AN EMPIRICAL APPROACH TO MODELING UNCERTAINTY IN INTRUSION ANALYSIS

by

SAKTHIYUVARAJA SAKTHIVELMURUGAN

B.E., Anna University, India, 2005

A THESIS

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas
2009

Approved by:

Major Professor
Xinming Ou

Abstract

A well-known problem in current intrusion detection tools is that they create too many low-level alerts and system administrators find it hard to cope up with the huge volume. Also, when they have to combine multiple sources of information to confirm an attack, there is a dramatic increase in the complexity. Attackers use sophisticated techniques to evade the detection and current system monitoring tools can only observe the symptoms or effects of malicious activities. When mingled with similar effects from normal or non-malicious behavior they lead intrusion analysis to conclusions of varying confidence and high false positive/negative rates.

In this thesis work we present an empirical approach to the problem of modeling uncertainty where inferred security implications of low-level observations are captured in a simple logical language augmented with uncertainty tags. We have designed an automated reasoning process that enables us to combine multiple sources of system monitoring data and extract highly-confident attack traces from the numerous possible interpretations of low-level observations. We have developed our model empirically: the starting point was a true intrusion that happened on a campus network we studied to capture the essence of the human reasoning process that led to conclusions about the attack. We then used a Datalog-like language to encode the model and a Prolog system to carry out the reasoning process. Our model and reasoning system reached the same conclusions as the human administrator on the question of which machines were certainly compromised. We then automatically generated the reasoning model needed for handling Snort alerts from the natural-language descriptions in the Snort rule repository, and developed a Snort add-on to analyze Snort alerts. Keeping the reasoning model unchanged, we applied our reasoning system to two third-party data sets and one production network. Our results showed that the reasoning

model is effective on these data sets as well. We believe such an empirical approach has the potential of codifying the seemingly ad-hoc human reasoning of uncertain events, and can yield useful tools for automated intrusion analysis.

Table of Contents

Table of Contents iv

List of Figures vi

List of Tables vii

Acknowledgments viii

Dedication ix

1 Introduction 1
- 1.1 Motivation . . . 1
  - 1.1.1 A true-life incident . . . 3
- 1.2 Empirical approach . . . 6
- 1.3 Contributions . . . 7

2 Background and related work 11
- 2.1 Intrusion detection system . . . 11
  - 2.1.1 Types of IDS . . . 11
  - 2.1.2 IDS detection types . . . 12
  - 2.1.3 Problems in IDS . . . 13
- 2.2 Related work . . . 14
  - 2.2.1 Reasoning under uncertainty . . . 14
  - 2.2.2 Alert reduction techniques . . . 15
  - 2.2.3 Attack graph . . . 17
  - 2.2.4 Other closely related work . . . 18

3 Reasoning model 19
- 3.1 Modeling uncertainty . . . 19
- 3.2 Observation correspondence . . . 21
- 3.3 Internal model . . . 22

4 Reasoning methodology 25
- 4.1 Reasoning framework . . . 25
- 4.2 Application of inference rules . . . 26
- 4.3 Proof strengthening . . . 27
- 4.4 Implementation . . . 29
- 4.5 Pre-processing . . . 31

4.6 Handling time 32

5 Automating model building for Snort **33**
5.1 Snort 33
5.1.1 Snort architecture 34
5.1.2 Snort signature 35
5.1.3 Snort alert 36
5.2 Knowledge base creation 37
5.2.1 Classtype 39
5.2.2 Snort rule document 39
5.3 Snort alert translation 41

6 Experiments **43**
6.1 Experimental setup 43
6.2 Data sets 44
6.2.1 Treasure Hunt data set 44
6.2.2 Experiment on data collected on a honeypot 46
6.2.3 Experiments on a production system 46
6.3 Results 47
6.4 Observations from the data sets 47

7 Conclusion **49**
7.1 Future work 49

Bibliography **57**

A Snort – Intrusion detection system **58**
A.1 Snort default classtype 59
A.2 An example Snort rule document 60
A.3 Snort database schema 62

List of Figures

1.1 Case Study: A true life incident . . . 4

3.1 Observation correspondence . . . 21
3.2 Internal model . . . 23

4.1 System architecture . . . 26
4.2 Result of applying the reasoning system on the case study . . . 30
4.3 Preprocessing: Summarization . . . 31

5.1 SnIPS system architecture . . . 34
5.2 An example Snort rule . . . 35
5.3 An example Snort alert . . . 37
5.4 Example translated Snort alerts . . . 41

6.1 Treasure hunt network topology . . . 44
6.2 Partial output trace from the reasoning system . . . 45
6.3 Data distribution in the three data sets . . . 48

List of Tables

3.1 Uncertainty modes 20

5.1 Automatically created obsMap 38

6.1 Reduction of alert 47

Acknowledgments

I express my deepest appreciation to advisor, Dr. Xinming Ou, for his support. Under his guidance, I have learned many things in the last two years. It has been a pleasure and an honor working with him.

I am grateful to Dr. S. Raj Rajagopalan for his continuous support from the beginning of this work. His suggestions were instrumental to improving the quality of my work over the years.

My special thanks to supervisory committee members, Dr. Doina Caragea and Dr. David Schmidt for their comments and valuable feedback that helped in refining the quality of this work.

I would like to thank my friends, Vijaya Iyer and Aditi Shukla for their assistance in refining the earlier draft versions of this thesis.

This work was partially supported by the U.S. National Science Foundation under Grant No. 0716665. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

Dedication

To Amma, Appa and Meera, who are always there for me.

Chapter 1

Introduction

1.1 Motivation

Universities, business and government organizations depend on computer networks for most of their day-to-day activities. Increasingly personal information such as credit card numbers, social security numbers, and other sensitive information are stored on computers. This has resulted in a rise in attempts made to compromise a computer and to gain access to unauthorized information. An attacker gains access to remote network resources using vulnerabilities found in software and protocols or through social engineering. Proper software maintenance (operating system patches, software updates, *etc.*) and restricted network access using firewalls is no longer a reliable solution to prevent attackers from infiltrating the network. Also, recent studies show zero-day vulnerabilities are exploited more aggressively than in the past. In order to identify intrusion of computing resources and have a situational awareness of the network, system administrators today perform intrusion analysis, a combination of intrusion detection and computer forensics. This activity is an inexact science; system administrators use a combination of intuition, experience, and low-level tools to create and support positive or negative judgment about a security event.

However, with increase in the number of computing resources in a network, the volume of the low-level system events collected is too high. This strains the intuitive capacity of the human analyst. Many intricate attacks common today, make use of multiple vulnerabilities

in a system. To identify such attacks, system administrators have to look into several places (network packets, system logs, host configuration, *etc.*) and confirm an attack. For example, in most enterprise networks, the web server accessible from the Internet happens to be the first target. This sets a stage for the attacker to penetrate deeper into the network. What follows might be a series of host compromise to achieve the attacker's goal, such as steeling credit card information from a SQL server. With diverse sources to look for intrusion evidence, the complexity involved increases dramatically for a system administrator.

Another important problem with the output from low-level tools is that they give a varying degree of uncertainty about a security event. Some observations [1] give only low confidence about attacks, and few give a high degree of confidence. The degree of uncertainty associated with the observations root from many factors. Some of the reasons include the nature of the attack, effectiveness of the tool, frequency of the attack or how similar the attack is to a normal activity. An increased network activity during office hours gives low confidence about an ongoing attack compared to a high network activity around midnight. Again, both these events could be benign or malignant depending on what caused the network activity. It either can be an attacker downloading a list of credit card information or an authorized event like software updates[2]. Thus, it is hard to rely on a single piece of evidence to conclude an attack or machine compromise.

While the low-level observations (network packets, server logs, *etc.*) all have potential implication for attack possibilities, few, if any of them directly provide a zero/one judgment at the high-level abstraction (e.g. a machine has been compromised). Nevertheless, in many remote intrusions a relatively small number of critical observations *taken together* are sufficient to show that an attack has certainly happened as well as how it progressed. The bottleneck emerges from consuming disparate sources of information to infer about a complex attack.

[1] Alert and observation are used interchangeably in this text, they both mean the report issued by a tool on observing a suspicious activity

[2] Software updates are usually scheduled during off peak hours.

System administrators are highly time-constrained – an automatic tool that can sift through the ocean of uncertain data to quickly and accurately locate the problem areas or reduce the search space will be invaluable in practice. There are several technical challenges such as combining multiple sources of information and handling uncertainty. The most difficult of the two is quantifying the degree of certainty in various assertions regarding possible attack activities, and tracking how the uncertainty changes through correlation. Through our interaction with highly capable system administrators who face the challenge of uncertainty every day while protecting enterprise systems, we observed that human analysts do well without relying on any numerical measures. We illustrate a true-life story in section 1.1.1. This example clearly shows progress made by the human analyst for identifying the intrusion and the various levels of uncertainty associated with each incident.

1.1.1 A true-life incident

Consider the following sequence of events (see figure 1.1) that actually occurred at a university campus. The system administrator noticed an abnormally large spike in campus-network traffic (*Observation 1*). He took the netflow dump for that time period, searched it for known malicious IP addresses, and identified that four Trend Micro (anti-malware) servers had initiated IRC connections to some known BotNet controllers (*Observation 2*). The system administrator suspected that the four Trend Micro servers had been compromised. At the console of one of the servers he dumped the memory, from which he found what appeared to be malicious code modules (*Observation 3*). He also looked at the open TCP socket connections and noticed that the server had been connecting to some other Trend Micro servers on campus through the IRC channel (*Observation 4*). He concluded that all those servers were compromised with a zero-day vulnerability in the Trend Micro server software. He did the same for the other identified servers and found even more compromised servers. Altogether, the system administrator identified 12 compromised machines and took them offline. When the administrator first noticed the spike in network traffic,

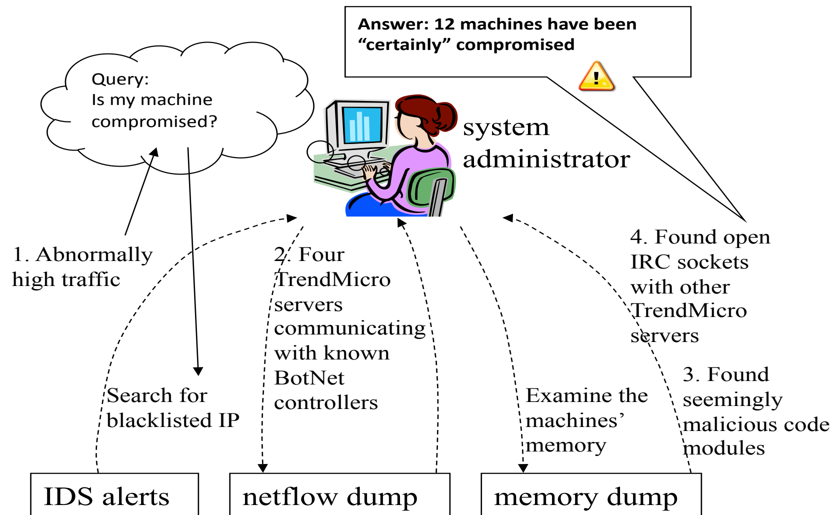

Figure 1.1: *Case Study: A true life incident*

the questions facing him were: is the network experiencing an attack and if so, which machines were compromised. However, none of the low-level observations *alone* could give him a definitive answer to these high-level questions. Observation 1 (traffic spike) could indicate a variety of causes, many of which are benign[3]. Observation 2 (connections to BotNet controllers) has a higher chance of being an indication of malicious activity and hence a higher degree of likelihood that the identified hosts are compromised. However, an IRC connection being made to a "known" BotNet controller does not necessarily mean that the machine has been taken over (compromised). The list of "known" BotNet controllers may contain false positives or it could be because somebody was probing BotNet controllers for

[3]To take an example, the "Microsoft Patch Tuesday" often significantly increases network traffic in an enterprise network.

research purposes. (The interviewed system administrator suggested this false positive as he did this himself on a periodic basis.) Observation 3 (suspicious code in memory) is also a strong indication that the machine may have been controlled by an attacker. But it is not always easy to determine whether a suspicious module found in the memory dump is indeed malicious, especially with zero-day vulnerabilities. So this alert also has some false positive. Observation 4, like observation 2, cannot definitely prove that the machines observed are under the control of attackers because IRC channels are occasionally (rarely) used as a communication channel between servers. However, when we put all the four pieces of evidence together, it seems clear that an attack has certainly happened and succeeded and we can say that which machines have been almost certainly compromised.

In handling this incident, the system administrator had to admit that the observations could have multiple interpretations, but he could conclude that one interpretation is most likely to be the case by linking the semantics of multiple observations. For example, although neither observation 3 nor 4 alone can give us high enough confidence to say that the host is definitely compromised, by linking their semantics we can dramatically strengthen our confidence in the assertion, since both 3 and 4 point to the same interpretation. We observed the same pattern in many other incidents we learned from interviewing system administrators. It appears that even without quantitative measures on uncertainty, the semantic links among possible evidence can dramatically increase one's confidence on whether an attack has actually happened and its consequences. As a result, humans can handle the uncertainty pretty well by "connecting the dots" among various pieces of evidence. However, manual analysis alone is not scalable and sustainable in the face of large-scale automated attacks we face today. In this incident, the human system administrator had all the common security tools at his disposal but none of the tools could provide the crucial capability of analysis and the manual analysis took a long time.

1.2 Empirical approach

As a first step, we intend to design tools that help reduce the amount of time that the system administrator has to spend in the process to identify an intrusion. For this, we propose an empirical approach to automate reasoning with uncertainty in intrusion analysis. We design a reasoning model where human knowledge used to draw conclusions about uncertain events can be codified and applied mechanically to future incidents. Although reasoning about intrusions on different incidents can vary significantly, the basic principles are quite consistent. We design a reasoning model that captures the essence of the generic reasoning rationale, not specific features of any particular incident. The model provides a language whereby human experts can share knowledge useful for intrusion analysis in a machine-readable format, and an automated reasoning engine can make use of the knowledge, significantly expanding a human's capability. Both the model and the reasoning engine are designed empirically through studying real security incidents.

This is certainly not the first attempt at automating reasoning about intrusions. Past work has applied rule-based systems to correlate audit logs and find out attacks [1, 2]. There is also a large amount of work on IDS alert correlation [3–8]. These previous approaches do not address the uncertainty problem explicitly, *i.e.*, the reasoning systems do not model when and how confidence levels on assertions can be strengthened in the correlation process[4]. An explicit model for uncertainty in reasoning is crucial to making alert correlation tools useful in practice. Zhai, *et al.* [9] has pioneer work in this area by combining alert-correlation and Bayesian-Network techniques to reason about complementary intrusion evidence from both IDS alerts and system monitoring logs so that high-confidence traces can be distinguished from ones that are less certain. Recent years have also seen the application of quantitative mathematical methods such as Bayesian Network [10, 11] and Dempster Shafer theory [12,

[4]The DIDS (Distributed Intrusion Detection System) project [2] uses a *Rule Value* (RV) to represent the confidence that a rule is useful in detecting intrusions. The RV is trained and adjusted through feedback to the expert system. While the RV could be thought of as a measure of uncertainty, its use does not lead to the reduction of uncertainty through connecting the dots in observations. Thus it is more a heuristic-based optimization than an explicit model of how uncertainty changes in reasoning.

13] in intrusion detection [14–16]. We have chosen not to start from those mathematical theories, because to utilize them we need *a priori* the logical structures among the various observations and hypotheses. In Bayesian Network, the logical structure is the graphical model that encodes the causality relation among assertions; in Dempster-Shafer theory, the logical structure is the "compatibility relation" among assertions. Such mathematical theories work well when a "good structure" is given and their power lies in the capability of combining numerical measures (probability or belief functions) over the structure in a systematic way. For intrusion analysis, identifying the structure of attacks is the biggest challenge itself, which is intermingled with the challenge caused by uncertainty. Thus, it is not obvious how the aforementioned mathematical theories can be directly applied to yield practical tools. Instead of starting from these mathematical theories, we start from true-life experiences like the one described above, and design a model that systematically "simulates" what a human can do manually, formulating such empirical experiences such that the model can be applied mechanically to future incidents. We believe this empirical, bottom-up approach is an important first step in understanding the nature of reasoning about uncertainty in cyber security, and gaining experience that may make some quantitative approaches viable in the future.

1.3 Contributions

In this thesis, I present a method and tool for automated intrusion analysis for combining alerts tagged with uncertainty and from disparate sources. The following are the original contributions through this work.

Some materials presented in this thesis were published in the 25th Annual Computer Security Applications Conference (ACSAC) in 2009 [17].

An empirical model for uncertainty We present a model for capturing the meanings of low-level system observation data in terms of high-level conditions of interest to intru-

sion analysis. We adopt an empirical approach where the model and inference process are based on how human administrators reason about attacks in real security incidents. We use qualitative rather than quantitative assessment to capture the uncertainty inherent in such assertions. The qualitative assessment reflects the imprecise nature of the certainty levels' semantics and it also makes it easier to understand by humans, enabling discussion/refinement of the reasoning model in an open forum. Such a model can be linked to existing knowledge bases such as the Snort [5] rule repository. Our model is capable of expressing logical connections among the high-level conditions (also with qualitative uncertainty assessment) so that it can reason about multi-host, multi-stage intrusions with traces spread across various types of system monitoring data.

Reasoning methodology We present a reasoning process that can utilize such a model and existing IDS tools to automatically identify high-confidence attack traces from large diverse sets of system monitoring data not restricted just to IDS alerts. We also present within this model a method for *strengthening* the confidence in an assertion by combining multiple independent pieces of evidence of low or moderate confidence. Our model of high-level conditions consists of generic predicates such as "compromised," "exploit sent," *etc.* that are independent of the scenario at hand. What can change from one scenario to another are the instantiation of the predicates and the certainty tags associated with them as the scenario events are processed and the paths that the reasoning process takes through these conditions. We believe that human administrators similarly have a small set of "target" conditions in mind when they process intrusion data and there is value in capturing those target conditions directly in an automated reasoning process. We implemented a prototype of a reasoning engine using the true-life case study as a guide and showed that the tools reasoning tracked the system administrator's reasoning process and achieved the same set of high-level conclusions with high confidence.

[5] Snort is an open source network intrusion detection system. http://www.snort.org

A Snort add-on based on our model and method We automatically generated the significant part of our reasoning model from the `classtype`, `impact`, and `ease of exploit` fields associated with Snort rule descriptions and show that the knowledge base needed by our reasoning engine can be readily created if security monitoring tools use our model language, instead of natural language, to describe the potential meanings of various types of security alerts. Based on this automatically-derived knowledge base and the prototype implementation of our reasoning engine, we provide a Snort add-on, called SnIPS [6], that analyzes millions of Snort alerts reported from an enterprise network and only report those with high-confidence evidence associated with them.

Evaluation of the methodology We applied the SnIPS tool to two third-party datasets, and computer science department network at Kansas State University. The core reasoning engine and model was kept unchanged in the evaluation. We found remarkably that our tool discovered interesting scenarios from the three data sets, even though our core reasoning model was developed from a very simple and completely different incident. The application of SnIPS also results in dramatic reduction in the amount of data a system administrator needs to analyze. The false positives from the analysis helped us identify imprecision from the automatically generated Snort knowledge base and some subtle but minor gaps in the core model as well. This indicates that such an empirical approach could produce a shared knowledge base that can be constantly refined among security practitioners to yield agile and accurate tools.

Rest of the thesis is organized as follows: chapter 2 provides the background information on intrusion detection system and discusses related work in the areas of intrusion detection system and reasoning with uncertainty. In chapter 3, we discuss the reasoning model developed based on the true life incident. Chapter 4 deals with our reasoning framework and proof strengthening technique. We discuss the automated model building for Snort in chapter 5 and in chapter 6 we show the experiments done to evaluate the empirical model

[6] http://cis.ksu.edu/~xou/argus/software/snips

developed. In chapter 7 we conclude with a summarization of our work along with future direction of research.

Chapter 2

Background and related work

2.1 Intrusion detection system

Intrusion can be defined as "an unwanted and unauthorized intentional access of computer and network resources" [18]. Intrusion Detection Systems (IDS), are systems used to identify such intrusive activity in computers and network services. One can compare IDS to a car alarm or a burglar alarm installed to detect theft or unwanted access to the property.

2.1.1 Types of IDS

IDS has been around for a couple of decades now. IDS can be software-based or a combination of software and hardware in a stand-alone system. Current IDSes are specialized and developed based on the services it is monitoring. They can be roughly classified into the following categories:

Network-based

A network-based IDS, or NIDS, monitors a network segment for intrusive activity. Typically they are installed in a network tap or span port, a location in the network where it can monitor the entire network traffic. Also, the network interface card (NIC) should support *promiscuous mode*[1]. NIDS performs protocol analysis or inspects the payload of the network traffic to identify malicious behavior.

[1] Promiscuous mode allows NIC to view the traffic not sent to its MAC address

Host-based

A host-based IDS, or HIDS, monitors only the host it resides in. Most HIDS identifies malicious behavior by analyzing system call, system log, file system modification, *etc*.

Application-based

An application-based IDS (AIDS) protects only the application it monitors. Since AIDS are tailored to a particular application, *e.g.* Apache web server, they usually are more accurate than the generic IDSs previously discussed. They are similar to HIDS or NIDS, but with more specific rules or signatures for the application.

2.1.2 IDS detection types

In order to identify an intrusion, IDS system has to understand an intrusion and differentiate it from a benign activity. To do this it can be either provided with the information about the intrusion and asked to report such events, or provide with benign events and report all events that are not benign. The detection type can be broadly categorized as signature-based detection and anomaly-based detection.

Signature-detection

Exploits, viruses, trojans and other malicious programs exhibit a pattern in their behavior or payload. For example, buffer over flow exploit might contain NO OP code [2]. In signature detection, we search for these patterns of known attacks. IDS uses these patterns (also known as rules or signatures) and compare them against the network traffic or log files. When a pattern is matched it is reported as an alert. Signature based IDS may fail to detect novel attacks as the pattern of such attacks may be unknown. Even with this drawback signature-based IDSes are popular compared to anomaly-based IDS discussed in the next subsection.

[2] NOOP (short for No Operation) is an assembly language instruction, sequence of programming language statements, or computer protocol command that effectively does nothing at all (Wikipedia).

Anomaly-detection

In anomaly detection, a model is developed by training it with information about benign activity. Once trained the model will be able to identify benign activities and anything that is unidentifiable is flagged as potentially malicious.

Even though using anomaly detection can identify novel attacks, this approach has a very high false positive rate making it less popular than signature-based IDS. Also, training an IDS for normal behavior is very difficult in practice.

IDS response type

On seeing an intrusion, IDS can be instructed to perform a predefined set of actions. A *passive response* system will issue an alert and/or log entries in response to the alert. An *active response* system may take actions such as drop the packet, send reset packet to the connection, add IP address to the block list, and so on.

2.1.3 Problems in IDS

False positives IDS alerts that are triggered on benign activity where no intrusion or attack is underway are called false positives. A false positive can be issued because the benign activity matched an attack signature.

False negatives When an IDS fails to detect an actual intrusion since it didn't have the rules to match the attack, it is called a false negative.

An ideal IDS should have very low false positive and false negative rate. But in reality, this is hardly achievable because there exists a correlation between false positive and false negatives in an IDS. Since undetected attacks are more detrimental than a false alarm, current IDSes try to have a low false negative rate at the cost of having false positives. In order to have low false negatives, the signatures or anomaly-model are generic in nature, which results in reporting more benign activities as attacks.

Because of the above mentioned trade off, current IDS suffers from high false positive rates. In an enterprise network comprised of many hosts, IDS sensors create thousands of alerts each day. With limited human resources, it is very difficult to analyze the low-level alerts and identify an actual attack. Moreover, an IDS issues an alert on the attempts made by an attacker to compromise a network element. This might result in generating a huge amount of alerts if the attacker is naive and trying many different attacks. While it is important to capture such attempts, security officers and system administrators are more interested in identifying potentially successful attacks as there may be an overwhelming number of failed attempts.

2.2 Related work

2.2.1 Reasoning under uncertainty

Uncertainty in data, even specifically in the context of security analysis, is a vast and fertile topic. Probabilistic reasoning appears to be a natural candidate for such problems and there have been several attempts along this direction [9, 14, 16]. A fundamental challenge in applying these techniques is how to obtain the logical structure needed for combining the probability measures. Moreover, these techniques require as inputs statistical parameters in terms of probability distributions of related events, conditional probability tables, *etc.* that have proven very hard to estimate or learn from real-life data because of overwhelming background noise. For security analysis, it is nearly impossible to obtain the ground truth in real traces and public data sets and it is hard if not impossible to realistically simulate attack scenarios. Consequently calibrating analysis techniques with metrics such as false positive/negative ratios is a huge challenge (See [8] for details) and intrusion analysis remains a manual and intuitive art, which has inspired us to formulate a logic that approximates human reasoning that works with a qualitative assessment on a few confidence levels which are relatively easy to understand. We acknowledge that this formulation not only hides the lack of knowledge of base probabilities but also reflects the great deal of ambiguity that

exists in intrusion analysis of real data. We hope that by creating an option to specify the confidence level explicitly and by providing general practical tools to manipulate these uncertain pieces of knowledge, we can bypass some of these fundamental problems and gain experience and insight that may make some statistical approaches viable in the future.

Friedman and Halpern used a new approach to modeling uncertainty based on "plausibility measures" [19]. Our levels of uncertainty are inspired by similar considerations but we do not use their precise definition because the meaning of negation is not yet clear in our context. Similarly, a formal description of our logic as a modal logic is the subject of future investigation. Along similar lines, the Dempster-Shafer (DS) theory [12, 20] has been used effectively in many kinds of evidential analysis. This theory allows us to construct belief functions and plausibility measures for beliefs by combining sets of events within certain constraints. There have been highly customized applications of the DS theory to application domains in AI and expert systems, where numerical quantification is not possible (see, for example, [21, 22]). These logics may be useful in building a theoretical underpinning for our system; we have only done a partial investigation in this direction as our focus has been on building usable empirical tools.

2.2.2 Alert reduction techniques

The literature on IDS alert reduction techniques is vast and they can be roughly categorized into alert correlation [1, 3–9, 23–27], alert clustering [28, 29] and alert verification [30–34]. The goal of these works are to eliminate redundancy, remove failed attempts and provide a logical correlation of the alerts so that a system administrator can easily comprehend the huge volume of alerts.

Alert correlation

Most of IDS alert correlation works model IDS-specific states using pre- and post-conditions that drive a correlation model and rely on the existence of a sparse (nearly deterministic) mapping from alerts to their pre-/post-conditions. It appears to be difficult to model in this

manner ubiquitous alerts such as “abnormally high network traffic” that could be indicative of any of a wide variety of possible conditions. Our observation correspondence model explained in chapter 3.2 assigns a *direct* meaning to an observation and our internal model allows such meanings to be flexibly linked together based on their semantics. Such flexibility is important when the evidence is tenuous and subject to multiple interpretations.

Alert clustering

In alert clustering, a set of similar alerts are clustered and represented as a single alert. For example, alerts having similar source and destination IP addresses, alert types but different time stamps are clustered and represented as a single alert. The time stamp of the new predicate will be a time range having the earliest and latest time stamp from the set of similar alerts. This step helps in reducing the redundancy in alerts generated by IDS. Julish [28] proposed a data mining based alert clustering technique to discover root causes. According to this work, a few dozen of root causes generally account for more than 90% of the alerts. By identifying the root causes it is possible to filter alerts caused by benign root cause and thus reduce the future alert load. Clustering is a good technique to reduce the redundant information in the alerts. A limitation of alert clustering is that they do not provide a high-level situational awareness of the network and causal relation between the clustered events.

To handle the possibly enormous inputs but at the same time keep the linkage between low-level traces to high-level meanings, our pre-processing step includes data reduction based on clustering and simple correlation of local observations. Much of the previous work in IDS has addressed this important problem [5, 8] (including the “layered approach” of Martignoni *et al.* [35]). We intend to use all applicable tools and approaches from these works.

Other kinds of intrusion analysis such as M2D2 [23] require classifying incoming information into various categories with mathematical notations which is a challenge for the typical users; an important principle of our model is to represent knowledge as simple “assertions” that can be easily translated into natural language.

Alert verification

Kruegel *et al.* [34] introduced the idea of alert verification, a method for filtering failed attempts from IDS alerts. Alert verification use active and passive techniques to verify if conditions for success are favorable or observe the effect of an exploit to determine success. For example, the system verifies if the services exist on the host the exploit is attempted at and if the version of the services is vulnerable to this attack. Host can be probed to gather such information in real time or gather information periodically (once a week or when the system is updated) to avoid disrupting the services at every attempt. Observing the effects of an attack can be useful too. If an attack is known to disrupt a service, probing to find if the service is up can help identify if the attempt was a failure or successful one. Atlantides [30] by Bolzoni *et al.*, proposed an architecture design of alert verification for network intrusion detection system.

2.2.3 Attack graph

Another related research area is attack graphs [36–41]. While primarily used in vulnerability assessment and security hardening, attack graphs can identify multi-stage attack possibilities. Lingyu Wang *et al.* [41] used attack graphs for correlating, hypothesizing and predicting alert correlation. They also proposed queue graph, an approach to retain the latest alert of each type. The correlation between old and new alert of each type is implicitly represented using the temporal order between alerts. This approach is efficient in handling a large amount of alerts if they are of the same type. Recently, Zhang [42] proposed a new representation of attack graphs which can better handle intrusion detection. However, we have not seen systematic treatment of uncertainty in intrusion analysis using attack graph techniques.

2.2.4 Other closely related work

BotHunter [43] is an application for identifying Bot machines by correlating Snort alerts with a number of other system-monitoring events. The notions of "confidence score" and "evidence threshold" are introduced to capture the uncertainty in the correlation process and specific processes are designed for the purpose of Bot detection. The goal of our work is to provide a simple but more general model for intrusion analysis.

Hollebeek and Waltzman [44] proposed a notion of "suspicion" in modeling uncertainty in intrusion analysis. It appears that the approach does not further differentiate the various meanings that could be associated with a "suspicious" event. The system relies on a "deductive graph" and a "suspicion graph" to propagate certainty levels within the context of the system under concern. But with just a single notion of "suspicion" for each event, it is not clear how to build or interpret the meanings of such graphs in a systematic and consistent manner.

Scyllarus [45] is another closely related work. Their goal is to perform IDS fusion using Bayes nets and System-Z+ [46] with qualitative probability. This system use the notion of "possible" and "likely" to determine if an event reported by IDS is a false positive. The primary goal of our work is to provide a framework for specifying varying degree of confidence level and use proof strengthening technique to elevate the confidence level.

Chapter 3

Reasoning model

In this chapter we describe our reasoning model empirically developed from the true-life incident. *Observation correspondence* relation maps low-level observations to high-level conditions, *internal model* captures the relationships among high-level conditions; and *modes* captures the degree of uncertainty. Our motivation behind the reasoning model was the true-life incident (case study) discussed in section 1.1.1. We use this case study as an example to understand the system administrator's analytical state in the course of investigation, identify the rationale behind his decision at various stages, and design a logic that is similar to the human reasoning process.

3.1 Modeling uncertainty

While the goal of intrusion analysis is detection of events at a high-level of abstraction (*e.g.*, a machine has been compromised and has been used to compromise others), tools today operate with any known accuracy only at low levels of abstraction (*e.g.*, network packets, server logs, *etc.*). Uncertainty arises from this semantic gap. For example, a packet pattern (a "signature") could be associated mostly with attacks but on occasion with legitimate use as well. Furthermore, it may not tell us whether the attack succeeded. A key step in tackling the uncertainty challenge is to develop a model that can link multiple low-level observations to the conditions under concern at the high level and simultaneously allow us to specify our confidence in the assertions.

Confidence level	Mode	
low	possible	p
medium	likely	l
high	certain	c

Table 3.1: *Uncertainty modes*

For example, an `abnormal high network traffic` (low-level observation) could mean that `an attacker is performing some network activity` (high-level condition). Similarly, the `netflow dump showing a host communicating with known BotNet controllers` (low-level observation) indicates that it is likely `an attacker has compromised the host` (high-level condition). All these assertions are associated with varying degrees of uncertainty. For example, compared with the anomalous high network traffic, a netflow filter that shows communication with known BotNet controllers is a more confident assertion on attacker activity. It is crucial that the reasoning model be capable of expressing such differences and it is desirable that this be done without relying on probability parameters that are difficult to obtain and hard to justify for humans.

The reasoning model uses three modes p, l, c, standing for "possible, likely, certain" to express low, moderate, and high confidence levels (See table 3.1). Even though one could think of certainty level as a continuous quantity ranging from completely unknown to completely certain, in practice, human system administrators only deal with a few confidence levels that roughly correspond to the ones defined here.These words are also used routinely in natural-language description of security knowledge bases such as the Snort rule repository. These uncertainty levels are assigned by humans and apart from the obvious ordering ($p < l < c$) there is no ascribing of probability ranges to each level.

With this qualitative notion of uncertainty, two types of logical assertions are used in the reasoning model: *observation correspondence* which maps low-level observations to high-level conditions, and *internal model* which captures relationships among high-level conditions (also called *internal conditions*). Correspondingly, $obs(O)$ is used to denote a fact about observation O, and $int(F)$ to denote an internal condition F. For example,

$$A_1: \quad obs(anomalyHighTraffic) \xrightarrow{p} int(attackerNetActivity)$$
$$A_2: \quad obs(netflowBlackListFilter(H, BlackListedIP)) \xrightarrow{l} int(attackerNetActivity)$$
$$A_3: \quad obs(netflowBlackListFilter(H, BlackListedIP)) \xrightarrow{l} int(compromised(H))$$
$$A_4: \quad obs(memoryDumpMaliciousCode(H)) \xrightarrow{l} int(compromised(H))$$
$$A_5: \quad obs(memoryDumpIRCSocket(H_1, H_2)) \xrightarrow{l} int(exchangeCtlMessage(H_1, H_2))$$

Figure 3.1: *Observation correspondence*

$obs(netflowBlackListFilter(ip_1, ip_2))$ is an observation from the netflow blacklist filter that "machine ip_1 is communicating with a known blacklisted (and hence likely malicious) host ip_2", whereas $int(compromised(ip_1))$ is an internal condition that "ip_1 is compromised."

3.2 Observation correspondence

Figure 3.1 shows the observation correspondence relation for the observations in the true-life incident described in section 1.1.1 along with the corresponding uncertainty modes. In A_1 an abnormal high network traffic $obs(anomalyHighTraffic)$ is mapped to $int(attackerNetActivity)$, meaning an attacker is performing some network activity. This is a low-confidence judgment because a high network activity could be caused by many benign events such as a authorized user downloading movies; thus the mode is p. Intuitively, the p mode means there are other equally possible interpretations for the same observation. A_2 and A_3 give the meaning to an alert identified in netflow analysis. There are a number of filtering tools that can search for potential malicious patterns in a netflow dump such as "capture daemon" and "flow-nfilter." These rules deal with one filter that identifies communication with known malicious IP addresses. Since any such activity is a strong indication of attacker activity and compromise of the machine involved, the modality of the two rules is l. There are still other possibilities, *e.g.* the communication could be issued by a legitimate user who wants to find out something about the malicious IP address. But the likelihood of that is

significantly lower than what is represented by the right-hand side of the two rules. It is legitimate to have multiple observation correspondence assertions for the same observation: they may represent different aspects or possibilities of an observation. A_4 says if memory dump on machine H identifies malicious code then H is likely to be compromised. A_5 says if the memory dump identifies open IRC sockets between machine H_1 and H_2 then it is likely that the IRC channel was used to exchange control messages between BotNet members.

We recognize that these observation correspondence assertions are subjective. Quantifying the results of intrusion sensing in a robust manner has remained a hard problem for a variety of reasons [47]. Our goal is to create a flexible and lightweight framework wherein a system administrator can feed in these beliefs of certainty and see what consequences arise. For example, a system administrator may think the mode of A_4 ought to be c, which would be acceptable. One advantage of such a logic is that it facilitates discussion and sharing of security knowledge. Given the large base of similar deployed infrastructures, shared experiences from a large community can likely help tune the modes in those assertions. We envision a rule repository model like that for Snort, where a community of participants contribute and agree upon a set of rules in an open language. Currently there are only coarse-grained classification and some natural-language explanations for the meanings behind each Snort alert. In chapter 5, we show how a small number of internal-model predicates can give meanings to the vast majority of Snort alerts and that the observation correspondence relation can actually be automatically generated from a Snort rules class type and the "impact" and "ease of attack" fields in the rules natural-language description. If the Snort rule writers had a standard language for such information they would be able to readily provide the observation correspondence assertions for Snort alerts.

3.3 Internal model

The reasoning model should also express the logical relations among the various high-level conditions so that such knowledge can be mapped to correlate low-level events. For example,

$$I_{1f}: \quad int(compromised(H_1)) \xrightarrow{f,\,p} int(probeOtherMachine(H_1,\ H_2))$$

$$I_{1b}: \quad int(probeOtherMachine(H_1,\ H_2)) \xrightarrow{b,\,c} int(compromised(H_1))$$

$$I_{2f}: \quad int(compromised(H_1)) \xrightarrow{f,\,p} int(sendExploit(H_1,\ H_2))$$

$$I_{2b}: \quad int(sendExploit(H_1,\ H_2)) \xrightarrow{b,\,c} int(compromised(H_1))$$

$$I_{3f}: \quad int(sendExploit(H_1,\ H_2)) \xrightarrow{f,\,l} int(compromised(H_2))$$

$$I_{3b}: \quad int(compromised(H_2)) \xrightarrow{b,\,p} int(sendExploit(H_1,\ H_2))$$

$$I_{4f}: \quad int(compromised(H_1)),\ int(compromised(H_2)) \xrightarrow{f,\,p} int(exchangeCtlMessage(H_1,\ H_2))$$

$$I_{4b_1}: \quad int(exchangeCtlMessage(H_1,\ H_2)) \xrightarrow{b,\,c} int(compromised(H_1))$$

$$I_{4b_2}: \quad int(exchangeCtlMessage(H_1,\ H_2)) \xrightarrow{b,\,c} int(compromised(H_2))$$

Figure 3.2: *Internal model*

the model should include knowledge such as "after an attacker has compromised a machine, he/she may perform some network activity from the machine." This is a generic action common to many attack scenarios. This knowledge can reveal potential hidden correlations between low-level observations, (*e.g.*, high network traffic and netflow filtering result). In the absence of any context to guide us, a traffic spike could be due to any of a number of things but in the context of a likely compromise, the parameters of the traffic burst become important — if the traffic emanated from the likely compromised machine it can be assigned a different meaning than if it did not.

Figure 3.2 shows the internal model we developed from studying the real-life incident. We use $C_l \xrightarrow{d,m} C_r$ to represent the inference rules for the internal conditions, namely condition C_l can infer condition C_r. There are two modality operators, d and m, associated with a rule. Like in observation correspondence, the m mode specifies the confidence in the inference and takes values from $\{p, l, c\}$. The d mode indicates the direction of the inference and could be either f (forward) or b (backward). In forward inference, C_r is caused by C_l, thus the arrow must be aligned with time, *i.e.* C_r shall happen after

C_l. This can specify knowledge for reasoning what could happen after a known condition becomes true, *e.g.* `after an attacker sends an exploit to a machine, he will likely compromise the machine` (I_{3f}). In the backward inference, we reason what could have happened before to cause a known condition, and thus the direction of inference is opposite to time. Example: `if a malicious probe is sent from a machine, then an attacker must have already compromised the machine` (I_{1b}). As another example, the forward inference rule I_{1f} specifies that "if an attacker has compromised machine H_1, he can perform a malicious probe from H_1 to another machine H_2." This inference has a low certainty: the attacker may or may not choose to probe another machine after compromising one. Thus the rule is qualified by the p mode. I_{4f} is the only rule in this model that has two facts on the left-hand side. As in typical logic-programming languages, the comma represents the AND logical relation.

The internal model discussed above provides a flexible framework for representing high-level abstract conditions and the relationship among various high-level conditions. Adding a new security incident will require a new observation correspondence to map the incident to any of the internal conditions developed from our case study. In chapter 5, we show our automated approach to build observation correspondence for Snort IDS. Similarly, adding new additional internal conditions will require minimal effort. The new internal conditions along with the relationship with other predicates must be specified in the internal condition. Adding new internal conditions or new observation correspondence relation will not require any modification to the reasoning engine discussed in chapter 4.

Chapter 4

Reasoning methodology

The reasoning model described in chapter 3 is analogous to human thinking – observations are reflected as beliefs with varying strengths and the beliefs are related to one another with varying strengths. This chapter will introduce the reasoning process to use such a model to "simulate" human thinking such that an automated inference process can allow us to combine observations to construct sophisticated attack conclusions along with a semi-quantitative measure of our confidence. This inference process is capable of deriving from a large number of possibilities high-confidence beliefs corroborated by a number of complementary evidence pieces logically linked together.

4.1 Reasoning framework

Figure 4.1 presents the architecture of our reasoning system. The two modules of the reasoning model — observation correspondence (described in chapter 3.2) and internal model (described in chapter 3.3) are input to the reasoning engine. Both modules are specified in Datalog [48], a simple logic-programming language. The raw observations are pre-processed and the distilled results are converted to Datalog tuples as input to the reasoning system. The reasoning engine is itself implemented in Prolog. An important feature of our design is that every component of the system is specified declaratively, which has the useful property that once all specifications are loaded into the Prolog system, a simple Prolog query will automatically and efficiently search for true answers based on the logic specification. For

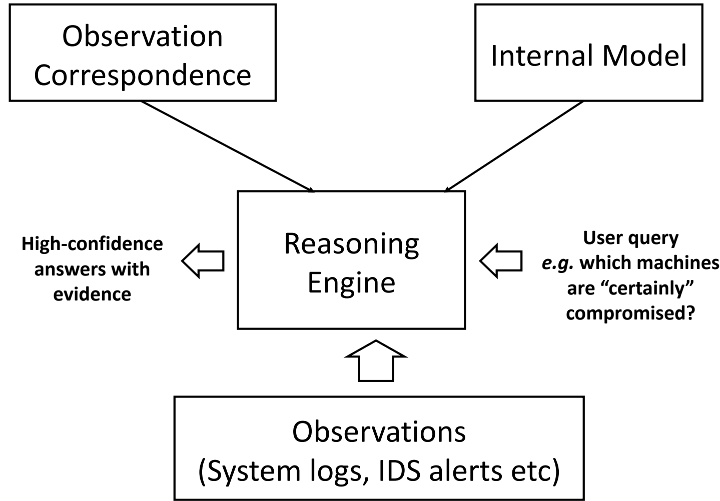

Figure 4.1: *System architecture*

example, a user can ask a question "which machines are certainly compromised?" in the form of a simple Prolog query. Our reasoning engine will then give the answer along with the evidence in the form of logical proofs.

4.2 Application of inference rules

All the rules in the observation correspondence and internal model can be viewed as inference rules. The reasoning engine applies those rules on the input Datalog tuples to derive assertions with various levels of certainty. The certainty of the derived fact is the lowest certainty of the facts and rules used to derive it.

We use $int(F, m) \Leftarrow Pf$ to represent that "the internal fact F is true with modality m", and Pf is the proof that shows the logical derivation steps arriving at the conclusion. From an observation one can derive an internal belief with some degree of certainty, based on the observation correspondence relation. As an example, the open IRC Socket identified through memory dump in the incident (Observation 4) described in section 1.1.1 will be an

input to our system: *obs*(*memoryDumpIRCSocket(172.16.9.20,172.16.9.1)*). Together with observation correspondence A_5 shown in figure 3.1 the rule will derive:

$$\begin{aligned}&int(\textit{exchangeCtlMessage('172.16.9.20','172.16.9.1')}, l) \Leftarrow \\ &\quad obs(\textit{memoryDumpIRCSocket('172.16.9.20','172.16.9.1')})\end{aligned}$$

There can be more than one derivation if there are multiple observation correspondences. This is the first step that happens internally in the reasoning engine. All the observations are mapped into an internal condition.

Using the internal-model rules shown in figure 3.2, further internal conditions are derived. The fact *int*(*exchangeCtlMessage(172.16.9.20,172.16.9.1)*, l) derived above, together with the internal-model rule I_{4b_1}, would yield the following derivation trace.

$$\begin{aligned}&int(\textit{compromised(172.16.9.20)}, l) \Leftarrow \\ &\quad int(\textit{exchangeCtlMessage(172.16.9.20,172.16.9.1)}, l) \Leftarrow \\ &\qquad obs(\textit{memoryDumpIRCSocket(172.16.9.20, 172.16.9.1)})\end{aligned}$$

At any point during the derivation, a newly inferred predicate will take the lower of two modes from the internal rule or derivation chain. For example, the certainty mode for I_{4b_1} is c, joined with the l mode in *int(exchangeCtlMessage(172.16.9.20, 172.16.9.1), l)*, we get l as the mode for the resulting fact *int*(*compromised(172.16.9.20)*, l).

One could argue that the certainty of the derived fact should be lower than that of the weakest fact in the derivation chain, especially when the derivation chain is long. However, given the observation that most enterprise network intrusions are carried out in just a few steps, we do not expect the derivation chains to be long in practice. Since the certainty modes only represent a rough guess, accounting for certainty level decay along short derivation paths is unlikely to be significantly valuable.

4.3 Proof strengthening

In the previous section, we discussed the rules used to derive internal conditions. All the derived internal conditions will have a mode equal or lower than the mode of the observation.

We use the proof strengthening technique to elevate the confidence level by combining multiple observations inferring the same internal condition. This step results in creating high-confidence proof traces which is later presented to a system administrator for further analysis.

The key purpose of reasoning about uncertainty is to derive high-confidence facts from low-confidence ones. In the true-life incident, the system administrator strengthened his belief that the Trend Micro server was compromised by combining three different pieces of evidence: netflow filter result showing communication with a blacklisted IP address, memory dump result showing likely malicious code modules, and memory dump result showing open IRC sockets with other Trend Micro servers. These three pieces of evidence are *independent* — they are rooted on observations at different aspects of the system, and yet they are *logically connected* — all of them indicate that the Trend Micro server is likely compromised. Thus, they altogether can strengthen our belief in the fact that the server is compromised.

We generalize this reasoning process in the following proof strengthening rule.

$$\frac{\mathit{int}(F, m_1) \Leftarrow \mathit{Pf}_1 \quad \mathit{int}(F, m_2) \Leftarrow \mathit{Pf}_2 \quad \mathit{Pf}_1 \parallel \mathit{Pf}_2}{\mathit{int}(F, \mathit{strengthen}(m_1, m_2)) \Leftarrow \mathit{strengthenedPf}\,(\mathit{Pf}_1, \mathit{Pf}_2)}$$

The $\parallel$ relation indicates that two proofs are independent, meaning they are based on disjoint sets of observations and internal conditions. This deduction rule states that if we have two reasoning paths to a fact with some confidence levels and if the two paths are based on independent observations and deductions, then the confidence level of the fact can be strengthened.

According to the deduction rule, two independent proofs, Pf_1 and Pf_2 leading to a internal fact F, with modes m_1 and m_2 respectively can be strengthened and a new strengthened predicate with an elevated mode is created. Mode elevation using *strengthen* function is defined below.

$$strengthen(l,l) = strengthen(l,p) = strengthen(p,l) = c$$

In simple words, two independent proofs can strengthen to "certain" if at least one of them can yield a "likely" mode. There is no definition for *strengthen* when both parameters are p or at least one of them is c. Since the p mode represents very low confidence we do not allow strengthening from just possible facts. There is no need to strengthen a fact if it has already reached the "certain" level.

We emphasize that the strengthening rules are defined here through our empirical study on real-life security incidents and these strengthening conditions do reflect the mental process a human system administrator goes through when catching real attacks.

4.4 Implementation

We use the XSB [49] system to evaluate the Prolog reasoning engine. We also implemented a simple proof-generator so that whenever a fact is derived, the proof trace for that fact can also be obtained simultaneously. We applied the reasoning system and the model described in 3.2 and 3.3 on the input for our case study. The result is shown in Figure 4.2 (IP addresses are sanitized.). The user enters the query `show_trace(int(compromised(H),c))`, to find all the provable facts in the form of `compromised(H)` with "certain" mode. This is essentially asking the question "which machines are certainly compromised". The reasoning engine prints out a derivation trace for `172.16.9.20`, the IP address for the compromised Trend Micro server first identified by the system administrator. It is clear that the derivation trace exactly matches the reasoning process the human system administrator used to identify the compromised server — the confidence level is strengthened from concordant evidence emanated from netflow dump and memory dump.

The execution time of the reasoning system is affected by both the internal model and the number of input observations. We expect the internal model developed by a user community contains a constant (and relatively small) number of internal predicates and assertions

```
| ?- show_trace(int(compromised(H),c)).

int(compromised('172.16.9.20'),c) strengthen
    int(compromised('172.16.9.20'),l) I_4b1
       int(exchangeCtlMessage('172.16.9.20',  '172.16.9.1'),l) A_5
           obs(memoryDumpIRCSocket('172.16.9.20',  '172.16.9.1'))
    int(compromised('172.16.9.20'),l) A_4
        obs(memoryDumpMaliciousCode('172.16.9.20'))
    int(compromised('172.16.9.20'),l) A_3
        obs(netflowBlackListFilter('172.16.9.20', '129.7.10.5'))
```

Figure 4.2: *Result of applying the reasoning system on the case study*

that capture the semantics of security conditions of interest. The simple reasoning rules described in Section 4.2 can be viewed as evaluating a fixed Datalog program against input tuples (observations) in XSB whose complexity is dominated by the maximum number of instantiations of subgoals [38]. Since our model can infer at most as many internal conditions as there are input observations (in an asymptotic sense), the complexity of applying the simple reasoning rules will be linear in the number of input tuples. The proof strengthening rule described in Section 4.3 needs to conduct pair-wise comparison among derivation paths leading to a fact and hence can have quadratic complexity. Consequently, a rough estimate of the reasoning time complexity is quadratic in the number of input tuples. This was empirically verified by our experiments described in chapter 6.

During the evaluation of this empirical model, we recognized a need for pre-processing and using time information in the reasoning process. The data given as input to the reasoning system contains a lot of redundant information which some times overwhelms the system, in the pre-processing step we reduce the redundancy in the input data. We use the time information as we have both forward and backward reasoning and they differ in the time relationship in the derived facts. We discuss the pre-processing and time handling in the following sections.

$$\left.\begin{array}{c} int(probe(ext_1, webServer), c, T_1) \\ int(probe(ext_2, webServer), c, T_2) \\ ... \\ ... \\ int(probe(ext_n, webServer), c, T_n) \end{array}\right\} int(probe(external, webServer), c, range(T_1, T_n))$$

Figure 4.3: *Preprocessing: Summarization*

4.5 Pre-processing

The pre-processing step is performed to compact the information entering the reasoning engine. We apply a data abstraction technique by grouping a set of similar "internal conditions" into a single "summarized internal condition".

The current implementation of summarization is done on time stamp and IP addresses. This step is done after the observations are mapped to their internal conditions. If a set of internal conditions differ only by timestamp, we merge them into a single summarized internal condition with a time range between the earliest and latest timestamp in the set. Similarly, we also abstract over external IP addresses so that alerts that differ only on the external source or destination addresses are merged and the corresponding IP address is abstracted as "external". Figure 4.3 depicts the process of summarization with an example. In this example, multiple observations map to the internal condition, *probe* with identical mode, c. These observations have *webServer* as the destination IP address and a source IP address from an external network. They may have the same time stamp or temporally close. When we apply the summarization technique, this yields a single predicate having a wild card variable *external* to represent source IP address belonging to external network and a time range having the earliest and latest time in observations.

In the current implementation, we do not summarize on internal IP addresses as this knowledge may be useful in the reasoning process. A mapping between the summarized internal condition and the raw internal conditions/observations identifies the low level facts belonging to the summarized predicates. The summarized tuples are then passed on to the reasoning engine.

4.6 Handling time

Timestamps associated with security monitoring events are important in tracking and diagnosing root causes of problems. When an observation correspondence rule is used, the internal condition on the right hand side simply inherits the time field of the observation on the left. When an internal model is used, the time stamps associated with the derived assertion will be derived from the timestamp on the left hand side and the direction of the rule in a straight-forward manner (*e.g.*, for a forward rule the right hand-side shall happen after the left hand side). Latency in detection and clock skews can also make timestamps imprecise and less useful, which needs to be addressed through techniques such as time windows.

Chapter 5

Automating model building for Snort

In chapters 3 and 4 we empirically developed a model and reasoning engine for intrusion analysis based on the true-life incident described in 1.1.1. To test the model developed from the case study on a publicly available IDS, Snort, we created an Snort add-on, called SnIPS. The purpose of this tool is to help the users to reason about Snort alerts using the techniques discussed in the earlier chapters. Figure 5.1 shows the system architecture of SnIPS. It consist of the reasoning engine and internal model developed from our empirical study, observation correspondence for Snort alerts created automatically from Snort rule repository and tools for translating Snort alerts. In this chapter we briefly introduce Snort IDS and describe our approach to automatically creating a observation correspondence knowledge base for Snort IDS.

5.1 Snort

Snort [50, 51] is an open source, rule-based network intrusion detection system and is widely used for real time network traffic analysis. It combines signature, protocol and anomaly-based inspection for identifying malicious events. Snort is cosmetically similar to tcpdump [52] but is more focused on the security applications of packet sniffing. The major feature of Snort is packet payload inspection. Snort decodes the application layer of the packet and can be given rules to collect traffic that has specific data contained within its application layer. This allows Snort to detect many types of hostile activity, including buffer

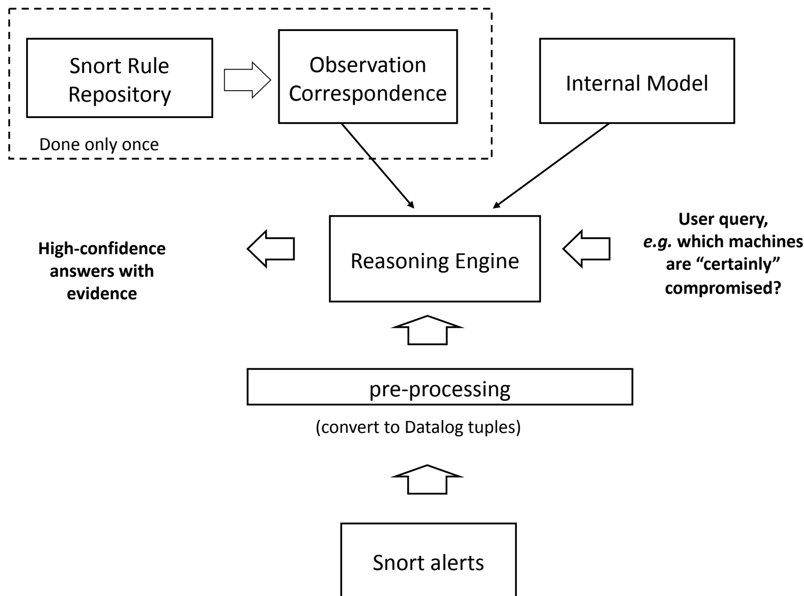

Figure 5.1: *SnIPS system architecture*

overflows, CGI scans, or any other data in the packet payload that can be characterized in a unique detection fingerprint.

5.1.1 Snort architecture

Internally, Snort is made up of a packet sniffer, preprocessor, detection engine and logging module.

Packet sniffer The packet sniffer module uses `libpcap` [52] library for reading the network packets. Typically, Snort listens to a network interface in promiscuous mode. This allows Snort to read all the packets in the network. The packet sniffer module sends the captured packet to the preprocessor module.

```
alert tcp $EXTERNAL_NET any -> $SMTP_SERVERS 25 (msg:"EXPLOIT x86 windows
MailMax overflow"; flow:to_server,established; content:"|EB|E|EB| [|FC|3|C9
B1 82 8B F3 80|+"; reference:bugtraq,2312; reference:cve,1999-0404;
classtype:attempted-admin; sid:310; rev:8;)
```

Figure 5.2: *An example Snort rule*

Preprocessor Snort's preprocessor is used to either examine packets for suspicious activity or modify packets so that the detection engine can properly interpret them. For example, the `frag3` preprocessor defeats the IP fragmentation attacks by reassembling fragmented packets and `ARPspoof` is a preprocessor designed to detect malicious Address Resolution Protocol (ARP) traffic. There are many pre-processor plugins available in Snort that can be added by system administrators.

Detection engine The detection engine is the most important feature of Snort. It receives data from the preprocessor and checks it with a rule set, which contains signatures written by experts for identifying malicious patterns. If a rule is matched, based on the configuration of the rule, the packet can be sent to the logging model, dropped or passed.

Alert/Logging module The logging module receives the alert and the associated rule that triggered the alert. It then writes the alert and rule information to a log file or a database. Snort logging module can be configured to log the alerts into a log file, database, windows event viewer, network connection, *etc.* In most production systems, where the number of alerts generated can be huge, Snort is configured to store the alerts in a database. This makes it convenient to manage the large volume and the ability to query for specific alerts. (Refer to appendix A.3 for schema of the Snort alert database).

5.1.2 Snort signature

A Snort signature consists of a header and rule options. The header section is made up of action (alert, drop, pass or log packet), protocol (IP, ICMP, TCP), source and destination IP address (individual IP, range of IPs, list of IPs, or "any"), port (individual port, range of

ports or "any") and the direction of the packet. The rule options section contains options for matching the payload, state of the connection and meta information about the rule. Figure 5.2 shows an example snort rule for identifying "MailMax overflow exploit" on x86 windows machines. This rule instructs Snort detection engine to issue on alert when both the header and options are matched. The header information is used by the detection engine as a filter. When the header criteria is met, the detection engine will compare the rule options against the packet. In our example, the rule header action is to alert on a `tcp` packet having source IP address matching `$EXTERNAL_NET` (A variable set in Snort configuration file) and `any` source port. The destination IP address should match `$SMTP_SERVERS` and destination port number should be 25 (SMTP). All packets matching this criteria will then be compared with the rule options, and an alert is issued if both criteria are met. Let us look at the options specified in the rule. The `msg` specifies a title for the rule and `flow` option specifies type of tcp connection. The `content` keyword is widely used in Snort rule. It is used for matching a specific pattern in the packet payload. The option `flow` allows the rule to be applied to only certain direction of the traffic flow. The rest of the rule options gives meta information about the rule. `sid` is for uniquely identifying the rule and `rev` gives the number of revisions made to the rule. The `reference` field points to the informations on the web and popular vulnerability data such as *bugtraq, cve, nessus, arachNIDS* and *McAfee.* The keyword `classtype` is a simple classification used by Snort to group rules based on the kind of attack.

5.1.3 Snort alert

A Snort alert provides information to a system administrator about a potentially malicious event. It contains source and destination IP address, type of attack, rule details, and other packet details to help in intrusion analysis. A Snort alert is created when a packet matches one of the Snort rules and it is stored in a log file or a database based on the configuration. Figure 5.3 shows a Snort alert stored in a log file (The number is added manually for ease

```
(1)      [**] [1:2080:8] RPC portmap nlockmgr request TCP [**]
(2)      [Classification: Decode of an RPC Query] [Priority: 2]
(3)      07/08-10:33:07.074650 129.130.10.171:43191 -> 129.130.10.33:111
(4)      TCP TTL:64 TOS:0x0 ID:40304 IpLen:20 DgmLen:204 DF
(5)      ***AP*** Seq: 0xF6970FCF  Ack: 0x976A7839  Win: 0x2E  TcpLen: 32
(6)      TCP Options (3) => NOP NOP TS: 161714848 318558254
(7)      [Xref => http://cgi.nessus.org/plugins/dump.php3?id=10220]
(8)      [Xref => http://cve.mitre.org/cgi-bin/cvename.cgi?name=2000-0508]
(9)      [Xref => http://www.securityfocus.com/bid/1372]
```

Figure 5.3: *An example Snort alert*

of explanation). This alert is issued for an RPC portmap exploit. Line 1 contains the Snort rule ID and title of the alert. Line 2 contains classtype of the rule that created the alert and its priority level. Line 3 has packet information such as time stamp, source/destination IP address and port number. Lines 4-6 contain TCP packet information and lines 7-9 have web address pointing to resources having the vulnerability information.

5.2 Knowledge base creation

To use Snort alerts for reasoning in the system developed from our empirical study, we need the observation correspondence relation for mapping Snort alerts to internal conditions. We automated the knowledge base creation process, since the current Snort system does not come with such mapping and creating the relation manually for approximately 9000 Snort rules[1] is tedious.

From our analysis, we found Snort rules and associated signature documents contained valuable meta information useful for determining the observation correspondence relation. We created a parser to extract the required information from these documents and using this information, we automatically created the observation correspondence relation for a large portion of the Snort rule. We discuss the meta information considered in our automated process below.

[1] In Snort rule set version 2.8

Snort Classtype	**Internal Condition**	**Mode**
attempted admin	compromised	l
attempted user		l
successful admin		c
successful user		c
trojan activity		l
shellcode detect	sendExploit	l
unsuccessful user		p
web application attack		l
web application activity		p
attempted recon	probeOtherMachine	l
successful recon largescale		c
successful recon limited		c
network scan		p
rpc portmap decode		l
icmp event		p
suspicious filename detect	Unknown	Unknown
suspicious login		
system call detect		
unusual client port connection		
misc activity		
not suspicious		
protocol command decode		
string detect		
unknown		
tcp connection		
policy violation		
kickass porn		
attempted dos		
bad unknown		
default login attempt		
denial of service		
misc attack		
non standard protocol		
successful dos		

Table 5.1: *Automatically created obsMap*

5.2.1 Classtype

The `classtype` keyword in a Snort rule is used for categorizing the rules into a more general attack class. For example, Snort rule `1:3156` (ORACLE XDB FTP UNLOCK overflow attempt), can possibly give the attacker administrator privileges on a victim machine. Thus, this rule comes under the class type *attempted-admin*. Snort currently has 30 default class types for a rule writer to classify the nature of the potential security implication. Some of these class types roughly correspond to the internal predicates developed during our empirical study. So, we manually created a mapping between class type and internal predicates as given in table 5.1. Using this table, we infer the observation correspondence for the rule in figure 5.2 as shown below:

```
obsMap(obsRuleId_3614,
        obs(snort('1:310', FromHost, ToHost)),
        int(compromised(FromHost, ToHost)), l).
```

Some of the class types is mapped to *unknown* as the class type information was insufficient for us to determine a relation. For such cases, we try to use the information available in the rule document to find the answer.

5.2.2 Snort rule document

For some cases the class type alone does not provide enough information for determining the observation correspondence relation, so we make use of the information available in the Snort rule documents.

Each Snort rule has a document which provides a brief description in the following sections: summary, impact, affected systems, ease of attack, false positives, false negatives, corrective actions, contributions and additional reference (Appendix A.2 has an example Snort rule document). The information found under "Impact" and "Ease of Attack" in particular are quite useful for inferring the internal predicates. For example, the Snort rule `1:1140` is

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS
(msg:"WEB-MISC guestbook.pl access"; flow:to_server,established;
uricontent: "/guestbook.pl"; nocase; metadata:service http;
classtype:attempted-recon; sid:1140; rev:12;)
```

The natural language description of the two fields ”Impact” and ”Ease of attack” for the above rule is:

```
Impact:
Information gathering and system integrity compromise. Possible
unauthorized administrative access to the server.  Possible
execution of arbitrary code of the attackers choosing in some cases.


Ease of Attack:  Simple. Exploits exist.
```

Using the keywords “possible administrative access” and “possible execution of arbitrary code” in the “Impact” section and keyword “exploits exist” in the “Ease of Attack” field, our automated program infers the following observation correspondence predicate:

```
obsMap(obsRuleId_3615,
       obs(snort('1:1140', FromHost, ToHost)),
       int(compromised(ToHost)), p).
```

In general, we found that these two fields are often composed of a set of fixed keywords. Examples are “possible unauthorized administrative access”, “possible execution of arbitrary code”, “exploits exist”, and so on. These phrases often indicate what internal predicate and certainty mode can be assigned to the alert. Our automated program searches for those keywords in the Snort rule document. Based on the combination of keywords contained in a Snort rule description, a simple heuristic algorithm infers the internal condition and the certainty mode for alerts generated by the Snort rule.

```
obs(100, snort('1:1140', '75.149.65.202', '64.40.147.175', 1254954059)).
obs(101, snort('1:1140', '75.149.65.202', '64.40.147.175', 1254954059)).
obs(102, snort('1:998', '10.23.3.10', '64.40.147.180', 1254954060)).
```

Figure 5.4: *Example translated Snort alerts*

Using this approach, we were able to automatically generate the observation correspondence relations for 60% of the 9000 Snort rules. This is a rough baseline, since the tool has to make an "educated guess" from imprecise and incomplete information. But such an initial model, derived from the knowledge already existent in the Snort rule repository, is helpful for Snort users to get started benefiting from our empirically developed technique. If our reasoning system turns to be useful tool by a significant number of users, there will be incentives for more people to help fine-tune observation correspondence, and even for future Snort rules to include such information. Security expert who writes a Snort rule (or any such specification) has the best knowledge on what the observation means and is best suited to provide this information. This will not incur additional burden since much of this information is already being maintained in an unformatted and ad-hoc manner.

5.3 Snort alert translation

Snort provides different plugins to store the output alerts. We store the alerts in a MySQL database. The tool provides a facility to query the mysql database and translate the Snort alerts stored as Datalog tuples, a format understood by the reasoning engine.

Snort alerts are represented as observations, `obs`. The format of `obs` is shown below.

```
obs(ObsID, snort(SnortID, FromHost, ToHost, TimeStamp)).
```

Each predicate has an observation ID, Snort ID, from and to host IP address along with the timestamp. ObsID is a unique value to identify a snort alert/observation. SnortID, a combination of Generator ID[2] and Snort Rule ID is used to identify the system responsible for raising the alert. FromHost and ToHost IP address gives the source and destination IP

[2] Generator ID identifies internal subsystem that raised the alert.

address of the packet that matched the rule. Timestamp gives us the time when the alert was raised by Snort. Figure 5.4 has some example of translated Snort alerts.

Chapter 6

Experiments

IDS evaluation is elusive because of the unavailability of data and lack of information for verifying the results. Data sets created from production network (real attacks) lack necessary information (ground truths) to verify success of an attack. And data sets created in a controlled setup have ground truths but in most cases do not depict the actual scenarios [53, 54]. In our evaluation our primary goal was to test weather a reasoning model developed empirically from studying one incident can find interesting attack traces in others. We evaluated our reasoning engine on data sets collected using different techniques. Of the three data sets we used, one was collected from a live exercise conducted as part of the graduate course in computer security. The second data set was from a honey pot, and the last data set was from a live production network. Our results clearly indicate that the reasoning engine is capable of identifying interesting traces on the data sets.

6.1 Experimental setup

We used the SnIPS tool for our experiments. First, alerts generated from running Snort on a live network or a tcpdump is sent to a MySQL database for storage. The stored alerts are then translated into observations in Datalog format and stored in a text file. The reasoning engine, internal model, observation correspondence and observations were loaded into a XSB prolog engine. Then the engine is issued with queries like "Is the webServer certainly compromised?" or "Is any host certainly compromised?". The reasoning system

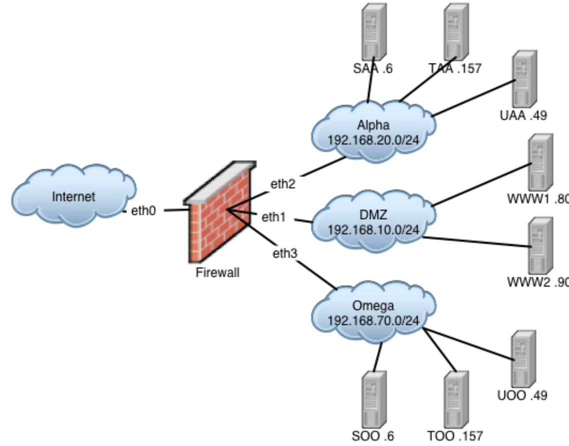

Figure 6.1: *Treasure hunt network topology*

provides the results in the form of strengthened proof and evidences, which is then provided to a system administrator for further evaluation.

6.2 Data sets

6.2.1 Treasure Hunt data set

The first set of experiments was performed on the Treasure Hunt (TH) data set [55]. This data set was created during a cyber-attack competition organized in a graduate security course at University of California, Santa Barbara. Our motivation to use this particular data set was that the data set provides valuable "meta data" such as the back story (competition task details) and network topology which can help us understand the result. We only used the TCPdump portion of the dataset to generate Snort alerts as input to the reasoning engine.

Two teams, Alpha and Omega, were participating in the exercise. In order to prevent the two teams from interfering with each other, two identical networks were setup. The topology of the network used in the exercise is show in figure 6.1[1]. The web server (WWW 1 & 2) was located in a DMZ accessible from the Internet. The MySQL server (SAA/SOO), the

[1] Figure source http://ictf.cs.ucsb.edu/data/treasurehunt2002/

```
(1) int(compromised('192.168.10.90'),c) strengthen
(2)   int(compromised('192.168.10.90'),p) I_3f
(3)       int(sendExploit(external,'192.168.10.90'), p) summarizedFact
            obslist(273)
(4)   int(compromised('192.168.10.90'),l) I_1b
(5)       int(probeOtherMachine('192.168.10.90', '128.111.48.35'), l) summarizedFact
            obslist(257)
```

Figure 6.2: *Partial output trace from the reasoning system*

file server (UAA/UOO), and the transaction server (TAA/TOO) were placed on a separate network, accessible only by the web server host (Servers with suffix AA belong to the Alpha team and suffix OO belong to Omega).

We discuss one of the proofs created by the reasoning engine. The first task in the TH competition was to gain access to the web server. This scenario was identified by our model and the high-confident output is shown in figure 6.2 (the parenthesized numbers are added for explanation purpose). The web server, 192.168.10.90, was certainly compromised (1) based on two independent proofs: (2) and (4), which capture the first step: an exploit being sent from an external host to the web server (3), and the second step: reconnaissance by the attacker from the web server to learn about the internal network (5). The two pieces of evidence both point to the compromise of the web server, strengthening our confidence level to "certain". The raw alerts belonging to the summarized internal condition is identified using the mapping variable `obslist(Var)`. In total, there were 18 such proofs verifying the two web servers were compromised. Table 6.1 shows the reduction in the amount of data that was presented by our tool to the system administrator for further analysis.

We manually validated the raw alerts of the 18 proofs generated by the reasoning engine for false positives. From our analysis we perceive all of them are plausible. The published TH data set did not include a truth file (a file containing information on how the actual attacks were carried out and to what extent they were successful). Thus it was impossible to identify if the reasoning engine missed any true attack traces (false negatives).

6.2.2 Experiment on data collected on a honeypot

We conducted our second set of experiments on a data set collected from a honeypot deployed at Purdue University. This data set contains network traffic information collected for an unrelated project whose main intention was to collect spam relayed using open proxies. The network setup of the honeypot was simple, it had a single host running misconfigured squid proxy server. The data was collected few hours every day for a period of two months. The total size of the zipped tcpdump files was about 68GB. When we applied Snort on the tcpdump, it reported 637,564 alerts from the network traffic. Our analysis on the Snort alert showed that 99% of the alerts were Snort alert '122:27' (portscan:port open). We filtered this particular alert from the input as it did not provide valuable information about host compromise.

The reasoning engine had enough information to conclude the honeypot host was compromised. With the knowledge about operating system and services running in the honeypot host, we validated the traces manually.

6.2.3 Experiments on a production system

The last two experiments also helped us find a few inaccuracies in the observation correspondence relations created by the automated model building process, which were subsequently corrected. In the last experiment, SnIPS with the updated knowledge base was applied on an university network. We installed Snort in our departmental network having around 200 machines including workstation, dedicated web servers, file servers, database servers *etc.* We were only able to analyze the alerts captured over a period of three days. Snort reported about 1.1 Million alerts with 150 different alert types and 15 class types. Table 6.1 shows the number of alerts generated by Snort and the number of high-confidence proofs presented to a system administrator. The 17 proofs pointed that 4 hosts had a higher chance of being compromised. To verify this result would require further analysis of other log data which we did not have access to. We analyzed the output traces with the corresponding

Data set	Snort alerts	Summarized alerts	High-confidence proofs
Treasure Hunt	4,849,937	278	18
Honeypot	637,564	30	8
Production system	1,138,572	6634	17

Table 6.1: *Reduction of alert*

low-level alerts. It appeared that a couple traces were worthy of further investigation and we forwarded those to the system administrator. The others were likely to be false positives.

6.3 Results

From our experiments with the available data sets, we observed that the tool we developed based on true-life incident was able to identify interesting traces of host compromise from low-level Snort alerts and considerably reduced the search space and time involved in identifying intrusion from Snort alerts.

Our current implementation of the reasoning engine only used Snort alerts as input. Since Snort alerts provided a limited view of an intrusion, some of the proof strengthening appeared to be false positives. These experiments helped us to identify potential information and techniques to improve the quality of the reasoning engine. We discuss some of the ideas in chapter 7.

6.4 Observations from the data sets

From our analysis on the data sets, we noticed that a small number of alerts occur in huge numbers in these data set. For example, we can see in the figure 6.3 that in Treasure Hunt data set more than 99% of the total alerts fall under two attack classes "web-application-activity" and "web-application-attack" each having only one type of alert. This could be cased by repeated attempts made by the attacker to make the exploit work by trying different parameters. We also noticed that the traffic from the live network and data collected honeypot had similar characteristics. Huge volume of some alerts were created because

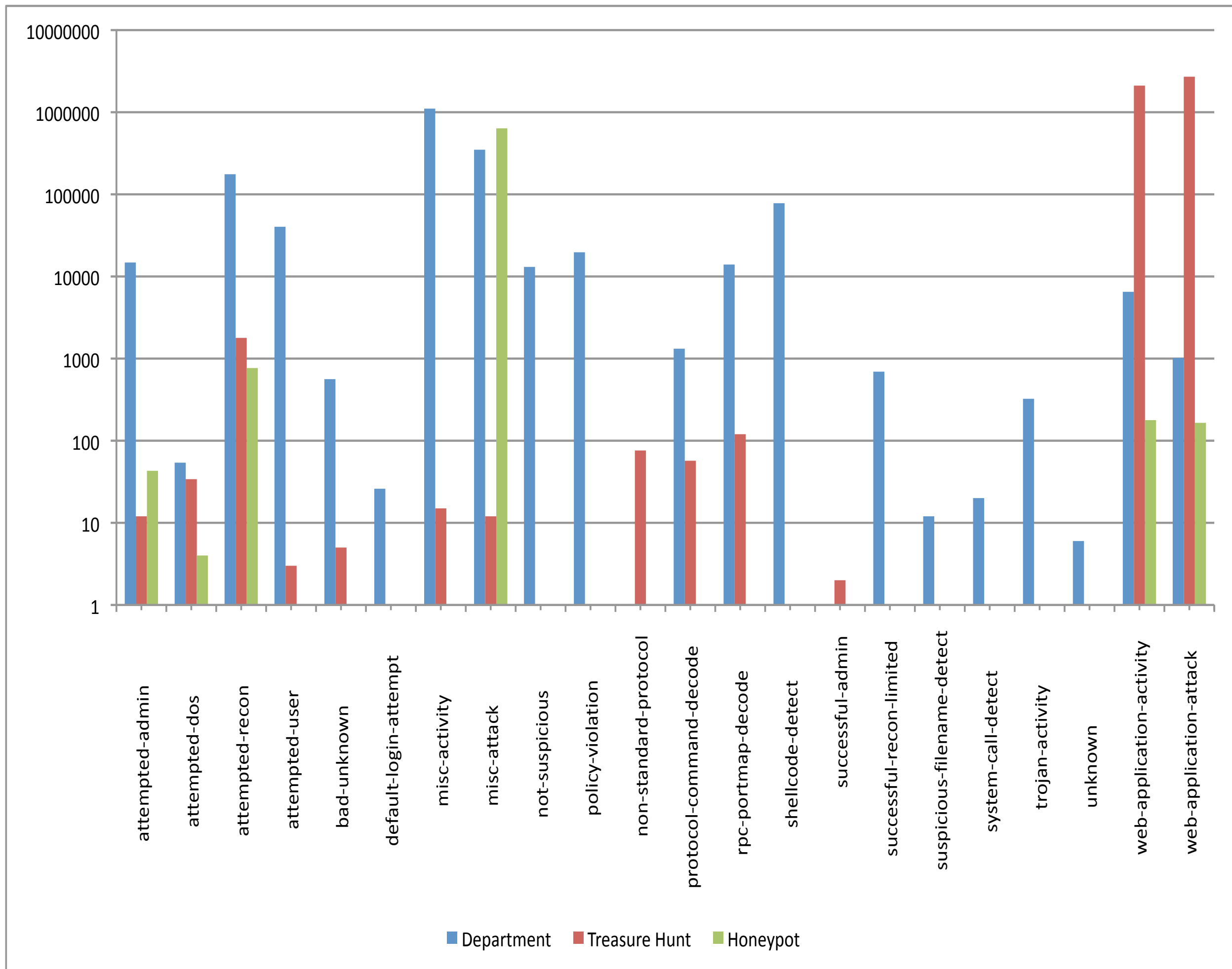

Figure 6.3: *Data distribution in the three data sets*

of noisy rules or repeated probes by Botnet. By using data sets collected from different environment, we were able to test our tool for many diverse cases and scenarios.

Chapter 7

Conclusion

In this thesis, we presented an empirical approach to modeling uncertainty in intrusion analysis. Our goal is to help the system administrator in reaching conclusions quickly about possible intrusions, when multiple pieces of uncertain data have to be integrated. The model language we designed has two components: observation correspondence and internal model. The observation correspondence gives a direct meaning to low-level system monitoring data with explicit uncertainty tags, and can be derived from natural-language description that already exists in some IDS knowledge bases, *e.g.* the Snort rule repository. The internal model is concise and captures general multi-stage attack conditions in an enterprise network. We developed a reasoning system that is easy to understand, handles the uncertainty existent in both observation correspondence and the internal model, and finds high-confidence attack traces from many possible interpretations of the low-level monitoring data. Our prototype and experiments show that the model developed from studying one set of data is effective for analyzing completely different data sets with very little effort. This is a strong indication that the modeling approach can codify the seemingly ad-hoc reasoning process found in intrusion analysis and yield practical tools for enterprise-network security defense.

7.1 Future work

From our empirical study we found new research problems and identified areas that need to be improved in our current implementation. We discuss the ideas in this section and leave

them for future work.

In our current proof strengthening approach, we consider two independent proof paths for proof strengthening. This approach provided a simple mechanism for elevating our beliefs. A graphical representation of the model will provide a greater flexibility for capturing the semantics of the proofs and elevating confidence levels.

Our current implementation uses only the host information for summarization. Having more information about the host, network topology and vulnerability information will help in preventing false strengthening and the alert verification technique discussed in related work can be used to improve the quality of input alerts.

As this is the first time that uncertainty has been dealt with in this explicit but qualitative manner, much work remains to be done, some of which we have already alluded to earlier. Specifically, our modality is a very crude operator – we do not distinguish the various forms of uncertainty. On a practical level adding too much complexity to the modality itself may be counter-productive. But with some maturity of modeling and experience, we hope to separate in our model these two sources of uncertainty as well. As in any modeling tool, there is a natural question of how granular our models have to be to achieve best results. Because the uncertainty of knowledge increases after a certain granularity, we expect that there is an optimal point that can only be discovered with experience. Our modeling and reasoning are monotonic and we do not deal with negation in our models. Although we have not needed it in the data we analyzed, it is plausible that a new observation can *reduce* the modality of an internal condition, *e.g.* from likely to possible. These are some areas subject to future research.

Bibliography

[1] A. Mounji, *Languages and Tools for Rule-Based Distributed Intrusion Detections*, PhD thesis, Purdue University, 1997.

[2] S. R. Snapp et al., Dids (distributed intrusion detection system) - motivation, architecture, and an early prototype, in *In Proceedings of the 14th National Computer Security Conference*, pages 167–176, 1991.

[3] S. Cheung, U. Lindqvist, and M. W. Fong, Modeling multistep cyber attacks for scenario recognition, in *DARPA Information Survivability Conference and Exposition (DISCEX III)*, pages 284–292, Washington, D.C., 2003.

[4] F. Cuppens and A. Miège, Alert correlation in a cooperative intrusion detection framework, in *IEEE Symposium on Security and Privacy*, 2002.

[5] P. Ning, Y. Cui, D. Reeves, and D. Xu, Tools and techniques for analyzing intrusion alerts, in *ACM Transactions on Information and System Security*, volume 7, pages 273–318, 2004.

[6] S. Noel, E. Robertson, and S. Jajodia, Correlating Intrusion Events and Building Attack Scenarios Through Attack Graph Distances, in *20th Annual Computer Security Applications Conference (ACSAC 2004)*, pages 350– 359, 2004.

[7] F. Valeur, *Real-Time Intrusion Detection Alert Correlation*, PhD thesis, University of California, Santa Barbara, 2006.

[8] F. Valeur, G. Vigna, C. Kruegel, and R. A. Kemmerer, A comprehensive approach to intrusion detection alert correlation, in *IEEE Transactions on Dependable and Secure*

Computing, volume 1, pages 146–169, Los Alamitos, CA, USA, 2004, IEEE Computer Society.

[9] Y. Zhai, P. Ning, P. Iyer, and D. S. Reeves, Reasoning about complementary intrusion evidence, in *Proceedings of 20th Annual Computer Security Applications Conference (ACSAC)*, pages 39–48, 2004.

[10] F. V. Jensen and T. D. Nielsen, *Bayesian Networks and Decision Graphs*, Springer Verlag, 2007.

[11] J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*, Morgan Kaufman, 1999.

[12] G. Shafer, *A Mathematical Theory of Evidence*, Princeton University Press, 1976.

[13] G. Shafer, Probability judgment in artificial intelligence and expert systems, in *Statistical Science*, volume 2, pages 3–44, 1987.

[14] M. Almgren, U. Lindqvist, and E. Jonsson, A multi-sensor model to improve automated attack detection, in *11th International Symposium on Recent Advances in Intrusion Detection (RAID 2008)*, RAID, 2008.

[15] T. M. Chen and V. Venkataramanan, Dempster-shafer theory for intrusion detection in ad hoc networks, in *IEEE Internet Computing*, 2005.

[16] G. Modelo-Howard, S. Bagchi, and G. Lebanon, Determining placement of intrusion detectors for a distributed application through bayesian network modeling, in *11th International Symposium on Recent Advances in Intrusion Detection (RAID 2008)*, RAID, 2008.

[17] X. Ou, S. R. Rajagopalan, and S. Sakthivelmurugan, An empirical approach to modeling uncertainty in intrusion analysis, in *Annual Computer Security Applications Conference (ACSAC)*, Honolulu, Hawaii, 2009.

[18] J. Beale, A. R. Baker, J. Esler, T. Kohlenberg, and S. Northcutt, *Snort IDS and IPS toolkit*, Syngress, 2007.

[19] N. Friedman and J. Halpern, Plausibility measures and default reasoning, in *J. ACM*, volume 48, pages 648–685, 2001.

[20] A. Dempster, Upper and lower probabilities induced by a multivalued mapping, in *Ann. Statistics*, volume 28, pages 325–339, 1967.

[21] P. R. Cohen and M. R. Grinberg, Heuristic reasoning about uncertainty, in *AI Magazine*, volume 4, pages 17–24, 1983.

[22] G. Shafer, Probability judgment in artificial intelligence and expert systems, in *Statistical Science*, volume 2, pages 3–44, 1987.

[23] B. Morin, Hervé, and M. Ducassé, M2D2: A formal data model for IDS alert correlation, in *5th International Symposium on Recent Advances in Intrusion Detection (RAID 2002)*, pages 115–137, 2002.

[24] P. Ning, Y. Cui, and D. S. Reeves, Analyzing intensive intrusion alerts via correlation, in *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID 2002)*, LNCS 2516, pages 74–94, 2002.

[25] P. Ning, Y. Cui, and D. S. Reeves, Constructing attack scenarios through correlation of intrusion alerts, in *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS 2002)*, pages 245–254, 2002.

[26] D. Xu and P. Ning, Correlation analysis of intrusion alerts, in *Intrusion Detection Systems*, volume 38 of *Advances in Information Security LNCS 4911*, pages 65 – 92, Springer US, 2008.

[27] C. Kruegel, F. Valeur, and G. Vigna, *Intrusion Detection and Correlation: Challenges and Solutions*, volume 14 of *Advances in Information Security*, Springer, 2005.

[28] K. Julisch, Clustering intrusion detection alarms to support root cause analysis, in *ACM Transactions on Information and System Security (TISSEC)*, volume 6, pages 443–471, New York, NY, USA, 2003, ACM.

[29] F. Cuppens, Managing alerts in a multi-intrusion detection environment, in *Annual Computer Security Applications Conference*, volume 0, page 0022, Los Alamitos, CA, USA, 2001, IEEE Computer Society.

[30] D. Bolzoni, B. Crispo, and S. Etalle, Atlantides: an architecture for alert verification in network intrusion detection systems, in *LISA'07: Proceedings of the 21st conference on Large Installation System Administration Conference*, pages 1–12, Berkeley, CA, USA, 2007, USENIX Association.

[31] X. Liu, D. Xiao, and X. Peng, Towards a collaborative and systematic approach to alert verification, in *JSW*, volume 3, pages 77–84, 2008.

[32] D. J. Chaboya, R. A. Raines, R. O. Baldwin, and B. E. Mullins, Reliably determining the outcome of computer network attacks, in *18th Annual FIRST conference*, 2006.

[33] J. Zhou, A. J. Carlson, and M. Bishop, Verify results of network intrusion alerts using lightweight protocol analysis, in *Annual Computer Security Applications Conference*, volume 0, pages 117–126, Los Alamitos, CA, USA, 2005, IEEE Computer Society.

[34] C. Kruegel and W. Robertson, Alert verification - determining the success of intrusion attempts, in *Proc. First Workshop the Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA 2004)*, pages 1–14, 2004.

[35] L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. Mitchell, A layered architecture for detecting malicious behaviors, in *11th International Symposium on Recent Advances in Intrusion Detection (RAID 2008)*, RAID, 2008.

[36] K. Ingols, R. Lippmann, and K. Piwowarski, Practical attack graph generation for network defense, in *22nd Annual Computer Security Applications Conference (ACSAC)*, Miami Beach, Florida, 2006.

[37] S. Jajodia, S. Noel, and B. O'Berry, Topological analysis of network attack vulnerability, in *Managing Cyber Threats: Issues, Approaches and Challanges*, edited by V. Kumar, J. Srivastava, and A. Lazarevic, chapter 5, Kluwer Academic Publisher, 2003.

[38] X. Ou, W. F. Boyer, and M. A. McQueen, A scalable approach to attack graph generation, in *13th ACM Conference on Computer and Communications Security (CCS)*, pages 336–345, 2006.

[39] X. Ou, S. Govindavajhala, and A. W. Appel, MulVAL: A logic-based network security analyzer, in *14th USENIX Security Symposium*, 2005.

[40] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. M. Wing, Automated generation and analysis of attack graphs, in *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 254–265, 2002.

[41] L. Wang, A. Liu, and S. Jajodia, Using attack graphs for correlating, hypothesizing, and predicting intrusion alerts, in *Computer Communications*, volume 29, pages 2917–2933, 2006.

[42] Y. Zhang, X. Fan, Y. Wang, and Z. Xue, Attack grammar: A new approach to modeling and analyzing network attack sequences, in *Annual Computer Security Applications Conference (ACSAC)*, 2008.

[43] G. Gu, P. Porras, V. Yegneswaran, M. Fong, and W. Lee, BotHunter: Detecting malware infection through ids-driven dialog correlation, in *Proceedings of the 16th USENIX Security Symposium (Security'07)*, 2007.

[44] T. Hollebeek and R. Waltzman, The role of suspicion in model-based intrusion detection, in *Proceedings of the 2004 workshop on New security paradigms*, 2004.

[45] R. P. G. S. A. Harp, Model-based intrusion assessment in common lisp, in *International Lisp Conference, 2009, Association of Lisp Users and ACM SIGPLAN, Cambridge, MA. (2009)*, Cambridge, MA, 2009.

[46] M. Goldszmidt and J. Pearl, System-z+: A formalism for reasoning with variable-strength defaults, in *AAAI*, pages 399–404, 1991.

[47] J. McHugh, Intrusion and intrusion detection, in *International Journal of Information Security*, volume 1, pages 14 – 35, 2001.

[48] S. Ceri, G. Gottlob, and L. Tanca, What you always wanted to know about Datalog (and never dared to ask), in *IEEE Transactions Knowledge and Data Engineering*, volume 1, pages 146–166, 1989.

[49] P. Rao, F. S. Konstantinos, T. Swift, D. S. Warren, and J. Freire, XSB: A system for efficiently computing well-founded semantics, in *Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR'97)*, pages 2–17, Dagstuhl, Germany, 1997, Springer Verlag.

[50] M. Roesch, Snort - lightweight intrusion detection for networks, in *LISA '99: Proceedings of the 13th USENIX conference on System administration*, pages 229–238, Berkeley, CA, USA, 1999, USENIX Association.

[51] http://www.snort.org.

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

[53] J. McHugh, Testing intrusion detection systems: a critique of the 1998 and 1999 darpa intrusion detection system evaluations as performed by lincoln laboratory, in *ACM Trans. Inf. Syst. Secur.*, volume 3, pages 262–294, 2000.

[54] M. V. Mahoney and P. K. Chan, An analysis of the 1999 darpa/lincoln laboratory evaluation data for network anomaly detection, in *RAID*, pages 220–237, 2003.

[55] G. Vigna, Teaching Network Security Through Live Exercises, in *Proceedings of the Third Annual World Conference on Information Security Education (WISE 3)*, edited by C. Irvine and H. Armstrong, pages 3–18, Monterey, CA, 2003, Kluwer Academic Publishers.

Appendix A

Snort – Intrusion detection system

A.1 Snort default classtype

Classtype	Description	Priority
attempted-admin	Attempted Administrator Privilege Gain	high
attempted-user	Attempted User Privilege Gain	high
kickass-porn	SCORE! Get the lotion!	high
policy-violation	Potential Corporate Privacy Violation	high
shellcode-detect	Executable code was detected	high
successful-admin	Successful Administrator Privilege Gain	high
successful-user	Successful User Privilege Gain	high
trojan-activity	A Network Trojan was detected	high
unsuccessful-user	Unsuccessful User Privilege Gain	high
web-application-attack	Web Application Attack	high
attempted-dos	Attempted Denial of Service	medium
attempted-recon	Attempted Information Leak	medium
bad-unknown	Potentially Bad Traffic	medium
default-login-attempt	Attempt to login by a default username and password	medium
denial-of-service	Detection of a Denial of Service Attack	medium
misc-attack	Misc Attack	medium
non-standard-protocol	Detection of a non-standard protocol or event	medium
rpc-portmap-decode	Decode of an RPC Query	medium
successful-dos	Denial of Service	medium
successful-recon-largescale	Large Scale Information Leak	medium
successful-recon-limited	Information Leak	medium
suspicious-filename-detect	A suspicious filename was detected	medium
suspicious-login	An attempted login using a suspicious username was detected	medium
system-call-detect	A system call was detected	medium
unusual-client-port-connection	A client was using an unusual port	medium
web-application-activity	Access to a potentially vulnerable web application	medium
icmp-event	Generic ICMP event	low
misc-activity	Misc activity	low
network-scan	Detection of a Network Scan	low
not-suspicious	Not Suspicious Traffic	low
protocol-command-decode	Generic Protocol Command Decode	low
string-detect	A suspicious string was detected	low
unknown	Unknown Traffic	low
tcp-connection	A TCP connection was detected	very low

A.2 An example Snort rule document

A Snort rule document is a plain text file which contains a detailed information about a Snort rule. This document is typically created by the rule writer. The following is an example Snort rule document taken from "4333.txt":

```
Rule:


--
Sid:
4333


--
Summary:
This event is generated when an attempt is made to exploit a known
vulnerability is the Microsoft Plug and Play subsystem on a host.


--
Impact:
Serious. Execution of code is possible leading to full system
compromise.


--
Detailed Information:
A programming error in the Plug and Play (PnP) service used by
Microsoft Windows machines can present a remote attacker with the
opportunity to overflow a fixed length buffer, execute code on the
vulnerable system and escalate privileges on the host to the extent
that they could take complete control of the affected machine.
```

Specifically this event is generated when the UMPNGMGR interface is targeted via the function.

--

Affected Systems:

Microsoft Windows 2000

Microsoft Windows XP

Microsoft Windows 2003

--

Attack Scenarios:

An attacker merely needs to send extra data to the PnP service to overflow the buffer and execute code of their choosing on the affected system.

--

Ease of Attack:

Simple.

--

False Positives:

None known.

--

False Negatives:

None known.

```
--
Corrective Action:
Apply the appropriate vendor supplied patches.

Disallow access to ports 139 and 445 from sources external to the
protected network.

Consider disabling the use of those ports completely on Windows based
networks.

--
Contributors:
Sourcefire Vulnerability Research Team
Matt Watchinski <matthew.watchinski@sourcefire.com>
Nigel Houghton <nigel.houghton@sourcefire.com>

--
Additional References:



--
```

A.3 Snort database schema

Snort database schema taken from [51] gives the structure of the Snort alerts stored in MySQL database. We briefly describe the table structure here.

The schema is designed to store alerts reported by more than one sensor and the ta-

ble “sensor” contains information about each sensor. The “sid” field in this table helps uniquely identify a sensor. And each event reported by a sensor has an entry in the “event” table, which is uniquely identified by the key pair “cid” and “sid”. The event table also contains a field to identify signature of an event and timestamp. The “signature” table contains the meta information about events such as the class type, priority, Snort rule id and Snort rule revision. Using tables “sig_reference”, “reference” and “reference_system” we can find reference to various vulnerability resources. The table “iphdr” holds the source and destination IP address and other flags set in the packet that matched a rule. Tables “icmphdr”, “tcphdr” and “udphdr” contains the header information of icmp, tcp and udp packets respectively.

Snort Database Schema v1.06
Diagram created Sept 10, 2004
chris.reid@codecraftconsultants.com

Standard Snort Table
SnortDB Extra Table
Bold fields are “not null”

schema
PK **vseq**
ctime

sig_reference
PK,FK2 **sig_id**
PK,FK1 **ref_id**
ref_seq

reference
PK **ref_id**
FK1 **ref_system_id**
ref_tag

reference_system
PK **ref_system_id**
ref_system_name

signature
PK **sig_id**
sig_name
FK1 sig_class_id
sig_priority
sig_rev
sig_sid

sig_class
PK **sig_class_id**
sig_class_name

detail
PK **detail_type**
detail_text

encoding
PK **encoding_type**
encoding_text

sensor
PK **sid**
hostname
interface
filter
FK2 detail
FK1 encoding
last_cid

event
PK,FK2 **sid**
PK **cid**
FK1 **signature**
timestamp

protocols
protocol
name
description

iphdr
PK,FK1 **cid**
PK,FK1 **sid**
ip_src
ip_dst
ip_ver
ip_hlen
ip_tos
ip_len
ip_id
ip_flags
ip_off
ip_ttl
FK2 ip_proto
ip_csum

data
PK,FK1 **cid**
PK,FK1 **sid**
data_payload

opt
PK,FK1 **cid**
PK,FK1 **sid**
PK **optid**
opt_proto
opt_code
opt_len
opt_data

icmphdr
PK,FK1 **cid**
PK,FK1 **sid**
icmp_type
icmp_code
icmp_csum
icmp_id
icmp_seq

udphdr
PK,FK1 **cid**
PK,FK1 **sid**
FK2 **udp_sport**
FK3 **udp_dport**
udp_len
udp_csum

tcphdr
PK,FK2 **cid**
PK,FK2 **sid**
FK3 **tcp_sport**
FK4 **tcp_dport**
tcp_seq
tcp_ack
tcp_off
tcp_res
FK1 **tcp_flags**
tcp_win
tcp_csum
tcp_urp

flags
number
RES1
RES2
URG
ACK
PSH
RST
SYN
FIN
valid
description

services
port
protocol
name
description