">
      <Arg id="25" ArgName="DestIPAddress"></Arg>
    </Predicate>
    <Predicate Name="VulnerableSadmind">
      <Arg id="21" ArgName="DestIPAddress"></Arg>
    </Predicate>
  </Prerequisite>
  <Consequence>
    <Predicate Name="GainAdmindInfo">
      <Arg id="5" ArgName="DestIPAddress"></Arg>
      <Arg id="6" ArgName="DestPort"></Arg>
    </Predicate>
  </Consequence>
-->
</HyperAlertType>

<HyperAlertType Name="DNS_HInfo">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>

```

```

<Fact FactName="DestPort" FactType="int"></Fact>
<Prerequisite>
  <Predicate Name="ExistService">
    <Arg id="22" ArgName="DestIPAddress"></Arg>
    <Arg id="23" ArgName="DestPort"></Arg>
  </Predicate>
</Prerequisite>
<Consequence>
  <Predicate Name="GainOSInfo">
    <Arg id="8" ArgName="DestIPAddress"></Arg>
  </Predicate>
</Consequence>
</HyperAlertType>

<HyperAlertType Name="Email_Almail_Overflow">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <Prerequisite>
    <Predicate Name="ExistService">
      <Arg id="22" ArgName="DestIPAddress"></Arg>
      <Arg id="23" ArgName="DestPort"></Arg>
    </Predicate>
    <Predicate Name="VulnerableAlMailPOP3Server">
      <Arg id="20" ArgName="DestIPAddress"></Arg>
    </Predicate>
  </Prerequisite>
  <Consequence>
    <Predicate Name="GainAccess">
      <Arg id="4" ArgName="DestIPAddress"></Arg>
    </Predicate>
  </Consequence>
</HyperAlertType>

```

```

<HyperAlertType Name="Email_Debug">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <Prerequisite>
    <Predicate Name="ExistService">
      <Arg id="22" ArgName="DestIPAddress"></Arg>
      <Arg id="23" ArgName="DestPort"></Arg>
    </Predicate>
    <Predicate Name="SendMailInDebugMode">
      <Arg id="14" ArgName="DestIPAddress"></Arg>
    </Predicate>
  </Prerequisite>
  <Consequence>
    <Predicate Name="GainAccess">
      <Arg id="4" ArgName="DestIPAddress"></Arg>
    </Predicate>
  </Consequence>
</HyperAlertType>

```

```

<HyperAlertType Name="Email_Ehlo">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <Prerequisite>
    <Predicate Name="ExistService">
      <Arg id="22" ArgName="DestIPAddress"></Arg>
      <Arg id="23" ArgName="DestPort"></Arg>
    </Predicate>
    <Predicate Name="SMTPSupportEhlo">
      <Arg id="15" ArgName="DestIPAddress"></Arg>
    </Predicate>
  </Prerequisite>

```

```

<Consequence>
  <Predicate Name="GainSMTPInfo">
    <Arg id="28" ArgName="SrcIPAddress"></Arg>
    <Arg id="29" ArgName="DestIPAddress"></Arg>
  </Predicate>
</Consequence>
</HyperAlertType>

<HyperAlertType Name="Email_Turn">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <Prerequisite>
    <Predicate Name="ExistService">
      <Arg id="22" ArgName="DestIPAddress"></Arg>
      <Arg id="23" ArgName="DestPort"></Arg>
    </Predicate>
    <Predicate Name="SMTPSupportTurn">
      <Arg id="30" ArgName="SrcIPAddress"></Arg>
      <Arg id="31" ArgName="DestIPAddress"></Arg>
    </Predicate>
  </Prerequisite>
  <Consequence>
    <Predicate Name="MailLeakage">
      <Arg id="13" ArgName="DestIPAddress"></Arg>
    </Predicate>
  </Consequence>
</HyperAlertType>

<HyperAlertType Name="FTP_Pass">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>

```

```

    <Prerequisite>
<!--    <Predicate Name="ExistFTPService">
        <Arg id="3" ArgName="DestIPAddress"></Arg>
    </Predicate> -->
    <Predicate Name="ExistService">
        <Arg id="22" ArgName="DestIPAddress"></Arg>
        <Arg id="23" ArgName="DestPort"></Arg>
    </Predicate>
</Prerequisite>
</HyperAlertType>

<HyperAlertType Name="FTP_Put">
    <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
    <Fact FactName="SrcPort" FactType="int"></Fact>
    <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
    <Fact FactName="DestPort" FactType="int"></Fact>
    <Prerequisite>
<!--    <Predicate Name="ExistFTPService">
        <Arg id="3" ArgName="DestIPAddress"></Arg>
    </Predicate> -->
    <Predicate Name="ExistService">
        <Arg id="22" ArgName="DestIPAddress"></Arg>
        <Arg id="23" ArgName="DestPort"></Arg>
    </Predicate>
    <Predicate Name="GainAccess">
        <Arg id="4" ArgName="DestIPAddress"></Arg>
    </Predicate>
</Prerequisite>
    <Consequence>
        <Predicate Name="SystemCompromised">
            <Arg id="18" ArgName="DestIPAddress"></Arg>
        </Predicate>
    </Consequence>
</HyperAlertType>

```

```

<HyperAlertType Name="FTP_Syst">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <Prerequisite>
<!--    <Predicate Name="ExistFTPService">
      <Arg id="3" ArgName="DestIPAddress"></Arg>
    </Predicate> -->
    <Predicate Name="ExistService">
      <Arg id="22" ArgName="DestIPAddress"></Arg>
      <Arg id="23" ArgName="DestPort"></Arg>
    </Predicate>
  </Prerequisite>
  <Consequence>
    <Predicate Name="GainOSInfo">
      <Arg id="8" ArgName="DestIPAddress"></Arg>
    </Predicate>
  </Consequence>
</HyperAlertType>

```

```

<HyperAlertType Name="FTP_User">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <Prerequisite>
<!--    <Predicate Name="ExistFTPService">
      <Arg id="3" ArgName="DestIPAddress"></Arg>
    </Predicate> -->
    <Predicate Name="ExistService">
      <Arg id="22" ArgName="DestIPAddress"></Arg>
      <Arg id="23" ArgName="DestPort"></Arg>
    </Predicate>
  </Prerequisite>

```

```

    </Prerequisite>
</HyperAlertType>

<HyperAlertType Name="HTTP_ActiveX">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <Prerequisite>
    <Predicate Name="ActiveXEnabledBrowser">
      <Arg id="2" ArgName="SrcIPAddress"></Arg>
    </Predicate>
  </Prerequisite>
  <Consequence>
    <Predicate Name="SystemCompromised">
      <Arg id="18" ArgName="SrcIPAddress"></Arg>
    </Predicate>
  </Consequence>
</HyperAlertType>

<HyperAlertType Name="HTTP_Cisco_Catalyst_Exec">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <Prerequisite>
    <Predicate Name="CiscoCatalyst3500XL">
      <Arg id="1" ArgName="DestIPAddress"></Arg>
    </Predicate>
  </Prerequisite>
  <Consequence>
    <Predicate Name="GainAccess">
      <Arg id="4" ArgName="DestIPAddress"></Arg>
    </Predicate>
  </Consequence>

```

```
</HyperAlertType>
```

```
<HyperAlertType Name="HTTP_Java">
```

```
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
```

```
  <Fact FactName="SrcPort" FactType="int"></Fact>
```

```
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
```

```
  <Fact FactName="DestPort" FactType="int"></Fact>
```

```
  <Prerequisite>
```

```
    <Predicate Name="JavaEnabledBrowser">
```

```
      <Arg id="12" ArgName="SrcIPAddress"></Arg>
```

```
    </Predicate>
```

```
  </Prerequisite>
```

```
  <Consequence>
```

```
    <Predicate Name="SystemCompromised">
```

```
      <Arg id="18" ArgName="SrcIPAddress"></Arg>
```

```
    </Predicate>
```

```
  </Consequence>
```

```
</HyperAlertType>
```

```
<HyperAlertType Name="HTTP_Shells">
```

```
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
```

```
  <Fact FactName="SrcPort" FactType="int"></Fact>
```

```
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
```

```
  <Fact FactName="DestPort" FactType="int"></Fact>
```

```
  <Prerequisite>
```

```
    <Predicate Name="VulnerableCGIBin">
```

```
      <Arg id="19" ArgName="DestIPAddress"></Arg>
```

```
    </Predicate>
```

```
    <Predicate Name="OSUNIX">
```

```
      <Arg id="26" ArgName="DestIPAddress"></Arg>
```

```
    </Predicate>
```

```
  </Prerequisite>
```

```
  <Consequence>
```

```
    <Predicate Name="GainAccess">
```

```
      <Arg id="4" ArgName="DestIPAddress"></Arg>
```

```

    </Predicate>
  </Consequence>
</HyperAlertType>

<HyperAlertType Name="Mstream_Zombie">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <Prerequisite>
    <Predicate Name="SystemCompromised">
      <Arg id="18" ArgName="DestIPAddress"></Arg>
    </Predicate>
    <Predicate Name="SystemCompromised">
      <Arg id="18" ArgName="SrcIPAddress"></Arg>
    </Predicate>
  </Prerequisite>
  <Consequence>
    <Predicate Name="ReadyToLaunchDDOSAttack">
    </Predicate>
  </Consequence>
</HyperAlertType>

<HyperAlertType Name="Port_Scan">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <Consequence>
<!--    <Predicate Name="GainServiceInfo">
      <Arg id="9" ArgName="DestIPAddress"></Arg>
    </Predicate> -->
    <Predicate Name="ExistService">
      <Arg id="22" ArgName="DestIPAddress"></Arg>
      <Arg id="23" ArgName="DestPort"></Arg>

```

```

    </Predicate>
  </Consequence>
</HyperAlertType>

<HyperAlertType Name="RIPAdd">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
</HyperAlertType>

<HyperAlertType Name="RIPExpire">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
</HyperAlertType>

<HyperAlertType Name="Rsh">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <Prerequisite>
    <Predicate Name="GainAccess">
      <Arg id="4" ArgName="DestIPAddress"></Arg>
    </Predicate>
    <Predicate Name="GainAccess">
      <Arg id="4" ArgName="SrcIPAddress"></Arg>
    </Predicate>
  </Prerequisite>
  <Consequence>
    <Predicate Name="SystemCompromised">
      <Arg id="18" ArgName="DestIPAddress"></Arg>
    </Predicate>
  </Consequence>
</HyperAlertType>

```

```

    <Predicate Name="SystemCompromised">
      <Arg id="18" ArgName="SrcIPAddress"></Arg>
    </Predicate>
  </Consequence>
</HyperAlertType>

<HyperAlertType Name="Sadmind_Amslverify_Overflow">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <Prerequisite>
    <Predicate Name="VulnerableSadmind">
      <Arg id="21" ArgName="DestIPAddress"></Arg>
    </Predicate>
    <Predicate Name="OSSolaris">
      <Arg id="25" ArgName="DestIPAddress"></Arg>
    </Predicate>
  <!--    <Predicate Name="GainAccess">
    <Arg id="4" ArgName="SrcIPAddress"></Arg>
  </Predicate> -->
</Prerequisite>
<Consequence>
  <Predicate Name="GainAccess">
    <Arg id="4" ArgName="DestIPAddress"></Arg>
  </Predicate>
</Consequence>
</HyperAlertType>

<HyperAlertType Name="Sadmind_Ping">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <Prerequisite>

```

```

    <Predicate Name="OSSolaris">
      <Arg id="25" ArgName="DestIPAddress"></Arg>
    </Predicate>
  </Prerequisite>
  <Consequence>
    <Predicate Name="VulnerableSadmin">
      <Arg id="21" ArgName="DestIPAddress"></Arg>
    </Predicate>
  </Consequence>
</HyperAlertType>

<HyperAlertType Name="SSH_Detected">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <Consequence>
<!--   <Predicate Name="GainInformation">
      <Arg id="7" ArgName="DestIPAddress"></Arg>
    </Predicate> -->
  </Consequence>
</HyperAlertType>

<HyperAlertType Name="Stream_DoS">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <Prerequisite>
    <Predicate Name="ReadyToLaunchDDOSAttack">
    </Predicate>
  </Prerequisite>
  <Consequence>
    <Predicate Name="DDOSAgainst">
      <Arg id="27" ArgName="DestIPAddress"></Arg>

```

```

    </Predicate>
  </Consequence>
</HyperAlertType>

```

```

<HyperAlertType Name="TCP_Urgent_Data">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <Consequence>
    <Predicate Name="SystemAttacked">
      <Arg id="17" ArgName="DestIPAddress"></Arg>
    </Predicate>
  </Consequence>
</HyperAlertType>

```

```

<HyperAlertType Name="TelnetEnvAll">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <Consequence>
    <Predicate Name="SystemAttacked">
      <Arg id="17" ArgName="DestIPAddress"></Arg>
    </Predicate>
  </Consequence>
</HyperAlertType>

```

```

<HyperAlertType Name="TelnetTerminaltype">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <Consequence>
    <Predicate Name="GainTerminalType">

```

```

        <Arg id="11" ArgName="DestIPAddress"></Arg>
    </Predicate>
</Consequence>
</HyperAlertType>

<HyperAlertType Name="TelnetXdisplay">
    <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
    <Fact FactName="SrcPort" FactType="int"></Fact>
    <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
    <Fact FactName="DestPort" FactType="int"></Fact>
    <Consequence>
        <Predicate Name="SystemAttacked">
            <Arg id="17" ArgName="DestIPAddress"></Arg>
        </Predicate>
    </Consequence>
</HyperAlertType>

<HyperAlertType Name="UDP_Port_Scan">
    <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
    <Fact FactName="SrcPort" FactType="int"></Fact>
    <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
    <Fact FactName="DestPort" FactType="int"></Fact>
    <Consequence>
<!--    <Predicate Name="GainServiceInfo">
        <Arg id="9" ArgName="DestIPAddress"></Arg>
    </Predicate> -->
        <Predicate Name="ExistService">
            <Arg id="22" ArgName="DestIPAddress"></Arg>
            <Arg id="23" ArgName="DestPort"></Arg>
        </Predicate>
    </Consequence>
</HyperAlertType>

</HyperAlertTypes>

```

`</KnowledgeBase>`

### A.3 Knowledge Base Used for DEFCON 8 Dataset

Because of the length of the XML file, I present it in an alternative format.

Hyper-alert Type	Prerequisite	Consequence
Admind	ExistService(DestIP, DestPort)	{VulnerableAdmind(DestIP)}
BackOrifice	OSWin95orWin98(DestIP)	{GainRootAccess(DestIP)}
Bind_Version_Request	BINDServer(DestIP) AND ExistService(DestIP, DestPort)	{VulnerableDNS(DestIP)}
Cisco_Syslog_DoS	VulnerableCiscoEquipment(DestIP) AND GainHardwareInfo(DestIP) AND GainOSInfo(DestIP)	{SystemAttacked(DestIP)}
CyberCop_Scanner		{GainVulnerability(DestIP)}
DNS_Iquery	VulnerableDNS(DestIP) AND ExistService(DestIP, DestPort)	{GainNetworkInfo(DestIP)}
DNS_Length_Overflow	GainOSInfo(DestIP) AND ExistService(DestIP, DestPort)	{GainRootAccess(DestIP)}
DNS_Zone_High_Port		{GainNetworkInfo(DestIP)}
Email_Almail_Overflow	VulnerablePOP3Client(DestIP) AND ExistService(DestIP, DestPort)	{GainAccess(DestIP)}
Email_Ehlo	SMTPSupportEhlo(DestIP) AND ExistService(DestIP, DestPort)	{GainSMTPInfo(DestIP)}
Email_Helo_Overflow	VulnerableMailServer(DestIP) AND ExistService(DestIP, DestPort)	{GainAccess(DestIP)}
Email_Qmail_Length	VulnerableMailServer(DestIP) AND ExistService(DestIP, DestPort)	{SystemAttacked(DestIP)}
Email_QuickStore	VulnerableWebStore(DestIP)	{GainRootAccess(DestIP)}
Email_Turn	SMTPSupportTurn(DestIP)	{MailLeakage(DestIP)}
Finger_Bomb	ExistFingerService(DestIP) AND ExistService(DestIP, DestPort)	{SystemAttacked(DestIP)}
Finger_Perl	ExistPerlFingerd(DestIP) AND VulnerableFingerService(DestIP) AND ExistService(DestIP, DestPort)	{SystemCompromised(DestIP)}
Finger_RTM	VulnerableFingerService(DestIP) AND GainOSInfo(DestIP) AND ExistService(DestIP, DestPort)	{GainRootAccess(DestIP)}

Hyper-alert Type	Prerequisite	Consequence
FSP_Detected	ExistFSPService(DestIP) AND GainOSInfo(DestIP) AND ExistService(DestIP, DestPort)	{SystemCompromised(DestIP)}
FTP_Args	VulnerableFTPService(DestIP) AND ExistService(DestIP, DestPort)	{GainRootAccess(DestIP)}
FTP_Bounce	ExistService(DestIP, DestPort)	{BypassFirewall, SpoofedAddress}
FTP_Format_String	VulnerableFTPService(DestIP) AND ExistService(DestIP, DestPort)	{SystemAttacked(DestIP)}
FTP_Pass	ExistService(DestIP, DestPort)	
FTP_PrivilegedBounce	ExistService(DestIP, DestPort) AND GainOSInfo(DestIP)	
FTP_PrivilegedPort	ExistService(DestIP, DestPort) AND GainOSInfo(DestIP)	
FTP_Put	ExistService(DestIP, DestPort) AND GainAccess(DestIP)	{SystemCompromised (DestIP)}
FTP_Root	ExistService(DestIP, DestPort) AND VulnerableFTPService(DestIP)	{GainRootAccess(DestIP)}
FTP_Syst	ExistService(DestIP, DestPort)	{GainOSInformation(DestIP)}
FTP_User	ExistService(DestIP, DestPort)	
Generic_Intel_Overflow	GainHardwareInfo(DestIP) AND ExistService(DestIP, DestPort)	{GainAccess(DestIP)}
HP_OpenView _SNMP_Backdoor	VulnerableHP(DestIP) AND GainOSInfo(DestIP) AND ExistService(DestIP, DestPort)	{GainNetworkInfo(DestIP)}
HTTP_ActiveX	SystemCompromised(DestIP)	{SystemAttacked(SrcIP)}
HTTP_AnyFormPost	VulnerableCGI(DestIP) AND GainFileInfo(DestIP) AND ExistService(DestIP, DestPort)	{GainAccess(DestIP)}
HTTP_BAT_Execute	OSWindows(DestIP) AND VulnerableWebServer(DestIP) AND GainFileInfo(DestIP) AND ExistService(DestIP, DestPort)	{GainAccess(DestIP)}

Hyper-alert Type	Prerequisite	Consequence
HTTP_Cachemgr	VulnerableCgi(DestIP) AND GainFileInfo(DestIP) AND ExistService(DestIP, DestPort)	{GainInformation(DestIP)}
HTTP_Campas	VulnerableCgi(DestIP) AND GainFileInfo(DestIP) AND ExistService(DestIP, DestPort)	{GainRootAccess(DestIP)}
HTTP_Carbo_Server	VulnerableWebServer(DestIP) AND ExistService(DestIP, DestPort)	{GainFileInfo(DestIP)}
HTTP_Cdomain	VulnerableCdomain(DestIP) AND GainFileInfo(DestIP) AND ExistService(DestIP, DestPort)	{GainAccess(DestIP)}
HTTP_Cold_Fusion	VulnerableColdFusion(DestIP) AND ExistService(DestIP, DestPort)	{GainAccess(DestIP)}
HTTP_ColdFusion _Admin	VulnerableColdFusion(DestIP) AND GainOSInfo(DestIP) AND ExistService(DestIP, DestPort)	{SystemAttacked(DestIP)}
HTTP_ColdFusion _FileExists	VulnerableColdFusion(DestIP) AND ExistService(DestIP, DestPort)	{GainFileInfo(DestIP)}
HTTP_ColdFusion _SourceWindow	VulnerableColdFusion(DestIP) AND ExistService(DestIP, DestPort)	{ReadFile(DestIP)}
HTTP_ColdFusion _ViewExample	VulnerableColdFusion(DestIP) AND ExistService(DestIP, DestPort)	{GainAccess(DestIP)}
HTTP_Cookie	CookieAccepted(SrcIP)	{GainInformation(SrcIP)}
HTTP_DotDot	VulnerableWebServer(DestIP) AND ExistService(DestIP, DestPort)	{GainFileInfo(DestIP)}
HTTP_EZMall2000	VulnerableWebServer(DestIP) AND ExistService(DestIP, DestPort)	{GainOrderLogInfo(DestIP)}
HTTP_FaxSurvey	VulnerableCGI(DestIP) AND GainFileInfo(DestIP) AND ExistService(DestIP, DestPort)	{GainAccess(DestIP)}
HTTP_FormMail	VulnerableCGI(DestIP) AND GainFileInfo(DestIP) AND ExistService(DestIP, DestPort)	{GainAccess(DestIP)}

Hyper-alert Type	Prerequisite	Consequence
HTTP_Head	ExistService(DestIP, DestPort)	BypassIDS
HTTP_IE_BAT	VulnerableWebServer(DestIP) AND MSIIS(DestIP) AND ExistService(DestIP, DestPort)	{GainAccess(SrcIP)}
HTTP_IIS\$DATA	VulnerableWebServer(DestIP) AND MSIIS(DestIP) AND ExistService(DestIP, DestPort)	{GainSourceCode(SrcIP)}
HTTP_Info2www	VulnerableCGI(DestIP) AND GainFileInfo(DestIP) AND ExistService(DestIP, DestPort)	{GainAccess(DestIP)}
HTTP_Java	JavaEnabledBrowser(SrcIP) AND SystemCompromised(DestIP) AND ExistService(DestIP, DestPort)	{SystemCompromised(SrcIP)}
HTTP_MachineInfo	VulnerableCgi(DestIP) AND GainFileInfo(DestIP) AND ExistService(DestIP, DestPort)	{GainHardwareInfo(DestIP)}
HTTP_Netscape _PageServices	NetscapeEnterpriseServer(DestIP) AND ExistService(DestIP, DestPort)	{GainFileInfo(DestIP)}
HTTP_Netscape _SpaceView	NetscapeEnterpriseServer(DestIP) AND ExistService(DestIP, DestPort)	{GainInformation(DestIP)}
HTTP_Novell_Files	VulnerableCGI(DestIP) AND GainFileInfo(DestIP) AND ExistService(DestIP, DestPort)	{GainFileInfo(DestIP)}
HTTP_NphTestCgi	VulnerableWebServerDestIP) AND GainFileInfo(DestIP) AND ExistService(DestIP, DestPort)	{GainFileInfo(DestIP)}
HTTP_PDGSoft	VulnerableWebStore(DestIP) AND ExistService(DestIP, DestPort)	{GainOrderLogFile(DestIP)}
HTTP_PHF	VulnerableCGI(DestIP) AND GainFileInfo(DestIP) AND ExistService(DestIP, DestPort)	{GainRootAccess(DestIP)}
HTTP_PHP_Read	VulnerableCGI(DestIP) AND GainFileInfo(DestIP) AND ExistService(DestIP, DestPort)	{GainFileInfo(DestIP)}

Hyper-alert Type	Prerequisite	Consequence
HTTP_RobotsTxt	ExistService(DestIP, DestPort)	{GainInformation(DestIP)}
HTTP_SGI_Wrap	VulnerableCgi(DestIP) AND GainOSInfo(DestIP) AND GainFileInfo(DestIP) AND ExistService(DestIP, DestPort)	{GainFileInfo(DestIP)}
HTTP_Shells	VulnerableCGIBin(DestIP) AND GainOSInfo(DestIP) AND ExistService(DestIP, DestPort)	{GainAccess(DestIP)}
HTTP_SiteCsc_Access	VulnerableWebServer(DestIP) AND ExistService(DestIP, DestPort)	{GainDBAccess(DestIP), GainAccess(DestIP)}
HTTP_TestCgi	VulnerableWebServer(DestIP) AND GainFileInfo(DestIP) AND ExistService(DestIP, DestPort)	{SystemAttacked(DestIP), GainFileInfo(DestIP)}
HTTP_Unix_Passwords	ExistService(DestIP, DestPort) AND GainOSInfo(DestIP)	{GainRootAccess(DestIP)}
HTTP_WebFinger	VulnerableCgi(DestIP) AND GainFileInfo(DestIP) AND ExistService(DestIP, DestPort)	
HTTP_Websendmail	VulnerableCGI(DestIP) AND GainFileInfo(DestIP) AND ExistService(DestIP, DestPort)	{GainAccess(DestIP)}
HTTP_WebSite_Sample	VulnerableWebServer(DestIP) AND ExistService(DestIP, DestPort)	{GainAccess(DestIP)}
HTTP_WebSite_Uploader	VulnerableWebServer(DestIP) AND ExistService(DestIP, DestPort)	{GainAccess(DestIP)}
HTTP_WebStore	VulnerableWebStore(DestIP) AND ExistService(DestIP, DestPort)	{GainOrderLogFile(DestIP)}
Ident_Error	OSUNIX(DestIP)	{GainInformation(DestIP)}
Ident_Linux_DoS	VulnerableIdent(DestIP) AND OSLinux(DestIP)	{SystemAttacked(DestIP)}
IPDuplicate		
IPFrag		{SystemAttacked(DestIP)}
IPHalfScan	ExistService(DestIP, DestPort)	{GainServiceInfo(DestIP)}

Hyper-alert Type	Prerequisite	Consequence
IPUnknownProtocol		
IRDP_Gateway_Spoof	GainOSInfo(DestIP)	{SystemAttacked(DestIP)}
ISS		{GainVulnerability(DestIP)}
Kerberos_User_Snarf	KerberosIV(DestIP)	{GainUserName(DestIP)}
Loki	SystemCompromised(DestIP)	
Mstream_Zombie	SystemCompromised(SrcIP) AND SystemCompromised(DestIP)	{ReadyToLaunchStreamDOSAttack}
Netbus	OSWindows(DestIP)	{GainRootAccess(DestIP)}
Nmap_Scan		{GainNetworkInfo(DestIP), GainOSInfo(DestIP)}
PingFlood		{SystemAttacked(DestIP)}
PmapDump	OSUnix(DestIP)	{VulnerableRPCService(DestIP)}
POP_Fold_Overflow	VulnerablePOPDaemon(DestIP) AND ExistService(DestIP, DestPort)	{GainAccess(DestIP)}
POP_QPopAuth_Overflow	VulnerablePOPDaemon(DestIP) AND ExistService(DestIP, DestPort)	{GainRootAccess(DestIP)}
POP_QPopCommand_Overflow	VulnerablePOPDaemon(DestIP) AND ExistService(DestIP, DestPort)	{GainRootAccess(DestIP)}
Port_Scan	ExistHost(DestIP)	{ExistService(DestIP, DestPort)}
Queso_Scan		{GainOSInfo(DestIP), GainNetworkInfo(DestIP)}
Rexec	GainAccess(DestIP) AND GainAccess(SrcIP)	{SystemCompromised(DestIP)}
RIPAdd		
Rlogin_Froot	VulnerableRlogin(DestIP) AND GainOSInfo(DestIP)	{GainRootAccess(DestIP)}
SourceRoute		{BypassAuthentication}
Sadmind_Amslverify_Overflow	VulnerableRPCService(DestIP) AND ExistService(DestIP, DestPort)	{GainAccess (DestIP)}
Satan		{GainVulnerability(DestIP)}
ServiceScan		{ExistService(DestIP, DestPort), ExistHost(DestIP)}
Smurf	ExistHost(DestIP)	{SystemAttacked(DestIP)}

Hyper-alert Type	Prerequisite	Consequence
SNMP_Set	ExistService(DestIP, DestPort) AND GainOSInfo(DestIP)	{SystemAttacked(DestIP)}
SNMP_Suspicious_Get	ExistService(DestIP, DestPort)	{GainInformation(DestIP)}
SSH_Detected		{GainInformation(DestIP)}
Stream_DoS	ReadyToLaunchStreamDOSAttack	{DOSAttackLaunched}
Sun_SNMP_Backdoor	ExistService(DestIP, DestPort) AND Solaris2.6(DestIP)	{GainRootAccess(DestIP)}
SYNFlood	ExistService(DestIP, DestPort)	{SystemAttacked(DestIP)}
TCP_Overlap_Data		
TCP_Urgent_Data		{SystemAttacked(DestIP)}
TelnetEnvAll	ExistService(DestIP, DestPort)	{SystemAttacked(DestIP)}
TelnetTerminaltype	ExistService(DestIP, DestPort)	{GainTerminalType(DestIP)}
TelnetXdisplay	ExistService(DestIP, DestPort)	{SystemAttacked(DestIP)}
ToolTalk_Overflow	VulnerableRPCService(DestIP) AND GainOSInfo(DestIP)	{GainRootAccess(DestIP)}
Trace_Route		{GainNetworkInfo(DestIP)}
TrinooDaemon	SystemCompromised(SrcIP) AND SystemCompromised(DestIP)	{ReadyToLaunchDOSAttack}
UDP_Port_Scan	ExistHost(DestIP)	{GainServiceInfo(DestIP)}
Windows_Access_Error		

\* The *fact* component of all these hyper-alert types is {SrcIP, SrcPort, DestIP, DestPort}.

## Appendix B

### JAVA Classes

There are twelve Java classes in this tool: *AssignGraphID*, *CorrelationEngine*, *Correlator*, *FocusedAnalysis*, *GraphDecomposition*, *GraphOutput*, *GraphReduction*, *HATMapping*, *HyperAlertType*, *KnowledgeBase*, *SimplifiedGraphOutput*, and *TransitiveExclusion*. Among them, *Correlator* is the main one to call other classes.

*AssignGraphID* is to separate the prepare-for relationships into different graphs. Table *GraphEdges* is the output.

*CorrelationEngine* is the class to preprocess the alerts and do the correlation. First, it retrieves all the hyper alert types from the knowledge base; Then, it generates hyper alerts by mapping the original alerts to the hyper alert types. Finally, the algorithms we propose in 4.2 and 4.3 are used to generate the prepare-for relationships. Table *CorrelatedAlerts* is the output which contains the prepare-for relationships.

*FocusedAnalysis* is the class to support the focused analysis utility. It requests the user has some prior knowledge of the network you protect or the attacks so that the user can input the focusing constraint. It uses the algorithm in 4.7. Table *GraphEdgesFocus* is the output.

*GraphDecomposition* is the class to support the graph decomposition utility. The algorithm in 4.8 is used. It doesn't require that user has some prior knowledge. Table *GraphEdgesDecomposed* is the output.

*GraphOutput* is to generate the DOT formatted graph file which can be displayed by GraphViz using the original prepare-for relationships which are stored in *GraphEdges*, or *GraphEdgesFocus* or *GraphEdgesDecomposed*.

*GraphReduction* is the class to support the adjustable graph reduction utility. The algorithm described in 4.6 is used. Either *SimplifiedGraphEdges*, *SimplifiedGraphFocus* or *SimplifiedGraphDecomposed* is the output table depends on different input table: *GraphEdges*, *GraphFocus* or *GraphDecomposed*.

*HATMapping* is the class to map the original alert name to hyper alert type name. To accommodate the situation that the hyper alert type name may be different with the original alert name, or several original alert names map to one hyper alert type, user can write this mapping relationship in the property file. The tool gets the hyper alert type information from knowledge base and expects there could be such mappings that the keywords are the hyper alert type names, their values are the respective original alert names. Once these types of information are located, they are inserted into the table *HATMapping* which will be used to map an original alert to a hyper alert type in the *CorrelationEngine*. If no such kind of information is appeared in the property file, we assume the hyper alert type has the same name with the original alert.

*HyperAlertType* is the class to get the hyper alert type information from the knowledge base.

*KnowledgeBase* is a class to parse the XML file and transfer the contents in the XML file to the knowledge base tables in the database. In our implementation, in order to work in whatever Java version, we use an outside open source XML parser – Xerces.

*SimplifiedGraphOutput* is to generate the DOT formatted graph file which can be outputted using GraphViz based on the aggregated prepare-for relationships. The input table could be *SimplifiedGraphEdges*, *SimplifiedGraphFocus* or *SimplifiedGraphDecomposed*.

*TransitiveExclusion* is to delete the transitive edges in a graph. It uses the algorithm described in 4.5.

## Appendix C

# User Installation & Operation Manual

### C.1 Introduction

This tool takes the alerts generated by Intrusion Detection System(IDS) as input and applies our correlation techniques upon them. The false alerts will be filtered out by the tool and a set of correlation graphs will be generated to show the attack scenarios hiding behind the alerts.

In a word, this tool aims to assist human user in analyzing the potentially large amount of intrusion alerts and uncovering the high-level attack strategies.

This tool is written using Java, with JDBC to access the database. The GraphViz package [1] is adopted to visualize the results.

The techniques that this tool was based on can be found in [4] and [5].

### C.2 Installation

#### C.2.1 System Environment

This tool is developed with Java 2. A relational database is a must to use this tool. In order to support knowledge base generation, Xerces Java Parser is also needed. We use Java 2 SDK Standard Edition v1.3.1\_04, and Xerces Java Parser v1.4.4 [3].

Our tests were conducted on Windows 2000 with Microsoft SQL Server 2000. The original alerts were generated by RealSecure Network Sensor 6.0 [2].

#### C.2.2 Checklist

Here is the list you need to run this tool.

- Java

- Database
- JDBC Driver
- Raw alerts generated by IDS which are stored in database
- Knowledge Base
- Xerces Java Parser v1.4.4

If you haven't a knowledge base ready, and you want to generate it through an XML file, you need it.

1. Go to <http://xml.apache.org> and download a Xerces Java Parser;
2. Install it and make sure the package is in your classpath;

- GraphViz

GraphViz is necessary to visualize the correlation results.

### C.2.3 Download

The software package can be downloaded at <http://discovery.csc.ncsu.edu/software>. A sample knowledge base, a sample test database, and instructions to repeat our experiments are also available on the above page.

## C.3 How to Use This Tool

The core of this tool consists of a knowledge base, an alert preprocessor, a correlation engine, a hyper-alert correlation graph generator, and three useful utilities: adjustable graph reduction, focused analysis and graph decomposition.

### C.3.1 Knowledge Base

The knowledge base contains the necessary information about hyper-alert types as well as relationships between predicates. To simplify the implementation, we assume that each hyper-alert type is uniquely identified by its name, and there is no negation in the prerequisite nor the consequence of any hyper-alert type.

In the XML file, there are basically three sections: *Predicates*, *Implications* and *HyperAlertTypes*.

In the *Predicates* section, the predicate name which works as the key of the predicate and the arguments of the predicate are specified. In order to make the validation strict and easier, a unique argument id is needed for each argument for all the predicates. Here is an example,

```
<Predicate Name="ExistService">
<Arg id="14" Pos="1" Attr="varchar(15)"/>
```

```
<Arg id="15" Pos="2" Attr="int"/>
</Predicate>
```

In this example, the predicate *ExistService* has two arguments, one is of char, the other is integer. Each of them should have a unique id. The *Pos* specifies that the char is the first argument and the integer is the second argument. The whole predicate is *ExistService(varchar, int)*.

Each predicate which appears in the *Implications* section or *HyperAlertTypes* section must be declared here. This part goes into the *Predicate* table in the database.

In the *Implications* section, there are two kinds of implications: normal and phantom. The phantom implications mean that the implied predicate is unclear to us, but it is clear to the attacker. For example, the consequence of *FTP\_Syst* is *GainOSInfo*. Through it, the attacker knows what operating system is in the target machine, either *OSLinux* or *OSWindows*, etc. But we, the correlator, cannot get this detailed information. So, we call this kind of implication, *GainOSInfo* implies *OSLinux*, phantom implication. Phantom implications and normal implications have the same effect in the correlation process. Having phantom can correlate more related alerts, which may be missed otherwise; however, it may also increase the false correlation rate.

Here is an example of implication:

```
<Implication Phantom="Yes">
<ImpliedName>OSLinux</ImpliedName>
<ArgMap>
<ImpliedArg id="39"></ImpliedArg>
<ImpliedArg id="23"></ImpliedArg>
</ArgMap>
</Implication>
```

It represents *GainOSInfo(GainOSInfoArg)* implies *OSLinux(OSLinuxArg)*. The *ImpliedName* and *ImpliedName* are obvious. *ArgMap* represents the argument mapping relationship. In this example, the argument id of *GainOSInfoArg* defined in the *Predicates* section is 23 and the argument id of *OSLinuxArg* is 39. They should match the id which is defined in the *Predicates* section.

This part goes into the *Implication* table in the database.

The hyper alert types are defined in the *HyperAlertTypes* section. Each *HyperAlertType* may consist several parts: *Fact*, *Prerequisite* and *Consequence* if it has. Among them, *Fact* is a must of a hyper alert type; *Prerequisite* and *Consequence* are optional. The *HyperAlertTypes* section will be mapped into several tables: *HATFact* to store the fact information, *HATPrereq* to store the prerequisite information and *HATConseq* to store the consequence information.

Please refer to our papers [4] [5] for detailed table structure.

We provide a module to parse this knowledge base XML file and put them into database.

### C.3.2 Property File

There is a property file named “Correlator.properties”. Here is an example:

```
#section 1: database connection
dbDriver = com.microsoft.jdbc.sqlserver.SQLServerDriver
dbURL = jdbc:microsoft:sqlserver://balisong:1433;DatabaseName=defcon8-vol1;SelectMethod=cursor

#newTable = insert into events select distinct e.sid, e.cid, e.signature, s.sig_name, e.timestamp,
i.ip_src, i.ip_dst, t.tcp_sport, t.tcp_dport from event e, signature s, iphdr i, tcphdr t where e.signature=s.sig_id
and e.sid=i.sid and e.cid=i.cid and e.sid=t.sid and e.cid = t.cid;

#section 2: knowledge base
Generate_Knowledge_Base = true
Knowledge_Base_XML_File = DARPA_2K_final.xml

#section 3: correlation engine
AlertTable = events
# column names of RealSecure & MSSQL
# the following 4 are static keywords
AlertID = EventID
AlertName = OrigEventName
BeginTime = EventDate
EndTime = EventDate
# the following are “dynamic” keywords based on knowledge base
SrcIPAddress = SrcIPAddress
SrcPort = SrcPort
DestIPAddress = DestIPAddress
DestPort = DestPort

Original_Graph_Output = darpa_dmz1.txt

#Below are the three utilities
#section 4: graph reduction
Aggregation_Time_Interval = -1
```

```
Graph_Reduction_Output = dmz_a.txt
```

```
#section 5: focused analysis
```

```
Focused_Analysis_GraphID = 9
```

```
Focusing_Constraint = SrcIPAddress='152.14.53.39'
```

```
Focused_Aggregation = true
```

```
Focused_Aggregation_Time_Interval =
```

```
Focused_Output =
```

```
#section 6: graph decomposition
```

```
Decomposition_ID = 9
```

```
Clustering_Constraint = h1.SrcIPAddress=h2.SrcIPAddress
```

```
Decomposition_Aggregation = true
```

```
Decomposition_Aggregation_Time_Interval =
```

```
Decomposition_Output =
```

```
#section 7: mapping between original alert name and hyper alert name (OPTIONAL)
```

```
original_alert_name = hyper_alert_name
```

You can specify the parameter needed to communicate with database in section 1. The *dbDriver* is used as the jdbc driver name and the *dbURL* is used as the database url.

The knowledge base parameters are specified in section 2. If you want to generate a knowledge base through an XML file, please set *Generate\_Knowledge\_Base* to *true* and give the relative path of the XML file in *Knowledge\_Base\_XML\_File*. If you don't want to generate the knowledge base, just set *Generate\_Knowledge\_Base* to *false*.

We assume that all original alerts generated by IDS are stored in a single table. So, if there are more than one table involved, please preprocessing them by merging into one table. You can use the *newTable* property to merge it if it can be done by SQL statements. If the *newTable* is set, we will process the SQL statements you have specified here before other processes.

The correlation engine parameters are specified in section 3. These properties are used to specify the alert table name and the column names. *AlertTableName* is a fixed keyword parameter to tell the tool which table stores the original alerts. *AlertID*, *AlertName*, *BeginTime* and *EndTime* are also fixed keyword parameters. They are used to tell the column names of original alert id, alert name, alert begin time and alert end time in the *AlertTableName*. The name and number of the other parameters in this section are based on knowledge base. The keywords are the attribute names

used in knowledge base and the values are their column names in *AlertTableName*. In this example, we have attribute name *SrcIP*, *SrcPort*, *DestIP* and *DestPort* in the knowledge base, so we need to specify their corresponding column names in *AlertTableName*. They have the same names here.

Set a file name to *Original\_Graph\_Output* will generate a DOT formatted graph file based on the original prepare-for relationships which can be displayed by GraphViz.

Sections 4, 5 and 6 contain the parameters used for three utilities: graph reduction, focused analysis and graph decomposition. Please refer to the section “Execution” for detailed usages.

We assume that hyper alert has the same name as its original alert. But to accommodate the situation that hyper alert has the different name with its original alert, or several different alerts could be mapped to one hyper alert, we use section 7 to address it. You can write the name mapping relationship as:

*ORIGINAL\_ALERT\_NAME = HYPER\_ALERT\_NAME*, such as:

*Sadmind = Portmap request sadmind*

The property keyword is the original alert name and the property value is the hyper alert name in knowledge base. If nothing is specified in section 7, we assume they have the same name.

Any line started with “#” is a comment line.

### C.3.3 Execution

If you have configured the “Correlator.properties” file and have a knowledge base XML file, you are ready to go. What you need to do is to set property parameters in the property file and turn on of turn off the corresponding switch. The command line would be:

```
java Correlator [-Correlation] [-TransitiveExclusion] [-GraphReduction]
[-FocusedAnalysis] [-GraphDecomposition] user [password]
```

After it executes successfully, you can expect some graph files output with the name you specified in the property file. Now, you can use GraphViz to generate the graph with the command line:

```
dot -Tps inputFileName -o outFileName.ps
```

Let’s explain the first command line in detail. When you put *-Correlation* in the command line, the correlation engine will be activated and it will search the correlation related parameters in the section 3 of the property file.

When *-TransitiveExclusion* appears in the command line, the tool will delete the transitive edges in the prepare-for relationship.

When *-GraphReduction* is in the command line, the graph reduction utility will be activated and it will search the graph reduction parameters in the section 4 of the property file.

When *-FocusedAnalysis* is in the command line, the focused analysis utility will be activated and it will search the needed parameters in the section 5 of the property file.

Similarly, when *-GraphDecomposition* is in the command line, the graph decomposition utility will be activated and it will search the needed parameters in the section 6 of the property file.

For example, the command line

```
java Correlator -Correlation -TransitiveExclusion -GraphReduction -GraphDecomposition user-Name userPassword
```

will activate the correlation engine, the graph reduction utility and the graph decomposition utility, but not the focused analysis utility. And, keeps the transitive edges.

The user and password are used for database authentication. This user needs to have full rights in the database.

### Correlation

Set *-Correlation* in the command line; Set the alert related parameters in the property file following the instructions in the section “property file”, you can correlate the alerts as you want.

### Graph Reduction

Graph reduction is to aggregate the alerts with same type in each graph. To activate it, you must set *-GraphReduction* in the command line. Also, you must give the aggregation time interval. It is in milli second unit. You can set *Aggregation\_Time\_Interval* to *-1* which will make the aggregation engine to aggregate all the alerts with same type no matter how far in time they are. Give a value to *Graph\_Reduction\_Output* will generate a DOT formatted graph file based on the aggregated prepare-for relationship, which can be displayed by GraphViz.

### Focused Analysis

Focused analysis is implemented on the basis of focusing constraint. To activate it, you must set *-FocusedAnalysis* in the command line. Also, please specify which graph you want to analyze and what constraint you want to use. So you must have some knowledge of the graph and the alerts before you can do the focused analysis.

*Focused\_Analysis\_GraphID* is on which graph you want to do the focused analysis;

*Focusing\_Constraint* is an SQL valid presentation that can be inserted into a query like “select alertID from alertTable where *Focusing\_Constraint*”;

*Focused\_Aggregation* will activate/inactivate the aggregation engine on the focusd analysis;

*Focused\_Aggregation\_Time\_Interval* is similar to *Aggregation\_Time\_Interval* in section 4;

*Focused\_Output* is used for the output graph.

### Graph Decomposition

Graph decomposition is to decompose a big graph into several sub graphs based on a user-specified clustering constraint [4]. To activate graph decomposition, you must set *-GraphDecomposition* in the command line.

*Decomposition\_ID* is on which graph you want to do the decomposition;

*Clustering\_Constraint* is an SQL valid presentation that can be inserted into an SQL query directly. It must be in a relationship between *h1.attribute* and *h2.attribute* since we use *h1* and *h2* as two alerts in the query;

*Decomposition\_Aggregation* will activate/inactive the aggregation engine on the decomposed sub graphs;

*Decomposition\_Aggregation\_Time\_Interval* is similar as *Aggregation\_Time\_Interval* in section 4;

*Decomposition\_Output* is used to specify the output graph.

## C.4 Sample Execution Procedure

1. java Correlator -Correlation -TransitiveExclusion userName password
2. dot -Tps darpa\_dmz1.txt -o outputFileName.ps

To generate the graph using GraphViz. Here, “darpa\_dmz1.txt” is the file name we specified for the “Original\_Graph\_Output” in “Correlator.properties” file. The “outputFileName” is a ps formatted file which will be generated by GraphViz.

## C.5 Trouble Shooting

1. Make sure the *dbDriver* and *dbURL* in the property file is set correctly. Remember the tool uses *DataManager.getConnection(url, user, password)* to connect to the database.
2. If you use the JDBC-ODBC driver, it may take very long time to execute the SQL statement to delete the transitive edges. If this does happen, either you have patience to wait, or you just execute this SQL statement by hand.
3. If you use Java 2 SDK v1.4 and the JDBC-ODBC driver at the same time, you may have some problem to communicate with the database. You can use Java 2 SDK v1.3 to get around the problem.

## List of References

- [1] <http://www.research.att.com/sw/tools/graphviz/>
- [2] <http://www.iss.net>
- [3] <http://xml.apache.org>
- [4] Peng Ning, Yun Cui, Douglas S. Reeves, "Analyzing Intensive Intrusion Alerts Via Correlation," To appear in *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID 2002)*, Zurich, Switzerland, October 2002.
- [5] Peng Ning, Yun Cui, Douglas S. Reeves, "Constructing Attack Scenarios through Correlation of Intrusion Alerts," To appear in *Proceedings of the 9th ACM Conference on Computer & Communications Security*, Washington D.C., November 2002.