INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI[®]

**USING OBJECT-ORIENTED MODELING FOR SPECIFYING
AND DESIGNING A NETWORK-CONTEXT SENSITIVE
INTRUSION DETECTION SYSTEM**

By

Frédéric Massicotte

A thesis submitted to

the Faculty of Graduate Studies and Research

In partial fulfilment of the requirements for the degree of

Master of Applied Science

Ottawa-Carleton Institute of Electrical and Computer Engineering

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario, Canada

August 2005

© 2005 by Frédéric Massicotte

Library and
Archives Canada

Bibliothèque et
Archives Canada

0-494-08379-4

Published Heritage
Branch

Direction du
Patrimoine de l'édition

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file *Votre référence*

ISBN:

Our file *Notre référence*

ISBN:

NOTICE:

The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protègent cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

The undersigned hereby recommend to
the Faculty of Graduate Studies and Research
Acceptance of the thesis

**Using object-oriented modeling for specifying and designing a
network-context sensitive intrusion detection system**

Submitted by
Frédéric Massicotte

**In partial fulfilment of the requirements for the degree of
Master of Applied Science**

Professor L. C. Briand (Supervisor)

Professor Y. Labiche (Co-Supervisor)

Carleton University

August 2005

Abstract

Intrusion Detection Systems (IDS) have the reputation of generating many false positives. To resolve this issue, several IDS vendors have decided to use a stateful approach to intrusion detection at the TCP session layer and the application layer. Such an approach allows to take into account the state of the session when an intrusion is detected. Since intrusions can only occur during specific session states, this in turn allows to reduce the number of false positives. However, a stateful approach can only reach its full potential if these Stateful IDS (SIDS) are able to correlate the intrusion information with additional network information gathered before the intrusion. The state of the sessions is not sufficient to reduce the number of false positives. Information such as the configuration of a system, its operating system, its role in the network and its active services is also needed by IDS to derive the context of an intrusion, correctly place the attack in this context and thus reduce the number of false positives. In this thesis, we show how the Object Constraint Language can be used as an intuitive high-level language to specify passive network monitoring rules, allowing the addition of more context to network intrusion detection rules.

Acknowledgements

I would like to warmly thank Dr. Briand and Dr. Labiche for their guidance and for allowing me to work on such a stimulating subject throughout the course of my Master's degree.

I would also like to thank Dr. Esfandiari, Dr. Miri and Dr. Wainer for accepting to review and evaluate this thesis.

I am grateful for the help provided by a number of my fellow students and colleagues, in particular Alexandre Sauvé, Mathieu Couture, John Robinson and Tim Symchych. Your continued support was greatly appreciated.

Table of contents

1	Introduction	1
1.1	Thesis Contribution	4
1.2	Thesis Organization	5
2	Related Work and Requirements	8
2.1	Related Work	9
2.1.1	Network Model	9
2.1.2	Packet Model	13
2.1.3	Multi-Packet Monitoring System	14
2.2	System Requirements	18
2.2.1	Intrusion Detection System Requirements	19
2.2.2	Packet and Network Model Requirements	20
3	PNMT Approach and Design	23
3.1	System Approach Overview	23

3.2	Monitored Objects	25
3.2.1	Qualifying Network Information	26
3.2.2	Network Model	29
3.2.2.1	Network Model Overview	29
3.2.2.2	Description of Classes	33
3.2.2.3	Description of Relationships	35
3.2.2.4	Dynamic Information	38
3.2.2.5	Static Information	39
3.2.3	Packet Model	40
3.2.3.1	Packet Model Overview	40
3.2.3.2	Description of Classes	42
3.2.3.3	Description of Relationships	46
3.3	Rule Specification	47
3.4	Design and Implementation	53
3.4.1	System Design and Implementation Overview	53
3.4.2	Monitoring Engine	54
3.4.2.1	Framework Specification	54
3.4.2.2	Description of Framework Classes	57
3.4.2.3	AbstractScenario Example	59
3.4.2.4	AnalysePacket Algorithm	62

3.5	Discussion about System Requirements	64
4	Populating the Network Model	67
4.1	Description of Information Sources	68
4.1.1	Snort Rules	70
4.1.2	Nessus Scripts	70
4.1.3	Bugtraq Vulnerability Database	71
4.1.4	CVE Dictionary	71
4.1.5	Information Sources Used with the Network Model	72
4.2	Analysis of Database References	74
4.2.1	Analysis of Snort References	74
4.2.2	Analysis of Snort Relationships	75
5	Tests and Results	85
5.1	Test Procedure	86
5.2	Implemented Passive Network Monitoring Rules	89
5.2.1	Passive Network Context Gathering	89
5.2.1.1	Open Session Rule	90
5.2.1.2	Operating System Inference Rules	91
5.2.1.3	Product Detection Rules	95
5.2.2	Context-Based Intrusion Detection	98

5.2.2.1	Network Context in Intrusion Detection Rules	99
5.2.2.2	Utility Rules	101
5.2.2.3	Microsoft IIS FTP Connection Status Request Denial of Service	103
5.2.2.4	Microsoft MSDTC Service Denial of Service	109
5.2.2.5	Microsoft IIS Server 5.0 printer ISAPI Extension Buffer Over- flow	112
5.2.2.6	Sendmail Header Processing Buffer Overflow	115
5.2.2.7	Microsoft DCOM RPC Interface Buffer Overflow	118
5.2.2.8	Wu-ftpd File Globbing Heap Corruption	124
6	Future Work	128
6.1	Information Gathering Rules	129
6.2	Proposed Modification to PNMT	130
6.2.1	System Overview	131
6.2.2	Past Linear Temporal Logic	132
7	Conclusion	137

List of Tables

3.1	Summary of the complexity of the code	53
4.1	Summarized Description of Intrusion and Vulnerability Databases	69
4.2	Static Information Sources	73
4.3	Dynamic Information Sources	73
5.1	Snort Signature Field Description	105
5.2	Abbreviations: Result Tables	108
5.3	Microsoft IIS FTP Status Request Denial of Service Results	108
5.4	Microsoft MSDTC Service Denial of Service Results	111
5.5	Microsoft IIS 5.0 .printer ISAPI Extension Buffer Overflow Results	114
5.6	Sendmail Header Processing Buffer Overflow Results	118
5.7	Microsoft DCOM RPC Interface Buffer Overrun Results	122
5.8	Wu-Ftpd File Globbing Heap Corruption Results	126

List of Figures

3.1	PNMT Collaboration Diagram	25
3.2	Network Model	30
3.3	Packet Model	42
3.4	OCL Formula Modeling an Open Port	50
3.5	TCP/IP Flow Rules	51
3.6	Proposed Design for the PNMT Collaboration Diagram	55
3.7	Framework Model	56
3.8	Open Port Rule Programmed Using the Framework	60
3.9	AnalysePacket Algorithm	63
4.1	Distribution of Snort Rule References	75
4.2	Snort, Nessus and Bugtraq Relationships Diagram	76
4.3	Distribution of Snort Rule Group References	77
4.4	Distribution of Nessus References	82

5.1	Open TCP Session Rule	91
5.2	SYN Packet OS Fingerprinting Rule for FreeBSD 4.5 to 5.1	93
5.3	SYN_ACK Packet OS Fingerprinting Rule for FreeBSD 3.0 to 4.3	94
5.4	Session Utility Rules	96
5.5	Microsoft IIS FTP Server Detection Rule	97
5.6	IPStack Utility Rules	101
5.7	logAlarm Utility Rule	103
5.8	Snort "FTP EXPLOIT STAT ? dos attempt" Rule	104
5.9	PNMT OCL "FTP Exploit STAT ? dos attempt" Rule	107
5.10	Snort "DOS MSDTC attempt" Rule	109
5.11	PNMT OCL "DOS MSDTC attempt" Rule	111
5.12	Snort "WEBIIS ISAPI .printer access" Rule	112
5.13	PNMT OCL "WEB-IIS ISAPI .printer access" Rule	114
5.14	Snort "SMTP From comment overflow attempt" Rule	116
5.15	PNMT OCL "SMTP From comment overflow attempt" Rule	117
5.16	Snort "NETBIOS DCERPC ISystemActivator path overflow attempt little endian unicode" Rule	119
5.17	PNMT OCL "NETBIOS DCERPC ISystemActivator path overflow attempt little endian unicode" Rule	121
5.18	Snort "FTP wu-ftp bad file completion attempt [" Rule	125

5.19 PNMT OCL "FTP wu-ftp bad file completion attempt [" Rule	126
---	-----

*À mon père,
à ma mère,
à ma famille,
à mes amis,
à Lucie, à Véronique, à Alexandre ...*

*Home is not a place,
it's where your passion drives you.*

Chapter 1

Introduction

In today's world, it is increasingly difficult to discover what is going on in networks. Because networks are huge and very dynamic, network administrators need better ways to monitor and protect them against intrusions and viruses. Many aspects of networks need to be monitored in real-time. To monitor networks, administrators use, among other components, Intrusion Detection Systems (IDS). An IDS is a software component responsible for identifying undesirable utilization of computer resources. Most of available IDS use signatures based on packets and protocol headers to protect networks. This approach is the most efficient and most successful one compared to other available methods. However, these Intrusion Detection Systems have the reputation of generating many false positives. To address this issue, IDS vendors have provided several options. First, they have proposed to add more context information to their intrusion detection signatures by using a stateful approach to intrusion

detection for communication sessions. These Stateful IDS (SIDS) are intended to reduce the number of false positives generated by IDS. They do this by keeping track of session states. This approach works because the attacks can only be successful during specific states of a session.

Correlation systems represent a second approach proposed by vendors. These systems are used with IDS and other network monitoring tools, such as passive and active Vulnerability Detection Systems (VDS), to correlate IDS alarms with vulnerabilities discovered by the VDS. Correlation systems gather events from these different sources and place IDS alarms within a larger network context than the one described by an IDS alone. The network administrator, using the correlation system, is better equipped to confirm or discredit the alarms raised by the IDS. This approach works because many IDS alarms include references to entries in vulnerability databases. Thus, the correlation engine is able to locate the vulnerability entries corresponding to the generated alarms and to determine, using the VDS events, whether these vulnerabilities are relevant for the computer associated with the alarm.

Despite the adoption of stateful approaches and of correlation systems, we believe that IDS still have two drawbacks. First, the problem with the false positives generated by IDS partly lies in their inability to incorporate intrusion alarms in a larger network context. Specifically, they are not able to fully incorporate the context and the state of the network in their intrusion signatures. A closer examination shows that more network context needs

to be incorporated into IDS signatures for this approach to reach its full potential, because key components need to be used by the targeted system for an attack to succeed. To provide this functionality, IDS also need to be able to acquire more network context to correlate it with potential intrusions. Information such as the configuration of a system, its operating system, its role in the network, its active services and other factors is also needed by IDS to derive the context of an intrusion.

A second problem with IDS is that they mostly rely on a single packet to identify attacks. IDS do have plug-ins able to detect attacks that can only be expressed using a sequence of packets, such as distributed denial of service attacks and port scans. IDS also have plug-ins to keep track of the state of TCP and application sessions. However, in both cases, the multi-packet attack plug-ins and the stateful plug-ins need to be programmed and the system must be rebuilt. Moreover, these plug-ins are not able to keep track of the full network context, such as the configuration of computers and where they are located in the network. Furthermore, the behavior of attacks that can only be expressed using a sequence of packets cannot be specified in the IDS signature language. The IDS signature languages are used by network administrators to specify intrusion signatures that IDS have to look for on the network. These intrusion signatures are usually logical expressions of specific characteristics corresponding to only one intrusion packet. One of the reasons that make IDS so popular is that their declarative intrusion signature language enables intrusion signatures to be updated without changing the monitoring engine. However, even if they have a flexible signature

language, such IDS could still be improved by using a more flexible declarative language that could express multi-packet intrusions and network context gathering.

1.1 Thesis Contribution

The main objective of this thesis is to add network context to intrusion detection rules to verify that such rules can reduce the number of false positives. To resolve these problems, we propose a system able to passively acquire network context information. This system has a declarative information acquisition language to specify information acquisition rules, which allow to capture network context information, as well as intrusion detection rules. These information acquisition rules and intrusion detection rules are specified in a multi-packet rule language to capture more complex behaviors than those captured by standard IDS. More specifically, our approach has five key components: a packet capturing engine, a declarative rule specification language, a multi-packet monitoring engine, a packet model and a network model.

In this thesis, we present a Passive Network Monitoring Tool (PNMT) that incorporates all these components. This tool results from a collaboration between Carleton University and the Communications Research Center of Canada (CRC)¹. This passive monitoring system can monitor communication patterns that can only be expressed using a multi-packet signature. It is also able to capture the network context while monitoring the network for intrusions.

¹The Communications Research Centre is an Agency of Industry Canada.

The main contributions of this thesis are the following:

- A multi-packet monitoring engine, a packet capturing engine and a framework to program multi-packet scenarios that are able to capture communication patterns using multiple packets.
- A declarative rule specification language that uses the Object Constraint Language (OCL) [41] to specify how to infer the network context and to identify intrusions based on a sequence of packets.
- A formal packet model and a formal network model to specify how to capture the network context based on a sequence of packets.
- An analysis of the relationships between the Snort intrusion detection rules, the Nessus vulnerability detection rules and the Bugtraq vulnerability database.
- A library of OCL passive information gathering rules to acquire the network context and a library of OCL intrusion detection rules with network context that reduce the number of false positives generated by intrusion detection systems.

1.2 Thesis Organization

Chapter 2 describes the related work conducted to integrate network context into IDS signatures. In particular, Section 2.1 describes the different approaches that have been suggested

to implement multi-packet IDS and to model the network context needed for event correlation and passive network monitoring. The different paradigms that can be used to define a monitoring system are also explained in this section. Section 2.2 presents the main requirements that a multi-packet IDS must provide to network administrators and describes the packet and the network model requirements for network context gathering.

Chapter 3 describes the approach used in PNMT, its design and implementation. Specifically, Section 3.1 presents the system approach for PNMT and each of its components. Section 3.2 describes the packet and the network model that are used by OCL to specify how to infer passive network information. These models are also used by PNMT to model the full network context. Section 3.3 describes OCL and how it can be used to specify rules for passive network monitoring. Section 3.4 presents the design and implementation of PNMT. This section also describes the framework developed in Java that allows to program OCL information gathering rules that can be used with PNMT to monitor network traffic.

Chapter 4 explains what information could be used to populate the network model and to derive information gathering rules. Section 4.1 provides a brief description of the different sources of information used in this analysis, a description of their relationships and how this information can be used to provide IDS rules with network context. Section 4.2 describes the results of the analysis of Snort rule relationships to the Nessus and Bugtraq databases.

Chapter 5 puts all these concepts together and describes some intrusion and information gathering rules. Section 5.1 describes how each programmed OCL rule is tested to prove

the effectiveness of PNMT to reduce the number of false positives. Section 5.2 explains how information sources can be used together to incorporate network context in intrusion signatures and to correlate information from different sources.

Chapter 6 provides a brief description of some of the future work that we plan to conduct to improve this concept. Section 6.1 summarizes the proposed development of a larger rule set and describes an improved testing technique for a larger sample of network traffic to be tested with PNMT. Section 6.2 describes some modifications to PNMT that have to be made to compile OCL rules into code. A brief description of the Past Linear Temporal Logic (PLTL) is also provided to describe how it could be used as a paradigm to compile OCL rules and to specify the algorithm of the monitoring engine.

The last chapter concludes this thesis by summarizing the results of the work conducted in this project and discussing some of their implications.

Chapter 2

Related Work and Requirements

To build a multi-packet passive monitoring tool, two components are needed: a multi-event monitoring engine and a declarative rule specification language. A number of multi-event paradigms have been proposed in the literature for multi-packet IDS engines. Their rule languages are typically based on a packet model used to specify intrusion scenarios. In the case of a passive information gathering tool, a packet and a network model are needed in order to capture the context of the network to correlate it with intrusions. Different models have been proposed to represent the packet model and the computer network model on which the signature language is specified. In the next sections, we review related work, as well as the system requirements needed to develop such a monitoring tool.

2.1 Related Work

In this section, related work on multi-packet monitoring engines, the network model and the packet model for network monitoring are presented. This survey of the literature is then used in the next section to derive the requirements needed to develop a passive network monitoring tool to reduce the number of false positives.

2.1.1 Network Model

To correlate network events such as IDS alarms with VDS events and to incorporate the network context in intrusion detection signatures, several components are needed. The key component required to enable network context gathering and event correlation is a formal network model that defines relationships between network components, their configuration, as well as known exploits and vulnerabilities. Such a component is missing from most current IDS, which only maintain an alarm database that they populate based on streams of packets. SIDS also maintain the state of communication session, but they do not provide a network model that captures all the network context. Thus, both IDS and SIDS are unable to perform information correlation between the network context and the alarms.

As far as can be determined from available research, few methods have been proposed to model a computer network from a perspective useful to our research activities. The most important contributions come from Vigna [66], Goldman et al. [22] and M2D2 [39]. In 1996, Vigna [66] proposed a model for network topology to address spoofing and hijacking

problems by automatically deriving the location of IDS sensors on the network. In particular, the Vigna model was used in [65] to develop a correlation system and in [67] for use in an intrusion detection system. In 2001, Goldman et al. [22] also introduced another computer network model to correlate intrusion detection reports with the proper network context. In 2002, M2D2 [39] incorporated both computer network concepts to create a more formal network model.

M2D2 is probably the most formal and complete of these three network models. It is mostly inspired from the two other network models and provides a model for information correlation with the full network context. In particular, M2D2 was proposed to address the problem of correlating intrusion alerts with contextual information.

The M2D2 network model is defined as a set of relations and functions over given primitive sets such as network interfaces and products. For formalism and notation, M2D2 relies on Z and B . As with the Vigna model, the network topology with M2D2 is defined from the primitive set of network interfaces. Then, since a host may have multiple network interfaces, and since each interface must belong to exactly one host, the host set is defined as a partition of the interface set. In a similar way, the data links are also defined as a partition of the network interfaces. Then, the host software configuration is defined as a function assigning products to hosts, and so on.

Compared to other models we have mentioned, M2D2 seems to be the most appropriate for the purpose of modeling networks. First, it has a wide scope, including not only net-

work topology, but also host configurations, which encompass executing software and their vulnerabilities (as opposed to the model proposed by [66], which does not include these features). Second, it explicitly defines the cardinality relationships among the represented entities, which is useful to infer new information about the network.

However, the complete M2D2 model does not exactly meet our needs for a number of reasons. First, this model does not use an object-oriented concept to model the network. We think that networks are more appropriately modeled as objects for two reasons: because network components are objects; and because the inheritance relationship between objects is well-suited for the expression of different sub-categories of network components with the same behavior. Second, the representation of dynamic information in M2D2 is only modeled for security-related events and not for network and host configurations. Thus, this model does not allow for real-time passive network information gathering of the network and host configurations. In fact, the only dynamic information this model allows us to represent is primitive network events, such as IDS alarms and IP, TCP, UDP or HTTP events.

In fact, none of the reviewed models is flexible enough for use in a real-time monitoring system such as PNMT. They are designed for ideal situations where the network context is entirely known or where the network model is only used in a correlation system. In particular, they were not developed to add network context to an IDS signature language.

IDS vendors have made several attempts to incorporate network context into their IDS products and to provide correlation system consoles that allow the network administrator

to make decisions. In particular, SourceFire [61], Tenable [54] and ISS [62] have proposed different options to do this. SourceFire developed a system called Real-time Network Awareness [58] that passively captures information about the monitored network. ISS and Tenable also provide products, such as RealSecure Network [63] and NeVO [53, 16, 24], able to acquire information about networks to more effectively detect intrusions on a network. ISS and Tenable also offer correlation systems, such as Fusion and Lightning Console, to combine information from IDS and VDS. However, network models are usually not provided and when they are they are not expressed using well-defined formalisms but rather ad hoc information gathering mechanisms.

As mentioned before, M2D2 provides the most appropriate network model. To meet our needs, we designed a network model inspired from M2D2 to model the different characteristics of the information we have to handle, thus allowing it to add network context to an IDS signature language. In particular, we had to take into account that M2D2 does not provide a way to capture static and dynamic information. To add network context to IDS or to correlate IDS events with VDS, it is essential for the network model to be able to capture such information. For instance, information such as the relationship between a known vulnerability and a particular software is static (i.e., a given vulnerability will always be associated with a particular version of a product), while other information such as the relationship between a network interface and an IP address is dynamic (i.e., a network interface could change its IP address). More details on our approach are provided in section 3.2.2.

2.1.2 Packet Model

We found three different ways to model packets in the literature. Since higher protocol headers are placed in the payload of a lower protocol, the first implementation model one can imagine is a chained list, where each protocol header has a field *nextHeader*. However, a chained list forces us to write packet field specifications similar to *packet.ethernet.ip.udp.sourcePort* to access packet field information. This is only valid in a context where UDP runs over IP and IP runs over Ethernet. Although this is usually the case, it is also often the case that IP will be part of an ICMP Port Unreachable packet. In this case, IP is also part of the next header of this ICMP packet. Thus, the lower protocol stack is different (i.e., IP would not run over ICMP), and we wish to be able to handle such situations.

The second way to model packets is based on the M2D2 and Vigna models and proposes that an UPD diagram is an IP diagram [66]. If we make such an assumption, then it means that we see an inheritance relationship between the two entities. Hence, it would be possible to write packet field specification such as *packet.udp.timeToLive* even if the field *timeToLive* belongs to the IP header. This comes down to the same problem we previously noted, since it is possible that UDP is not running over IP. Nevertheless, this approach has already been adopted by the providers of *jpcap* [11], a Java library allowing packet capturing in Java.

The third approach to model packets is the one used by Snort developers and it is the model we adopted. It consists of a structure that may potentially contain every existing protocol headers. Hence, it allows writing packet field specifications such as *packet.udp.sourcePort*,

without worrying about the protocol over which UDP runs.

2.1.3 Multi-Packet Monitoring System

In a number of areas, real-time system monitoring is very important to enforce security policies and to make sure that systems are functioning according to their specification. Real-time monitoring systems often include features that enable the monitoring engine to access any previous states of the monitored system to discover complex pattern. These monitoring systems are used in power plants, assembly lines, critical software monitoring, etc. In many cases, these systems monitor series of events and try to discover problematic patterns. When a pattern is discovered, these monitoring systems have to put these series of events into their proper context. Although they provide real-time system monitoring, common IDS engines only use a single-packet monitoring engine. They do not provide a built-in, multi-packet monitoring engine to capture more complex intrusion and attack behaviors. Furthermore, they do not analyze events in their complete network context.

Different paradigms have been used to create monitoring engines that are able to monitor series of events, which in turns allow inferring alarms and discovering abnormal behavior. The most common approaches include Temporal Logic [50], Petri Nets [33, 31, 32], State Machines [17, 67], Expert Systems [2, 1, 35], Event Calculus [38], Regular Expression [57] and d hoc paradigms [44], [59, 23] and [53, 16, 24].

Through a collaboration with CRC, we conducted a survey of multi-packet IDS in [12]

and [13] to provide a classification of these IDS and to show that most of them do not provide an adequate solutions for the capture of network context as well as for the provision of a multi-packet monitoring engine. Some of the languages used by the IDS presented in this survey are not declarative and in most cases the rule specification language is dependent on the monitoring algorithm. Below we present a summary of this survey that provides our main conclusions regarding the usefulness of these IDS.

The intrusion detection system called STAT (State Transition Analysis Technique) [17] uses state machines to describe the monitored behavior. STAT uses a language called STATL to describe the state machines. This language can be used by a variety of tools such as USTAT [28], WinSTAT and NetSTAT [67]. These tools are signature-based IDS and they are used respectively to detect intrusions in a Unix log file, a Windows log file and on a network by capturing packets. One problem with STATL is that this language is not completely declarative. However, an interesting aspect of NetSTAT is that this IDS contains a network information model to represent the network context for intrusion. Unfortunately, the documentation available on this aspect of NetSTAT is too scant for us to use it in this project.

Another method that has been suggested to implement a multi-packet IDS is to use a colored Petri Net to model multi-packet signatures. Systems such as IDIOT [32, 31, 33] use this concept to express signatures developed from multiple packets to detect intrusions. This approach has two advantages. First, it allows the signature language to easily model

parallelism and synchronism between packets. The colored Petri Net also allows linking packets together to create a pattern representing an intrusion signature. However, parallelism and synchronism are not paradigms that are necessary to monitor a linear sequence of packets since most attacks can be described without these concepts. The examples provided by the authors did not convince us that parallelism and synchronism are required to specify attack scenario in the case of intrusion detection. Hence, it seems that a more appropriate and simpler solution than Petri Nets can be used to specify attack scenario on a sequence of packets.

P-BEST (Production-Based Expert System Toolset) [35, 1, 2] is a tool set that can be used to develop expert systems. This tool set has also been used to develop intrusion detection systems. However, even if several characteristics of an expert system such as P-BEST may be very interesting for intrusion detection, we believe that such systems do not provide the proper paradigm to use for a real-time multi-packet and context-based IDS. In particular, P-BEST uses forward-chaining to infer new information from its knowledge base. The forward-chaining solution has an advantage in terms of execution speed when lookups in the acquired knowledge are involved. However, it needs heavy computations to maintain and update information. In fact, it is complex and time-consuming to manage in real-time the network context and to correlate it with intrusions using this expert system. Thus, this solution is not practical for a real-time monitoring system even if a knowledge base is a very good paradigm to maintain the network context and to infer new information.

BSML (Behavior Specification Monitoring Language) [57] is a language and algorithm used to specify and monitor multi-packet rules expressed in Extended Regular Expression (ERE). The advantage of BSML, compared with the other tools, is that it can capture real-time constraints. For example, it allows to specify that a SYN_ACK packet has to arrive within 2 seconds after a SYN packet. The ERE are compiled and transformed into Extended State Machines to provide a more performing monitoring system. However, there are some properties that cannot be expressed with ERE and to take them into account, its authors had to incorporate principles that are not extensions of their proposed paradigm. Thus, this paradigm alone is not able to completely capture all the aspects needed to develop a multi-packet monitoring engine. Moreover, it does not provide a mechanism to capture the network context.

LogWeaver [50] is a temporal logic intrusion detection system. The approach of the authors is divided into two parts. They first tried to use Linear Temporal Logic (LTL) to write multi-event packet signatures. Then, they realized that they needed Extended Temporal Logic (ETL) to express intrusion detection signatures for attacks. However, this system does not appear to work in real-time and it is not able to acquire the network context. In particular, based on the intrusion detection signatures provided in their article, their intrusion detection language did not allow to infer the network context, was not used in real-time traffic traces and did not capture the real-time delay between packets that is needed in intrusion detection systems.

IDS such as Bro [44] and Snort [59, 23], and VDS such as NeVO [53, 16, 24], use only ad hoc paradigms to correlate intrusion alarms with the network context. Furthermore, their capability to express multi-packet signatures is typically limited, except to some extent for NeVO. They also provide only ad hoc and incomplete techniques to capture and use the network context. For example, Snort is only able to express such signatures using programmed plug-ins such as Flow and Flowbits. Also, it can only gather information about the context of a TCP session and the state of the applications using a programmed preprocessor.

2.2 System Requirements

Based on our review of the literature and on information provided in [12] and [13], we defined a set of requirements that reflects what is needed to develop a passive network monitoring tool able to acquire the full network context that we propose to incorporate into intrusion detection rules. Based on the analysis presented in [13], none of the IDS presented meets all these requirements. In most cases, they are not able to infer network-context information, they do not provide a formal network and packet model and they do not provide an IDS rule specification language independent from the monitoring engine. Thus, to address these gaps, we decided to focus on specifying a formal packet and network model, to propose a formal IDS rule specification language to acquire network context and to express IDS rules with network context.

Moreover, even if most IDS provide a multi-packet engine, most of them are not available to be tested with regards to their effectiveness and performance and they cannot be modified to meet our requirements. Thus, we decided to create our own multi-packet monitoring engine, inspired from the temporal logic approach presented in the previous section. It appears that temporal logic is a paradigm that fits very well attack scenario identification on packet traffic traces because a packet trace is a list of packets and is thus linear. In this section, we present the requirements specifically related to IDS, as well as the requirements for the packet and the network model.

2.2.1 Intrusion Detection System Requirements

Based on our review of the IDS literature, we defined a set of requirements that reflects what network administrators should expect from IDS. We believe that the most relevant criteria needed for the next generation of IDS are the following:

- **An IDS should provide a declarative rule specification language:** The language in which the rules are specified should use a declarative approach. This is because such a feature allows the user to only specify what to monitor. This way, the user does not have to specify how the rules have to be processed by the engine. Moreover, the modification of the rules does not require a modification of the engine.
- **An IDS should provide a rule specification language independent from the monitoring engine:** The independence between the rule specification language and

the monitoring engine allows the IDS to change its monitoring engine without changing the syntax and semantic of the rules.

- **An IDS should provide a multi-packet rule specification language and a multi-packet monitoring engine:** It is important for IDS to provide a specification language powerful enough to specify intrusion scenarios that can only be expressed using a sequence of packets, such as denial of services, port scanning and context-based intrusions.
- **An IDS should provide passive information gathering:** As attacks become more complex, many intrusions will only be confirmed if they are correlated with the full network context. In fact, intrusions are usually associated with a product, such as an operating system, a service, an equipment (router, switch) and a particular configuration. Thus, this information must be captured by the IDS using information gathering rules to incorporate network context in the intrusion rule. This allows reducing the number of false positives by specifying the network context within an intrusion rule.

2.2.2 Packet and Network Model Requirements

Based on our review of the literature on packet and network models that are used for network information management systems and event correlation systems, we have defined a set of requirements that reflects what a packet and a network model need to provide to enable

network context gathering by IDS. We believe that the most relevant criteria needed for a packet and a network model are the following:

- **A packet and a network model should be expressed formally and independently from the information gathering engine:** It is important to formalize the packet and the network model because some correlation systems do not have a formal network model and only offer an ad hoc mechanism. Often the two models are not independent from the information gathering engine and as a result modifications to the engine imply modifications to the packet and the network model.
- **A packet and a network model should be used with a rule specification language:** To define dynamic information gathering rules, it is important for the model formalism to provide a language allowing to express properties on elements in the packet model in relationship with elements in the network model.
- **A packet and a network model should be easily extendable and scalable:** It is important to be able to easily add new network entities to the model to derive new information. This should be done without having to change the rule specification language and the information gathering engine.
- **A packet and a network model should be able to model the different types of information required in a real-time system:** It is important for these models to be able to capture different types of information, such as static and dynamic associations

between elements of the packet and the network model.

Chapter 3

PNMT Approach and Design

3.1 System Approach Overview

To help fill the gaps noted with existing IDS technology, we developed a Passive Network Monitoring Tool (PNMT) that possesses all the desired IDS requirements specified in the previous chapter. This tool uses the Object Constraint Language (OCL) as its rule specification language. The use of OCL provides our passive network monitoring tool with a high-level declarative rule specification language for multi-packet scenarios. It is also able to specify passive information gathering rules for context-based alerting and it provides a specification rule language that is independent from the monitoring engine.

OCL is used to describe constraints on the Unified Modeling Language (UML) [8, 42] model elements. OCL is a standard constraint and query language based on set theory and

on first-order logic. It is part of the UML specification developed by the Object Management Group [40]. Modelers using OCL are able to express additional restrictions or constraints that cannot be expressed using UML diagrams alone. Thus, OCL adds precision to UML diagrams. OCL is used in PNMT as our rule specification language to express how the network context model can be inferred from the packet model.

The approach used to design PNMT includes five components: the packet capturing engine, the packet model, the rule specification language, the network model and the monitoring engine. Figure 3.1 presents the UML collaboration diagram of an object-oriented representation of these components which shows how they interact with one another. Specifically, the packet capturing engine of the approach is represented by the *Sniffer* object. Instances of the packet model are formatted by the *Dispatcher* object. The rule specification language is used to specify to the monitoring engine what to monitor on the network. The *OCLRule* and *OCLRuleCompiler* are objects that enable our tool to manipulate expressions from the rule specification language. The network model of the approach is represented by the *NetworkModel* object. The network monitoring engine is modeled by the *MonitoringEngine* object.

First, a *Sniffer* (our packet capturing engine) captures all the packets it sees on the network and a packet *Dispatcher* formats the information according to our packet model. Second, the compiled OCL rules defined by the administrator are checked against the produced instances of the packet model by the monitoring engine (to be defined), resulting in

the network model being populated (instance creation). Finally, this network model instance can be queried by the administrator to find out information about the network itself and about network problems.

Figure 3.1: PNMT Collaboration Diagram

3.2 Monitored Objects

Before we explain in further detail the way in which we suggest to model packets and a network, we discuss the different characteristics of the information PNMT has to handle. Here, we suggest three criteria to qualify this information: dynamism, computability, and

availability. Then, we describe the network and the packet model.

3.2.1 Qualifying Network Information

A first way of classifying information is to consider its static versus dynamic nature. Static information can be obtained without looking at the traffic transmitting on the network. For example, the fact that Internet Explorer is usually running on a Windows machine is a static information, since we don't have to perform traffic analysis to acquire this knowledge. Conversely, dynamic information concerns what is going on, or what has already been observed, on the currently monitored network. For example, the fact that there is a running FTP session between two given hosts is dynamic information.

Some information can be static in some cases and dynamic in other cases. An example of such information is the operating system of monitored hosts. One could claim that since he has a good knowledge of his network, operating system information can be statically acquired and added to the monitoring system. On the other hand, a monitored network may have a high number of user-administrated hosts, and the network administrator may not know, a priori, what are the installed operating systems. In that case, it would be desirable to be able to dynamically acquire this information. Thus, the classification of information between static and dynamic information is more a matter of how the information is acquired, rather than of what it actually represents.

Dynamic information can be acquired passively or actively. Each approach offers its ad-

vantages and disadvantages. Since passive information gathering techniques do not generate traffic when they are used, they do not disturb normal network operations. However, passive information gathering techniques are restricted to the information that they are able to see. They cannot discover information that does not transpire on the network. For example, an open port that has not been used by any other system will not be found with these techniques. On the other hand, active information gathering techniques are able to discover such information because they send requests to acquire it. Thus, even if the information has not been used by any other network component, active techniques will try to request it to check if it is available. Since these techniques sometimes need to generate a lot of traffic to discover information, one of their disadvantages is that they could disturb normal network operations. Furthermore, they only provide a snapshot of the network each time they are used. Thus, they are not able to capture dynamic network modifications in real-time. Passive network techniques are able to provide such real-time network modifications because their information gathering techniques rely on the packet stream in the network. Thus, a combination of both techniques is preferable for an information gathering system.

A second way of classifying information is to determine whether it is primitive or computed. For example, the association between MAC addresses and IP addresses, or between IP addresses and DNS names, can be considered as primitive information, since it can be acquired (although it may not be the best way to do it) by respectively looking at ARP and DNS reply packets. Once acquired, one may use these two information sources to compute

the association between MAC addresses and DNS names. However, the latter information could also be acquired by monitoring DHCP packets, if they are available. Here again, the difference between primitive and computed information mainly concerns how we get it. The main issue about computed information is the moment at which it should be computed. For instance, we have to be very careful regarding data integrity. Since all information can be subject to change, the removal of acquired information may have an important effect on our knowledge of the system. In such cases, the removal of information should be made *en cascade*, and guaranteeing data integrity in such a context is, if not impossible, a considerable challenge.

A third way of classifying information is to find out whether it is partially or totally available. This is an important feature we have to consider when expressing constraints about the monitored network. For example, under the (rarely met) hypothesis that the monitoring devices are well situated with regard to the network physical topology, it is reasonable to consider that the knowledge of currently active TCP sessions is totally available. On the other hand, if we don't have access to full DNS entries, the availability of name information highly depends on DNS and DHCP circulating traffic, and it may be possible to find some DNS entries for which we do not have any relevant knowledge. The importance of being aware of this fact is that it is then impossible to work under the hypothesis that every host on the network has a DNS name.

3.2.2 Network Model

When designing our network model, we incorporated some of M2D2's relationships as UML associations between classes. We decided to use UML to express the packet model and the network context model for two reasons. First, a UML class diagram provides a formal paradigm that can easily model packets and network context as objects. Second, UML provides a standard object constraint language, OCL, that allows us to express rules in the model. These UML packet and network models, combined with the use of OCL, meet all the requirements proposed in the previous chapter. While the M2D2 model was designed to express relationships between exploits, vulnerabilities and the different systems located in a network, our proposed model is designed to capture the dynamic aspect of the network in the case of intrusion detection. As a result, modifications to the M2D2 model had to be made. In this section, we present our network model, inspired from M2D2, and we show how it can be populated to capture the network context.

3.2.2.1 Network Model Overview

As mentioned above, the M2D2 model is able to express static relationships between exploits, vulnerabilities and products such as operating systems and services. However, it does not use an object-oriented concept to model the network and was not developed to handle dynamic information, as we explained in the previous chapter. As a result, we modified the M2D2 model and created a network model inspired from M2D2 that can be populated with static

Figure 3.2: Network Model

and dynamic network context information. Figure 3.2 presents the network model that we use to capture the network context.

We did not use inheritance for our network model to provide a simple solution for our prototype. However, there are many extensions planned for our network model to allow for the creation of a more complete network model that would require the usage of inheritance. As an example, the *Host* class could be extended to model specific network equipments such as routers and switches. Thus, a *Router* and *Switch* could be incorporated to the model to

capture a specific aspect of these network equipments.

In order to perform real-time network analysis, we need to represent higher-level entities with flexible relationships. For example, it is specified in M2D2 that each host has a name. However, even if that property could be verified by using the network context, it is not necessarily the case that we have this information on hand in the first place. In fact, M2D2 appears to expect that the system using this model has all the information needed to describe the full network context.

In the case of a real-time monitoring system, the availability of all necessary information is not always guaranteed and often the model is populated using only partial information, such as a partition of the associations between the names and the hosts. For that reason, the cardinality constraints we choose are usually less restrictive than the ones specified in M2D2.

The classes in Figure 3.2 can be split into three categories: the host network configuration classes, the host configuration classes and the security information classes. The host network configuration classes represent the network components. The *Host*, the *Interface*, the *IPStack*, the *Session* and *Port* classes are part of this group. The host configuration classes represent how each network component is configured. The *Product* and *Vendor* classes are part of this group. The information security classes represent security information that needs to be managed by the network model, in relation with the host network configuration and the host configuration classes. The *Alarm*, *Exploit*, *Vulnerability* and *Reference* classes are

part of this group.

In particular, the *Host* class models each component communicating on the network. This class includes infrastructure equipment such as routers and switches as well as workstations and servers. It is associated with a set of *Interface* objects that allow hosts to communicate on the network. All interfaces include a set of *IPStack* objects. The *IPStack* class models the configuration of the IP communication stack. This class is associated with *Product* objects that are installed on the system using this IP stack. The *IPStack* class is also associated with a set of *Port* objects that represent, with the *IPStack* objects, *Session* objects. The *Port* class is also associated with a *Product* object to capture information such as the fact that a port is associated with a product (e.g., a Microsoft IIS FTP server) running on a particular port of a particular host.

In the UML class diagram, we added a relationship between products themselves to express relationships between products such as the one between Internet Explorer and Windows operating system. In a context where host configurations are dynamically acquired, this information can be very useful to estimate the odds that an attack on a given host will be successful if we only have one of the dynamic associations between a network component and one of these products. The other product associations are derived using the static relationship between products. Another reason we have found it useful to add this relationship is that it is already available for most of the vulnerabilities listed in the Bugtraq [55] database. In this case, each vulnerability can be identified by one or many references. A reference is

characterized by an organization (CVE, CAN, Bugtraq, Arachnids, Nessus, Snort, etc.) and an identification number unique to that organization.

3.2.2.2 Description of Classes

In this section, we systematically describe the classes of the network model and their attributes.

- **Exploit:** The *Exploit* class represents known exploits that could affect computer systems. The *description* attribute represents a description of the exploit.
- **Vulnerability:** The *Vulnerability* class represents known software vulnerabilities of products running on computer systems. The *consequence* attribute describes the impacts of the vulnerability on products, while the *requirements* attribute describes what is required for this vulnerability to affect a product.
- **Reference:** The *Reference* class links equivalent vulnerabilities together. This class is necessary because equivalent vulnerabilities can belong to different databases. The *id* attribute describes the unique identifier of the vulnerability in the vulnerability database of an organization, while the *organization* attribute provides the name of the organization that identified the vulnerability.
- **Alarm:** The *Alarm* class represents the alarm generated by PNMT. The *time* attribute provides the time when the alarm was raised by the intrusion detection system.

- **Product:** The *Product* class represents all the software products that could be installed in or part of a computer system. In this class, the static variables OPERATING.SYSTEM, FTP, TELNET are examples of type values for the product. The *prodname* attribute provides the name of the product. The *type: String* attribute provides the type of the product (e.g., operating system, FTP server, TELNET server). The *version: String* attribute provides the version of the product.
- **Vendor:** The *Vendor* class represents product vendors. The *name* attribute provides the name of the vendor.
- **Port:** The *Port* class represents the TCP and UDP ports. The static variables CLOSE and OPEN are used to specify the state of the port. The static variables TCP and UDP are used to specify the type of port. The *number* attribute provides the port number. The *state* attribute represents the state of the port (OPEN or CLOSE). The *type* attribute represents the type of port (TCP or UDP).
- **IPStack:** The *IPStack* class represents the IP Stack used by computer systems to send information on the network. The static variables DOWN and UP are used to represent the state of the IPStack. The *dnsServers* attribute provides the list of DNS Servers. The *gateway* attribute represents the gateway used by the IP Stack. The *ipAddress* attribute provides the IP address of the IP Stack. The *mask* attribute provides the network mask of the IP Stack. The *name* attribute provides the list of names for the IP

Stack. The *role* attribute describes the role of the computer (e.g., workstation, switch router). The *state* attribute represents the state of the IP Stack (UP or DOWN).

- **Interface:** The *Interface* class represents the interface of the computer system used by the IP Stack to communicate on the network. The *macAddress* attribute represents the MAC address of the interface.
- **Session:** The *Session* class represents communication sessions between two IP Stacks using two ports. Thus, a session has a source IP Stack and port, and a destination IP Stack and port.
- **Host:** The *Host* class represents the computer systems located on the network.

3.2.2.3 Description of Relationships

In this section, we systematically describe the associations between the classes of the network model.

- **Vulnerability Exploit Association:** This association represents the known exploits that could be used to take advantage of a vulnerability. To model this association, there is an *exploits* attribute in the *Vulnerability* class and a *vulnerability* attribute in the *Exploit* class.
- **Vulnerability Reference Association:** This association represents the known references from different vulnerability databases associated with the vulnerability. Thus,

a *refs* attribute is added to the *Vulnerability* class.

- **Vulnerability Product Association:** This association represents the list of products that are affected by a given vulnerability. Thus, an *affected* attribute is added to the *Vulnerability* class and a *vulnerabilities* attribute is added to *Product* class.
- **Product Product Association:** This association represents the link between products (e.g., the relationship between a service product and an operating system). Thus, a *correlates* attribute is added to the *Product* class.
- **Alarm Exploit Association:** This association represents the known exploits that have triggered an alarm. Thus, an *exploit* attribute is included in the *Alarm* class.
- **Alarm Port Associations:** Two associations are represented here: one is the association between the source port and the alarm; the other one is the association between the destination port and the alarm. The *sourcePort* represents the *Port* of the attacker. The *destinationPort* represents the *Port* of the victim. Thus, *sourcePort* and *destinationPort* attributes are added to the *Alarm* class.
- **Alarm IPStack Associations:** Two associations are represented here: one is the association between the source address and the alarm; the other one is the association between the destination address and the alarm. The *sourceAddress* represents the *IPStack* of the attacker. The *destinationAddress* represents the *IPStack* of the victim. Thus, *sourceAddress* and *destinationAddress* attributes are added to the *Alarm* class.

- **Product Vendor Association:** This association represents the vendor of a product. Thus, a *vendor* attribute is added to the *Product* class.
- **Interface Vendor Association:** This association represents the vendor of an interface. Thus, a *vendor* attribute is added to the *Interface* class.
- **Port Product Association:** This association represents the product running on a given port. Thus, a *daemon* attribute is added to the *Port* class.
- **Session Port Associations:** Two associations are represented here: one is the association between the source port and the session; the other one is the association between the destination port and the session. Thus, *sourcePort* and *destinationPort* attributes are added to the *Session* class.
- **Session IPStack Associations:** Two associations are represented here: one is the association between the source address and the session; the other one is the association between the destination address and the session. Thus, *sourceAddress* and *destinationAddress* attributes are added to the *Session* class.
- **IPStack Port Association:** This association represents the list of ports associated with an IP Stack. Thus, a *ports* attribute is added to the *IPStack* class.
- **Interface IPStack Association:** This association represents the IP Stacks running on an interface. Thus, an *ipStack* attribute is added to the *Interface* class.

- **IPStack Product Association:** This association represents the product running on an IP Stack. Thus, a *configuration* attribute is added to the *IPStack* class.
- **Host Interface Association:** This association represents the interfaces contained in a computer system. Thus, an *interfaces* attribute is added to the *Host* class.

It is also worth noting that the relationships between the different entities can be either dynamic or static. For example, the correlation between products, or the correlation between products and vulnerabilities, is a static relationship since it does not depend on what is occurring on the network. However, since host network configurations and host configurations may be dynamically acquired, the relationship between host network configurations and products may be dynamic. In the next two sections, the network model is described in more detail based on the dynamic and static nature of the information it contains.

3.2.2.4 Dynamic Information

Figure 3.2 defines relationships between host network configurations, host configurations and their security-related information, such as potential vulnerabilities and exploits. In particular, based on information about the host configurations, such as its installed products and its active services, it is possible to know the potential exploits and vulnerabilities that could affect it.

All the information and associations between the classes representing the host network configurations are dynamic information representing the current state of the network. Thus,

Host, *Interface*, *IPStack*, *Session* and *Port* and their associations with one another are dynamic information.

The *Alarm* class represents the alarm generated by PNMT when an intrusion detection rule has been violated (these rules are discussed in section 5.2). Thus, this information is also dynamic, as well as its associations with *Exploit*, *IPStack* and *Port*.

The associations between the host network configuration classes and the host configuration classes are also dynamic. Thus, the associations between *IPStack* and *Product*, *Port* and *Product*, as well as between *Interface* and *Vendor* are dynamic information.

3.2.2.5 Static Information

The other classes of our model (*Exploit*, *Vulnerability*, *Product*, *Reference* and *Vendor*) and their associations with one another represent static information. For instance, the relationship between a product version, a known exploit and a known vulnerability is static. A product will always be vulnerable to a particular exploit if the network administrator does not update it. The recursive association between products is static, as explained in section 3.2.2.3. The relationship between *Product* and *Vendor* models the static relation between a product, such as Windows, and its vendor, Microsoft.

The *Exploit*, *Vulnerability*, *Product*, *Reference* and *Vendor* classes and their associations can be obtained from intrusion detection signature databases and vulnerability databases. This information is presented in Chapter 4.

3.2.3 Packet Model

In the case of passive monitoring systems and intrusion detection on a network, the main information used to perform network context gathering is obtained from traffic and packet analysis. Hence, it is important to properly represent network packets. In M2D2's, as well as in Vigna's model, packets are represented as tuples, where each component corresponds to a protocol field, the last one being the payload. Protocol stacks are modeled by putting a higher protocol header in the lower protocol's payload. Although this is a very natural way to proceed, because of the implications of this approach explained in section 2.1, we have opted for something different for our approach.

3.2.3.1 Packet Model Overview

Figure 3.3 presents the packet model we used to specify intrusions and information gathering rules. We decided to use a model similar to the one presented by Snort developers. The only difference between Snort's model and ours lies in the way we model ICMP unreachable packets. Those packets are specified in RFC 792 as containing the IP header of the undelivered packet plus the first 64 bytes of the higher protocol header. In the data structure used by Snort, only the IP header is maintained and the first 64 bytes of the higher protocol header are simply dropped. Since this information can be useful to perform traffic analysis, we chose to model the undelivered packet as a real (partially available) packet rather than just as an IP header. In this case, the ICMPHeader class is modeled to contain a Packet object in

the case of ICMP unreachable messages. This model can be easily extended to incorporate other protocol headers and it can also automatically change the monitoring language because OCL defines its predicates from the UML class diagram. For example, a class *HTTPHeader* could be added to the model with a composition association with the *Packet* class, as for the other classes. This class could have attributes such as *url* and *version* for the URL used in the packet and the version of the HTTP encoding. Thus, after this modification of the packet model, we would be able to use in our OCL specification language predicates such as *packet.http.url = "www.carleton.ca"*.

An expert in software modeling could also think of another packet model where we would have a *ProtocolHeader* class that every protocol header inherits from. Then, we would specify that a *Packet* object is composed of one or many instances of this class. However, even though this approach seems closer to an object-oriented paradigm, it has an impact on the complexity of the specification of OCL expressions. In particular, because a *Packet* object is composed of *ProtocolHeader* objects, a method to search and find the correct protocol header and a cast operation to access the particular attributes of its subclasses are needed because of this specification. This is not the case with our model. Thus, because this packet model would increase the complexity of the OCL rule specification, we decided to use our proposed packet model.

Since information gathering is dynamically performed on the monitored network, the way we model events (in this case packets) deserves careful attention. The packet model

Figure 3.3: Packet Model

is conceived to allow the set of created instances to be used as a real-time linear trace in order to model temporal properties between packets. Thus, a time value (referred to as timestamp) had to be added to the *Packet* class, presented in Figure 3.3, to order packets when they are created and to be able to express in OCL relationships through time between packets.

3.2.3.2 Description of Classes

In this section, we systematically describe the classes of the packet model and their attributes.

- **ARPHeader**: The *ARPHeader* class represents the ARP protocol header. The ARP

protocol is used to resolve associations between a physical layer address and its network layer address. For example, the ARP protocol is commonly used to resolve the association between a MAC address and an IP address in an IP network. More information on the ARP protocol is available in [45]. The *opcode* attribute represents the option code of an ARP header. The *protocolType* attribute contains the code corresponding to the protocol address type that the ARP protocol is trying to resolve. The *senderHardwareAddress* attribute describes the sender hardware address. The *senderProtocolAddress* attribute describes the sender protocol address. The *targetHardwareAddress* attribute describes the target hardware address. The *targetProtocolAddress* attribute describes the target protocol address.

- **ICMPHeader:** The *ICMPHeader* class represents the ICMP protocol header. The ICMP protocol is used to manage communication errors in a network. More information on the ICMP protocol is available in [48]. The *code* attribute describes the code of an ICMP header. The *identifier* attribute represents the identifier of the ICMP header. The *mask* attribute is used in an ICMP mask request or mask reply to acquire the network mask of a computer system. The *sequenceNumber* attribute represents the sequence number of the ICMP header. The *type* attribute represents the type of ICMP header.
- **TCPHeader:** The *TCPHeader* class represents the TCP protocol header. The TCP

protocol is used to manage a stateful communication on top of the IP protocol. More information on the TCP protocol is available in [49]. The *ack* attribute indicates whether the ACK flag is active or not in the TCP header. The *ackNumber* attribute represents the acknowledged number of the TCP header. The *destinationPort* attribute describes the destination port of the TCP header. The *fin* attribute indicates whether the FIN flag is active or not in the TCP header. The *psh* attribute indicates whether the PUSH flag is active or not in the TCP header. The *reserved* attribute represents the reserved flag of the TCP header. The *rst* attribute indicates whether the RESET flag is active or not in the TCP header. The *sequenceNumber* attribute represents the sequence number of the TCP header. The *sourcePort* attribute describes the source port of the TCP header. The *syn* attribute indicates whether the SYN flag is active or not in the TCP header. The *urg* attribute indicates whether the URG flag is active or not in the TCP header. The *window* attribute represents the window used in the TCP header. The *options* attribute represents the TCP options of the TCP header.

- **EthernetHeader:** The *EthernetHeader* class represents the Ethernet protocol header. The Ethernet protocol is commonly used as the physical protocol layer in an IP network. More information on the Ethernet protocol is available in [27]. The *sourceAddress* attribute represents the source MAC address of the Ethernet header. The *destinationAddress* attribute represents the destination MAC address of the Ethernet header.

- **Packet:** The *Packet* class describes a complete packet that has been sent on the network. The *number* attribute provides the unique number representing the packet. The *timestamp* attribute provides the time when the packet arrived on the network.
- **IPHeader:** The *IPHeader* class represents the IP protocol header. The IP protocol is used to establish communications between computer systems. More information on the IP protocol is available in [47]. The *destinationAddress* attribute represents the destination IP address of the IP header. The *dontFragmentFlag* attribute describes whether the IP packet could be fragmented or not. The *identification* attribute provides the unique identification number representing the IP header. The *sourceAddress* attribute represents the source IP address of the IP header. The *timeToLive* attribute represents the time to live of the IP header. The *typeOfService* attribute represents the type of service used by the IP header.
- **UDPHeader:** The *UDPHeader* class represents the UDP protocol header. The UDP protocol is used to manage a stateless communication on top of the IP protocol. More information on the UDP protocol is available in [46]. The *destinationPort* attribute describes the destination port of the UDP header. The *sourcePort* attribute describes the source port of the UDP header.

3.2.3.3 Description of Relationships

In this section, we systematically describe the associations between the classes of the packet model.

- **Packet ARPHeader Composition:** This composition represents the ARP header contained in a packet. Thus, an *arp* attribute is added to the *Packet* class.
- **Packet ICMPHeader Composition:** This composition represents the ICMP header contained in a packet. Thus, an *icmp* attribute is added to the *Packet* class.
- **ICMPHeader Packet Composition:** This composition represents the packet that caused an error which triggered the creation of the ICMP header. Thus, an *undeliveredPacket* attribute is added to the *ICMPHeader* class.
- **Packet TCPHeader Composition:** This composition represents the TCP header contained in a packet. Thus, a *tcp* attribute is added to the *Packet* class.
- **Packet EthernetHeader Composition:** This composition represents the Ethernet header contained in a packet. Thus, an *ethernet* attribute is added to the *Packet* class.
- **Packet IPHeader Composition:** This composition represents the IP header contained in a packet. Thus, an *ip* attribute is added to the *Packet* class.
- **Packet UDPHeader Composition:** This composition represents the UDP header contained in a packet. Thus, an *udp* attribute is added to the *Packet* class.

3.3 Rule Specification

We used OCL as the information gathering language to specify how to infer the network context from the packet model. In other words, OCL specifies the rules used to populate the network context model based on the packet model. These OCL rules represent the network monitoring rules used to discover the network context as well as to identify intrusions.

As mentioned before, OCL was added to UML because the class diagrams could not express certain constraints that are essential to model the behavior of many systems. In its original specification, UML only allowed constraint annotations in an informal way in the diagram. Thus, OCL now gives the UML diagram a formal and standard way of expressing constraints.

However, OCL has also been used to express rules other than constraints on UML diagrams. In particular, it has been used to monitor in real-time constraints on a program using Aspect-Oriented Programming (AOP) [9], to express rules to translate a program execution trace automatically into a sequence diagram [10] and as a component for Model Driven Architecture [30]. These authors suggest that OCL can be used to express rules to translate information from one model to another. In the case of context-based intrusion detection, we drew on the same idea and used OCL to express how to translate packet headers into network context such as alarms, host network configurations and host configurations.

We used this approach so it can be used for network monitoring in general to infer network context from packets. This way, our passive monitoring system is able to derive from

sequences of packets not only alarms, as a typical IDS could do, but also, by implementing OCL rules, to infer the full network context, such as Operating System (OS), active TCP sessions, application flows, computer configurations and other knowledge useful to acquire the network context. This model allows the user to describe, using OCL, attacks or intrusions that could only be expressed using a sequence of packets such as distributed denial of service attacks.

OCL was conceived to express the constraints on an object diagram, which can be viewed as a snapshot of the program memory taken at a precise moment (for example, before or after a method execution). Hence, it is not straightforward to express in OCL a constraint such as *if I see event e_1 , followed by event e_2* because the relationship *followed by* is not directly supported by an object diagram. In the context of multi-packet passive information gathering and multi-packet intrusion detection, being able to express time relationships between packets is crucial for the monitoring engine as well as for its users. For example, if we want to describe that a packet p_2 has to arrive within 2 seconds of a packet p_1 , this information must be part of the model.

Figure 3.4 is an example of how OCL may be used to infer that a given port is open on a given computer. It describes how a SYN packet and a SYN_ACK packet have to be related to discover that a particular host with a particular IP address in the network has a particular port open. The left-hand side of the *implies* represents what needs to happen on the network and the right-hand side of the *implies* specifies what needs to be in the network model if

the left-hand side is true. The invariant rule states that for any pair of existing packets $p1$ and $p2$, we have a corresponding open port in the network model, if the first packet is a SYN packet ($p1.syn()$), if the other packet is a SYN_ACK packet ($p2.synAck()$), if the packets are part of an opposite TCP flow ($oppositeTCPFlow(p1, p2)$), with fewer than 2 seconds between them ($p2.occurredWithin(p1, 2000)$) then there exists in the network model an *IPStack* object with its *ipAddress* attribute equal to the *sourceAddress* attribute of the *IPHeader* packet of packet $p2$ ($ipStack.ipAddress = p2.ip.sourceAddress$) and there is a *Port* object in its *ipconfig* attribute that is in an open state ($port.state = Port.OPEN$), its type is TCP ($port.type = Port.TCP$) and its port number is equal to the TCP port number of packet $p2$ ($port.number = p2.tcp.sourcePort$).

The rules defined in Figure 3.5 are used to define some properties that could be reused in other rules. The *syn* rule specifies that a TCP packet is a SYN packet if its *syn* attribute is true and if its *ack* attribute is false. The *ack* rule specifies that a TCP packet is a SYN_ACK packet if its *syn* attribute is true and if its *ack* attribute is also true. The *oppositeTCPFlow* rule specifies that a packet $p1$ and a packet $p2$ are in an opposite TCP flow if they are in an opposite IP flow and if each packet source port is the destination port of the other. The *oppositeIPFlow* rule specifies that a packet $p1$ and a packet $p2$ are in an opposite IP flow if each packet source address is the destination address of the other. The *occurredWithin* defines a past temporal operator that captures the real-time and the ordered aspects of monitored packets. In this case, past linear temporal logic operators are defined in OCL

Context Packet

```

inv: Packet.allInstances()->forAll(p1, p2 | p1.syn()
    and p2.synAck() and oppositeTCPFlow(p1,p2)
    and p2.occurredWithin(p1,2000)
implies
    IPStack.allInstances()->exists(ipStack | ipStack.ipAddress =
    p2.ip.sourceAddress and ipconfig.ports->exists(port | port.state =
    Port.OPEN and port.type = Port.TCP
    and port.number = p2.tcp.sourcePort)))

```

Figure 3.4: OCL Formula Modeling an Open Port

on the packet model to express temporal relationships between packets within a sequence. They are used with other passive network monitoring rules to properly infer the information, based on the order in which the packets were captured on the network.

To assess the achievements made in this project in terms, we must go beyond the mere number of implemented rules and take into account whether the implemented rules are signature-based rules or behavior-based rules. While a detailed discussion about these two types of rules and about their impact on the monitoring engine is outside the scope of this thesis, we provide in what follows an overview of the difference between these two types of rules to put into context our achievements and to explain their impact on the number of rules included in the rule database of PNMT.

Signature-based rules are rules such as intrusion detection rules and operating system

Context Packet

```
def: syn(): Boolean = tcp.syn = true and tcp.ack = false

def: synAck(): Boolean = tcp.syn = true and tcp.ack = true

def: oppositeTCPFlow(p1:Packet,p2:Packet): Boolean =
    oppositeIPFlow(p1,p2) and
    p1.tcp.sourcePort = p2.tcp.destinationPort and
    p2.tcp.sourcePort = p1.tcp.destinationPort

def: oppositeIPFlow(p1:Packet,p2:Packet): Boolean =
    p1.ip.sourceAddress = p2.ip.destinationAddress and
    p2.ip.destinationAddress = p1.ip.sourceAddress

def: occurredWithin(p:Packet,t:Integer): Boolean = self.time > p.time and
    ((self.time - p.time) < t)
```

Figure 3.5: TCP/IP Flow Rules

fingerprinting rules that are presented in section 5.2. They capture information based on precise information such as packet field values for operating system fingerprinting rules. Thus, signature-based rules are similar rules that look at the same sequence of packets and at the same packet attributes, the only difference between them being the values that need to be contained in those attributes. Based on the values found in the specific attributes of the left-hand side of the rule, different information needs to be found in the network model. Thus, there are many of those rules because they are in fact looking at particular behavior. For instance, in the case of operating system fingerprinting, there are hundreds of rules that correspond to the two implemented tests presented in Figure 5.2 and in Figure 5.3.

Furthermore, there are many similar rules to these two rules that can be derived for other operating system information.

The behavior-based rules capture the network context information at a higher level than the signature-based rules. These rules are not typically used to check whether certain packet attributes contain particular values. Rather, they look more at relationships between packets and network context information. They do not capture particular behavior as is done by signature-based rules. Instead, they capture standard behaviors, such as the OPEN Port TCP Rule presented in Figure 3.4. Thus, fewer of them are needed to infer the information in the network model.

In this thesis, we presented 5 network context information-gathering rules, 4 sets of utility rules and 6 enhanced intrusion detection rules. In particular, 3 of the network context information-gathering rules, the rules presented in Figure 5.2, Figure 5.3 and Figure 5.5, are signature-based rules and all the other rules are behavior-based rules. Based on the discussion above and given our results, we believe that the sample of rules presented in this thesis is sufficient to prove our approach to improve intrusion detection.

3.4 Design and Implementation

3.4.1 System Design and Implementation Overview

We decided that for the first version of the PNMT prototype, we would develop a framework to program the OCL rules instead of developing a complete system that includes an OCL compiler. This allows us to quickly create a prototype to verify the effectiveness of our proposed solution to reduce the number of false positives found by IDS. The OCL rules were programmed in Java by hand, using a step-scenario framework that we developed. This step-scenario framework allows us to rapidly implement each OCL rule without having to develop an OCL rule compiler, although ideally OCL rules should be transformed automatically in Java classes. Figure 3.6 presents a UML collaboration diagram of the implemented version of PNMT, which is an adapted version of the Figure 3.1. A brief summary of the complexity of the implemented system is presented in Table 3.1.

Components	Number of classes	Line of code
Sniffer	5	582
Dispatcher	15	4212
AbstractScenarioRecognizer	15	1050
NetworkModel	12	2305
NetworkView	5	912

Table 3.1: Summary of the complexity of the code

First, the user specifies the OCL rules that need to be monitored by PNMT. The programmer then translates these rules in Java. To do that he creates subclasses of the *Ab-*

AbstractScenarioRecognizer class for each OCL rule that he has to program. These *AbstractScenarioRecognizer* objects are programmed representation of the OCL rules specified by the user and they are added to the system. The packet capturing engine is a *Sniffer* object that uses the Windows version of the libpcap library included is TCPDUMP [29]. This *Sniffer* captures all packets it sees on the network. This information is then formatted based on the packet model using *Packet* objects and sent to the *Dispatcher* object. Then, these packets are sent to the *AbstractScenarioRecognizer* objects that were created, based on the satisfaction of OCL rules. If the scenario is completed, i.e., an OCL rule has been triggered, the *NetworkModel* is updated. Then, the network model can be queried by the network administrator using the *NetworkView* object.

3.4.2 Monitoring Engine

3.4.2.1 Framework Specification

To specify the rules used to infer the network context based on a sequence of monitored packets, we showed in the previous section that OCL rules use the *forAll* operator on the instances of the packet class. However, a formal and efficient algorithm is needed to monitor packets in real-time to discover alarms and to infer the network context because OCL rules only provide a specification for the monitoring algorithm.

The problem, from the point of view of the monitoring engine, is to seek information in

Figure 3.6: Proposed Design for the PNMT Collaboration Diagram

real-time in an ordered set structure. In particular, for a real-time system such a passive monitoring system, the engine must verify each rule each time a new packet is put into the set. It thus has to combine the current packet with packets previously added into the set. As a result, the engine needs to look at past information to infer the proper network context specified using the OCL rules. Moreover, packets are modeled with a time value and it is possible to see the set of all packet instances as the ordered set of all packets captured so far. These packets are ordered using the time value included in the packet instances.

We developed a monitoring engine framework to implement real-time verification of OCL rules on the network instead of having an OCL compiler with a monitoring engine. However,

these components are not necessary to meet the main objectives of this project, that is, to demonstrate that a passive network monitoring system such as PNMT can be used to passively acquire network context and that it can also be used as an IDS. Furthermore, our prototype allows us to show that the captured network context can be used in intrusion detection rules to reduce the number of false positives.

Figure 3.7: Framework Model

We thus developed an object-oriented framework that we programmed in Java. Figure 3.7 describes the framework that enables us to quickly program each rule specified in OCL. It is important to note that this framework was only developed to program specific OCL information gathering rules such as passive network context gathering rules and intrusion detection rules. The advantage of this framework is that the programmed OCL rules are independent from the monitoring engine. In fact, to program the OCL rules, the programmer only needs the OCL rule specification. He does not need to be aware of the implementation details of the monitoring algorithm. This framework contains three classes: *AbstractScenarioStep*, *AbstractScenario* and *AbstractScenarioRecognizer*.

3.4.2.2 Description of Framework Classes

The *AbstractScenario* class represents the implementation of the OCL rules. This class contains three attributes: *step*, the list of *AbstractScenarioStep* objects contained in the scenario, *packets*, the list of *Packet* objects that match a step of the scenario and an integer value *currentStep* to keep track of the current scenario step if this scenario is not completed.

The *AbstractScenarioStep* class is used to specify each step contained in an OCL information gathering rule. A step in an OCL rule is a group of predicates that have to be true at the same moment in time. Because packets arrive at different moments in time, a step can also be viewed as the group of predicates related to a packet. As an example, let's use the OCL rule for the open port specified in Figure 3.4. When programmed, this rule contains two steps: one for the packet *p1* and one for the packet *p2*. This is because each step has a temporal restriction on the other. Thus, the first step of the *AbstractScenario* programmed using this OCL rule will specify the requested properties needed for packet *p1* to fit the OCL rule. The first step will contain the verification of the TCP field SYN, to check whether *p1* is a SYN packet. If so, the first step is completed. When *p1* arrives, this is the only predicate in this OCL rule that can be verified. All the other predicates can only be verified if packet *p2* arrives. Thus, they are part of the second step of this scenario. The second step specifies the requested properties for packet *p2* and the requested restrictions, based on its relationship with packet *p1*. For instance, in the open port, OCL rule packet *p2* has to be in a TCP flow opposite to its corresponding packet *p1* in the scenario.

The *AbstractScenarioStep* has two attributes: *timeReferencePosition* which identifies the id of its previous *AbstractScenarioStep*, which is contained in the *steps* list; and *deltaTimeMillis* which describes the maximum time in milliseconds that this step has to be completed and the *AbstractScenarioStep*, specified using *timeReferencePosition*. These attributes are there to capture temporal operators, such as *occurredWithin*, used between packet in OCL rules. For example, if we use the example of Figure 3.4, the step that describes *p2* will have its *timeReferencePosition* equal to the id of the *AbstractScenarioStep*, which represents *p1*, and its *deltaTimeMillis* equal to 2000 because we have a predicate *p2.occureWithin(p1, 2000)* in this OCL rule.

The *AbstractScenarioRecognizer* class contains the algorithmic aspects of this framework. This class emulates the monitoring engine to monitor the OCL information gathering rules specified in Java using the *AbstractScenario* class. The *AbstractScenarioRecognizer* has only one attribute, *scenarioInstances*, which contains all the *AbstractScenario* objects that have been instantiated during the monitoring of this rule. Since OCL rules use first order logic, many packets that are captured by the sniffer fit the description of *p1* in Figure 3.4. Thus, an instance of the *AbstractScenario* has to be kept in the *scenarioInstances* list for each packet that fits a step in the scenario. To implement an OCL scenario, we have to extend *AbstractScenarioRecognizer* and program its constructor using the scenario and step classes based on the OCL rule. No method description is required for the actual monitoring of this OCL rule to keep the rule implementation as independent as possible from the monitoring

algorithm. The monitoring engine algorithm is provided through methods in the superclass.

3.4.2.3 AbstractScenario Example

To illustrate the implementation of a rule, Figure 3.8 describes the implementation version in Java of the Open Port OCL rule presented in Figure 3.4.

To use this framework, we first need to create a subclass of *AbstractScenarioRecognizer* which we called *ScenarioTCPPortRecognizer*. As explained previously, the advantage of this framework is that the monitoring algorithm is independent from the scenario specification.

The only programming that has to be done when translating an OCL rule into this framework is that of the different steps of the scenario in the constructor of this class. Thus, to program the Open Port OCL rule specified in Figure 3.4, we have to define two *AbstractScenarioStep*: one for the SYN packet *p1* and one for the SYN_ACK packet *p2*. For each *AbstractScenarioStep*, two methods have to be defined: *matches*, and *packetMatched*.

The method *matches* defines the properties needed by the packet and its relationship properties needed based on other packets. The parameter *firstStep* contains completed steps of the scenario and the parameter *currentPacket* contains the current captured packet. If this packet fits the required properties of this step of the scenario and if the required properties, based on other packets, fit all the previous steps of this scenario, the method returns true; if not, the method returns false and the packet is discarded by the algorithm. This method is programmed based on the predicates before the *implies* operator in the OCL rules.

```

1  public class ScenarioTCPPortRecognizer extends
2      AbstractScenarioRecognizer {
3
4      public ScenarioTCPPortRecognizer(){
5          AbstractScenarioStep syn = new AbstractScenarioStep(0, 0){
6
7              public boolean matches(Packet[] firsts, Packet current) {
8                  TCPHeader tcpHeader = current.getTCPHeader();
9                  if(tcpHeader == null)
10                     return false;
11                  else return tcpHeader.getSynFlag() && !tcpHeader.getAckFlag();
12              }
13
14              public void packetMatched(Packet[] packets) {
15              }
16          };
17
18          AbstractScenarioStep synack = new AbstractScenarioStep(1, 2000){
19
20              public boolean matches(Packet[] firsts, Packet current) {
21                  TCPHeader tcpHeader = current.getTCPHeader();
22                  if(tcpHeader == null)
23                     return false;
24                  else if(!tcpHeader.getSynFlag() && tcpHeader.getAckFlag())
25                     return false;
26                  IPHeader ipHeader = current.getIPHeader();
27                  TCPHeader synTCPHeader = firsts[0].getTCPHeader();
28                  IPHeader synIPHeader = firsts[0].getIPHeader();
29                  return ipHeader.getSAddr() == synIPHeader.getDAddr() &&
30                      ipHeader.getDAddr() == synIPHeader.getSAddr() &&
31                      tcpHeader.getSourcePort() == synTCPHeader.getDPort() &&
32                      tcpHeader.getDPort() == synTCPHeader.getSPort();
33              }
34
35              public void packetMatched(Packet[] packets) {
36                  TCPHeader synTCPHeader = packets[0].getTCPHeader();
37                  IPHeader synIPHeader = packets[0].getIPHeader();
38                  int p = synTCPHeader.getDPort();
39                  String addr = synIPHeader.getDAddr();
40                  System.out.println("Port " + p + " on " + addr + " is open.");
41              }
42          };
43
44          AbstractScenarioStep[] steps = new AbstractScenarioStep[1];
45          steps[0] = syn;
46          steps[1] = synack;
47          super.addAbstractScenario(new AbstractScenario(steps));
48      }
49  }

```

Figure 3.8: Open Port Rule Programmed Using the Framework

The method *packetMatched* defines the information that needs to be inferred if the scenario is completed. This method is programmed based on information contained in the predicates after the **implies** operator in the OCL rules. The parameter *packets* contains all the packets that have matched each step of this particular instance of the scenario. Thus, information from these packets is used to infer the proper information in the network model.

From line 5 to 16, we programmed an *AbstractScenarioStep* object called *syn*. This corresponds to the *p1.syn()* predicate that in fact corresponds to the predicate *p1.tcp.syn = true and p1.tcp.ack = false*. The first argument of the constructor 0 describes that this step does not depend on other steps. The second argument 0 specifies that this step does not have to meet a delay requirement based on a previous step. The method *matches* of this inner class object describes that the packet *currentStep* has to contain a TCP header (*tcpHeader*) with the TCP SYN flag enabled (*tcpHeader.getSynFlag()*) and the TCP ACK flag disabled (*tcpHeader.getAckFlag()*) to match this step of the scenario. In particular, the method *packetMatched* is not needed for this step of the scenario because the occurrence of a *SYN* packet does not complete the scenario.

From line 18 to 42, we programmed an *AbstractScenarioStep* called *synack*. This step corresponds to all the other predicates *oppositeTCPFlow(p1,p2)* and *p2.synAck()* in the Open Port OCL rule that are defined before the **implies** operator. The first argument of the constructor 1 describes that this step relies on the first step of the scenario and the argument 2000 describes that this step has to be completed 2000 milliseconds before the first step of

the scenario. This corresponds to the *p2.occurredWithin(p1,2000)* predicate in the OCL rule.

From line 35 to 41, the method *packetMatched* describes the predicates after the **implies** operator in the OCL rule since this step is the final step of this OCL rule. In this case, we simplify this example by only displaying in a command prompt a string describing the information inferred by the implemented OCL rule when the scenario is completed. In the actual version of PNMT, this information is added to the network model object, so it can be used by other rules and viewed by the user.

From line 43 to 49, *syn* and *synack* *AbstractScenarioStep* are added to the *AbstractScenario* that will be used to monitor the programmed version of this OCL rule. This is done using the supermethod *analysePacket* inherited from its superclass *AbstractScenarioRecognizer*.

3.4.2.4 AnalysePacket Algorithm

The algorithm for the method *analysePacket* is presented in Figure 3.9. Here is a brief description of this algorithm. First, this method receives the next packet captured by the *Sniffer*. Then, for each instantiation of the scenario in the *scenarioInstances* it verifies two possibilities. First, if this *AbstractScenario* is timed out (i.e., the next step did not arrive within the time specified in the scenario step), then the scenario is removed from the *scenarioInstances*. Otherwise, if this packet matches one of the scenario steps, this scenario

```

analysePacket(packet Packet)
  ForAll scenario in scenarioInstances
    If scenario is timed out
      remove scenario from scenarioInstances
    Else
      If packet matches scenario
        newScenario = scenario goes forward using packet
      If scenario is no completed
        add newScenario to newInstances
      Else
        Update Network Model
  add all newInstances to scenarioInstances

```

Figure 3.9: AnalysePacket Algorithm

is duplicated. PNMT keeps copies of this scenario both before and after moving forward to the next step in the scenario. This part of the algorithm is important to find out about all the scenario completion possibilities because another packet could match the same scenario step and could thus lead to another solution to complete the scenario. As a result, by keeping the original scenario before moving forward to the next step, we are able to capture this possibility. Then, if this new scenario is not completed, it is added to the *newInstances* list. If the scenario is completed when the scenario is moved to the next step, the *AbstractScenario* object updates the network model. After all scenarios have been tested with this packet, a new scenario group is added to the *scenarioInstances* so the next packet can be analyzed by the algorithm.

To be useful, PNMT also needs to include a dynamic rule set and static information.

Thus, the network model requires dynamic information acquisition rules, such as IDS signatures with network context, to be defined and it needs to contain static information. To provide PNMT with these data, we analyzed IDS signatures, VDS signatures and vulnerability databases from different sources. In the next chapter, we describe how information already available can be used to populate the static part of the network model and to derive passive network context gathering rules and efficient intrusion detection rules with network context in OCL.

3.5 Discussion about System Requirements

Using OCL provides PNMT with a multi-packet declarative rule specification language independent from the monitoring engine. Thus, the usage of OCL and the multi-packet monitoring framework allow us to address the first three intrusion detection system requirements presented in section 2.2. The first two requirements were to develop an IDS with a declarative rule specification language that needs to be independent from the monitoring engine. The third requirement was that the IDS should also provide a multi-packet rule specification language and a multi-packet monitoring engine.

The addition of the network model used by the monitoring framework provides PNMT with passive information gathering ability, thus meeting the fourth intrusion detection system requirement. In particular, the implemented information gathering rules are presented in section 5.2.

The usage of UML to specify the network model and the packet model and the usage of OCL as our rule specification language allow PNMT to express formally the network model and the packet model independently from the monitoring algorithm if we were to use the approach with the OCL compiler presented in Figure 3.1. Thus, the usage of UML and OCL allows us to address the first two packet and network model requirements presented in section 2.2. The requirements were that the packet and the network model should have no influence on the monitoring engine and that the rule specification language should be used with the packet and the network model. Thus, if one of these models is updated, the rule specification language should also automatically be updated.

The third requirement was to develop easily extendable and scalable packet and network models. Using UML to specify the packet and the network model and using OCL as our rule specification language allows us to meet this requirement. In particular, the addition of a new protocol header in the packet model and of a new entity in the network model only require the implementation of these new data classes in the code. This has no effect on the monitoring engine and on the rule specification language. In fact, the addition of these classes automatically updates the rule specification language because OCL defines its predicates from the UML class diagram.

The fourth requirement was that the packet and the network model could reflect the different types of information required in a real-time system. In particular, we wanted to address the static and the dynamic aspect of such information. As discussed in the next two

chapters of this thesis, the proposed packet and network models do allow PNMT to capture static and dynamic aspects of information, thus allowing to meet this requirement.

Chapter 4

Populating the Network Model

As explained in the previous chapter, most of the network context that needs to be captured to populate the network model can be inferred from the same packet sequence used to identify intrusions. Thus, an IDS with network context acquisition capacity would be able to derive the context of an intrusion, correctly place the attack in this context and thus reduce the number of false positives. However, other information sources than IDS signatures are needed to incorporate the network context in the IDS signatures and to correlate network context with static information such as vulnerabilities.

To develop an IDS rule database with network context for PNMT, and to prove this concept, information from different vulnerability databases is needed. Relationship information from IDS signatures, VDS scripts and information from vulnerability databases need to be analyzed and incorporated into the IDS rules to provide these rules with network context.

In this chapter, we present the results of an analysis on the strength of the relationships between IDS signatures, VDS scripts and vulnerability databases. In particular, we studied the relationships between Snort signatures, Nessus scripts and the Bugtraq vulnerability database. These results are used to determine how this information can help reduce IDS false positives by either populating the network context with static information, adding network context information to the IDS signatures or correlating IDS alarms with VDS.

4.1 Description of Information Sources

To develop PNMT and to be able to include more network context in IDS signatures, the relationships between exploits, vulnerabilities and products are needed. Intrusions are implementations of known vulnerability exploitations. These vulnerabilities are usually associated with a product, such as an operating system, a service, an equipment (router, switch) or a particular configuration. For example, let's assume that a given sequence of packets could exploit a particular vulnerability. This attack could only be successful if these packets are sent to a target that has the proper operating system, if there is an active session between the intruder and the target and if the target is running a particular service.

Some of the information gathering techniques and rules used to populate the dynamic part of the network model are discussed in [14, 15]. However, two key pieces of information are still needed: information sources to add network context to standard IDS rules and information sources to populate the static part of the network model.

Source	Entries	References to	Information Provided	Web site
Snort	2648	Nessus Bugtraq CVE ArachnIDS Mcafee	Exploit characteristics References to vulnerabilities	www.snort.org/rules
Nessus	5955	Bugtraq CVE	How to discover vulnerabilities How to identify products Vulnerability Product Associations	www.nessus.org/plugins
Bugtraq	13092	CVE	Vulnerability descriptions Vulnerability Product Associations	www.securityfocus.com/bid
CVE	9826	Bugtraq ISS FreeBSD Cisco ...	Vulnerability descriptions	www.cve.mitre.org

Table 4.1: Summarized Description of Intrusion and Vulnerability Databases

There are few open IDS signature databases and vulnerability databases providing the information needed for this survey. Nevertheless, efforts to create standardized information sources for intrusion rules and vulnerability descriptions, such as Snort [60], Nessus [52], Bugtraq [56] and CVE [37], make the correlation of events between those systems possible to a certain extent. In this section, we briefly describe these information sources as well as the information needed to integrate network context in IDS signatures and to populate the static network context information in the network model. Each source of information is further described in the following sub-sections, and summarized in Table 4.1.

4.1.1 Snort Rules

Snort [7, 23] is the cornerstone of open source IDS software. It offers a lightweight, efficient and flexible intrusion detection engine with a complete intrusion signature rule set. The Snort 2.3.2 (which is the version we used in this study) signature database provides more than 2648 signatures. Since version 1.8, Snort uses references to other vulnerability databases and to IDS signature databases such as Nessus, Bugtraq, CVE, ArachnIDS and McAfee. Because the ArachnIDS database is old and contains only 555 vulnerabilities and since there are few references from Snort to McAfee, it was decided to only use the Snort references to Nessus, Bugtraq and CVE databases in this survey. The IDS signatures included in these information sources provide the static information on exploit characteristics needed for our network model. Thus, information from Snort signatures and its references provide us with the relationships between exploits and vulnerabilities.

4.1.2 Nessus Scripts

Nessus [3, 4, 5] is an open source Active VDS maintained by Tenable. Tenable also offers a commercial Passive VDS, NeVO [53, 16, 24]. In this study, we used the current Windows version of Nessus: NeWT 2.1. The Nessus script database contains about 5955 scripts that are used to identify vulnerabilities. The Nessus scripts offer references to Bugtraq and CVE. These scripts can be used to define how to discover vulnerabilities and product versions. They can also be used to define relationships between vulnerabilities and products.

4.1.3 Bugtraq Vulnerability Database

Bugtraq [56] is a database of known vulnerabilities. It contains detailed information about the description of the vulnerability, the products they affect, their known exploits and ways to prevent these vulnerabilities. The Bugtraq database contains more than 13 092 vulnerabilities. However, Bugtraq does not provide information on how to discover the vulnerability and it only references CVE.

4.1.4 CVE Dictionary

CVE [37] is a dictionary that assigns to each vulnerability a unique number and a description. The vulnerability number is composed of three parts. The first part is the prefix CAN or CVE for candidate and confirmed vulnerabilities respectively. The second part is the year that the vulnerability was discovered. The last part is a sequential number that identifies vulnerabilities during a year. The CVE dictionary does provide reference numbers to several other vulnerability databases, but it does not provide associations between vulnerabilities and products or between exploits and vulnerabilities. Thus, even if CVE is recognized as a standard dictionary in terms of vulnerability, it does not provide relationships between information included in our network model.

4.1.5 Information Sources Used with the Network Model

As explained in the previous section, these databases can be used for two purposes in PNMT: to populate static information in the network model and to specify enhanced intrusion detection rules with network context to reduce the number of false positives found by IDS.

The static network information for the static classes of the network model can be obtained by using exploit descriptions and vulnerabilities available in the Snort [60], Nessus [52], CVE [37] and Bugtraq [56] databases. The exploits described in Snort as IDS signatures can be linked to vulnerabilities in the Nessus, CVE and Bugtraq databases. The *Vulnerability* objects can be gathered from the CVE and Bugtraq databases. The relationships between vulnerabilities, products and vendors can be gathered from the Bugtraq database. The relationships between the different types of products and their relationships with vendors can also be partially gathered from the Bugtraq database. Table 4.2 describes the information available from these information sources that are used to populate the static part of the network model presented in Figure 3.2.

Four types of OCL passive information gathering rules can be inferred from these information sources: the rules to derive the dynamic associations between *IPStack* and *Product*, *Port* and *Product*, *Alarm* and *Exploit* and the intrusion detection rules that derive *Alarm* objects.

In fact, if we are able to specify rules to capture the dynamic relationship in the network model between *IPStack* objects and *Product* objects, we are also able to define the potential

Class/Association	Information Sources
Exploit	Snort
Vulnerability	Bugtraq, CVE
Exploit Vulnerability Association	Snort
Reference	Bugtraq, CVE
Vulnerability Reference Association	Bugtraq, CVE
Product	Bugtraq
Vulnerability Product Association	Bugtraq
Vendor	Bugtraq
Product Vendor Association	Bugtraq

Table 4.2: Static Information Sources

vulnerabilities for each system in the network.

Intrusion detection rules that infer the relationships between *Port* objects, *IPStack* objects and *Exploit* objects to define an *Alarm* object can also be generated in part from those databases. The Snort intrusion rules provide the exploit characteristics and its references to vulnerabilities provide the network context. With these references, we are able to integrate in the intrusion rules which products are affected by a particular exploit. Thus, the products can be added in the intrusion signatures to capture the fact that this attack can only occur if the system is using this particular product.

Class/Association	Information Sources
IPStack Product Association	Nessus
Alarm	Snort, Nessus and Bugtraq
Port Product Association	Nessus
Alarm Exploit Association	Snort

Table 4.3: Dynamic Information Sources

The relationships between *Exploit*, *Vulnerability* and *Product* provide a more precise context when raising IDS alarms and can thus potentially help reduce the number of false positives. This can be achieved using once again Snort, Nessus and Bugtraq databases. Table 4.3 provides the information sources that are used to derive OCL passive information gathering rule to populate the dynamic classes and associations of the network model presented in 3.2.

4.2 Analysis of Database References

To include adequate network context in intrusion rules and to populate the static part of the network model, the accuracy of the relationships between Snort exploit descriptions, Nessus and Bugtraq needs to be analyzed carefully. First, we analyze the relationships between Snort rules and Nessus, Bugtraq and CVE, to verify how many rules in a given database actually refer to other databases and to which databases these rules are referring to most often.

4.2.1 Analysis of Snort References

We used a custom parser to transfer the rule references so they could be more easily analyzed with a query language. From the statistical analysis presented in Figure 4.1(a) and Figure 4.1(b), we show that nearly half of the references (44%) concern other databases than

Figure 4.1: Distribution of Snort Rule References

Nessus, Bugtraq and CVE or no database at all. However, more than half of the references (53%) pertain to Nessus and Bugtraq and can thus be used to infer intrusion rules with network context or to populate the static objects of the network context. It is also important to note that only 15% of the Snort rules actually refer to all three databases and that most of the rules (respectively 42% and 46%) have references to CVE or to Bugtraq.

4.2.2 Analysis of Snort Relationships

As a result of this preliminary analysis, a more precise examination of Snort rules was warranted to verify the relationships of Snort rules to their Bugtraq references and to verify the relationships of Snort rules to their Bugtraq references through Snort rules references to

Nessus. This is important to create accurate intrusion detection rules with network context. This way, we are able to assess their effectiveness in reducing the number of false positives when they are used with PNMT and IDS.

Figure 4.2: Snort, Nessus and Bugtraq Relationships Diagram

For this analysis, we decided to drop the CVE references. This is because these references are not useful to infer relationships between Snort, Nessus and Bugtraq, since CVE entries only have references to Bugtraq and do not provide more information that can be used in the network model than the information available in these databases. However, missing references between Snort, Nessus and Bugtraq can be inferred using the CVE number.

From Section 4.1 and the previous analysis, we know that Snort has references to Bugtraq and to Nessus and that Nessus has references to Bugtraq. Figure 4.2 presents these relationships. To analyze and identify the strengths of the relationships between Snort, Nessus and Bugtraq, we classified the Snort rule relationships into four disjoint categories. The results of this analysis are presented in Figure 4.3.

Some categories had to be split into subcategories to identify more precisely the accuracy of the relationships and the gaps in the relationships between these database references. Here is a description of the four categories, as well as of the sub-categories, used in our analysis:

Figure 4.3: Distribution of Snort Rule Group References

1. **Snort rules with direct and indirect relationships to Bugtraq:** These Snort rules are the rules for which all relationships are defined in Figure 4.2. These rules have references to Nessus and Bugtraq and their Nessus references have references to Bugtraq. Thus, the direct relationship is the Snort to Bugtraq relationship and the

indirect relationship is the Snort to Nessus to Bugtraq relationship. To validate the accuracy of the relationships leading to Bugtraq using Snort or using the Snort to Nessus to Bugtraq references, subcategories had to be defined. The results of this analysis are presented in Figure 4.3(b). This figure presents the distribution of the rules included in this category, which are defined using five subcategories:

- (a) **The direct references to Bugtraq that strictly include the indirect references to Bugtraq:** These are the Snort rules for which their direct Bugtraq references strictly contain all their Snort to Nessus to Bugtraq references.
- (b) **The indirect references to Bugtraq that strictly include the direct references to Bugtraq:** These are the Snort rules for which their Snort to Nessus to Bugtraq references strictly contain all their direct Bugtraq references.
- (c) **The direct and indirect references to Bugtraq that are disjoint:** These are the Snort rules for which their direct Bugtraq references are totally different from their indirect relationships to Bugtraq.
- (d) **The direct and indirect references to Bugtraq that strictly intersect:** These are the Snort rules for which their direct Bugtraq references have some references that are the same as when using their indirect Snort to Nessus to Bugtraq references.
- (e) **The direct and indirect references to Bugtraq that are the same:** These

are the Snort rules for which their direct Bugtraq references are equal to their indirect Snort to Nessus to Bugtraq references.

2. Snort rules with incomplete relationships that can be inferred: These Snort rules are the rules that are missing one of the relationships defined in Figure 4.2. Thus, the missing relationship has to be inferred using the two other relationships. The results for this analysis are presented in Figure 4.3(c). This figure presents the distribution of the rules included in this category, which are defined using three subcategories:

- (a) **The Snort to Bugtraq relationships that can be inferred:** These are the Snort rules that do not have Bugtraq references, but that have Nessus references that can in turn have references to Bugtraq. Thus, we should be able to infer their Bugtraq references using information provided by the Snort to Nessus and the Nessus to Bugtraq known references.
- (b) **The Nessus to Bugtraq relationships that can be inferred:** These are the Snort rules that have Nessus and Bugtraq references, but for which their Nessus references do not have any references to Bugtraq. Thus, we should be able to infer the Nessus to Bugtraq references using information provided by Snort to Nessus and Snort to Bugtraq known references.
- (c) **The Snort to Nessus relationships that can be inferred:** These are the Snort rules that have Bugtraq references, that do not have any Nessus references,

but for which there are Nessus scripts that refer to the same Bugtraq references.

Thus, we should be able to infer their Nessus references using information provided by Snort to Bugtraq and Nessus to Bugtraq known references.

3. **Snort rules with incomplete relationships that cannot be inferred:** These Snort rules are missing two of the relationships defined in Figure 4.2. Thus, they have either a reference to Bugtraq or to Nessus, but the missing relationships cannot be inferred using the only relationship they have. The results of this analysis are presented in Figure 4.3(d). This figure presents the distribution of the rules included in this category, which are defined using two subcategories:

- (a) **The Snort to Bugtraq relationships that cannot be inferred:** These are the Snort rules that do not have Bugtraq references, that have Nessus references, but for which their Nessus references do not have any references to Bugtraq. Thus, their Bugtraq references cannot be inferred because there is no relationship from Snort to Bugtraq and from Nessus to Bugtraq.
- (b) **The Snort to Nessus relationships that cannot be inferred:** These are the Snort rules that do not have Nessus references, that have Bugtraq references, but for which there are no Nessus scripts that have the same references to Bugtraq. Thus, their Nessus references cannot be inferred because there is no relationship from Snort to Nessus and from Nessus to Bugtraq.

4. **Snort rules without any relationship to Nessus and Bugtraq:** These Snort rules are the rules that do not have any references to Nessus or Bugtraq. In fact, they are missing the two relationships from Snort defined in Figure 4.2. Thus, these rules cannot be improved by adding more network context or be correlated with VDS such as Nessus using their relationships to Nessus and Bugtraq.

It is important to note that in Figure 4.3(a), 47% of the Snort rules do not have any references to Nessus or Bugtraq. Thus, network context information cannot be added to those rules using information provided by Nessus and Bugtraq. Moreover, the alarms from these rules cannot be correlated with VDS events provided by Nessus, nor with vulnerability information provided by Bugtraq. This implies that correlation systems cannot correlate these Snort alarms with Nessus events using the references provided by Snort and Nessus.

Figure 4.3(a) also shows that only 16% of the Snort rules have references to Bugtraq and Nessus. Of this 16%, 71% have the same set of Bugtraq references whether we use the Snort to Bugtraq references or the Snort to Nessus to Bugtraq references. Thus, only about 11.4% of the Snort rules are suitable for adding network context or for use in a correlation analysis with Nessus and Bugtraq databases. It is important to note that 29% of the Snort rules in Figure 4.3(b) present discrepancies, depending on whether we use the direct or indirect relationship to Bugtraq. In fact, 6% of them seem to refer to different Bugtraq vulnerabilities.

Figure 4.3(c) shows that we should be able to infer 79% of the missing relationships

Figure 4.4: Distribution of Nessus References

between Snort and Nessus using the Snort to Bugtraq and Nessus to Bugtraq relationships. However, it is difficult to assess whether those inferred relationships would be accurate because even if we have all the relationships, for 29% of them there seems to be disagreements between the three databases. Nevertheless, from the sample of rules we analyzed, the inferred relationships seemed to be coherent. A more in-depth analysis would be needed to support this conclusion. However, it should be noted that the inferred relationships from Nessus to Bugtraq are not useful to us for adding network context to intrusion rules. We decided to raise this issue only to provide a complete analysis of the Snort rules distribution.

From Figure 4.3(d), we find that most of the missing relationships between Snort and Nessus cannot be inferred using the Snort to Bugtraq relationships. In fact, as described in Figure 4.4, most of the Nessus references refer to Bugtraq and CVE and as explained in the previous section, Snort rules refer primarily to Bugtraq. One could think that CVE references could be used to infer the missing relationships between Snort rules to Nessus and

to Bugtraq entries. However, as described in Figure 4.1(a), only 24% of the Snort rules have CVE references that have either a Nessus or Bugtraq reference. Thus, at most 24% of the 19% (i.e., 4%) of the rule references could be inferred using the CVE references.

The same reasoning applies for the Snort rules without Nessus and Bugtraq references. There are only 3% of the Snort rules without references to Nessus and to Bugtraq that have a reference to CVE. Thus, at most 3% of the 47% (i.e., 1.4%) of the rule references in this situation could be inferred using the CVE references. Thus, to add network context to intrusion detection rules and to perform event correlation using Snort, Nessus and Bugtraq, the CVE references do not provide much more additional relationships between these three databases.

In conclusion, there are still many holes in the associations between the Snort, Nessus and Bugtraq databases. Thus, even if we use this information to populate the static part of the network model, there will still be many missing associations between the static classes of the network model such as *Exploit* and *Vulnerability*. We conclude that commercial correlation tools can only work to a limited extent if they use information from these databases. However, because of the size of these databases, the number of identified associations derived from this study is still large enough to test and prove the effectiveness of our approach and to allow for the comparison of a sufficient large number of results against those of a standard IDS. We assume that if the importance of the relationships between those databases is proven, as we do in this study, the missing associations will be completed.

In the next chapter, we thus use some of the inferred relationships between Snort, Nessus and Bugtraq to write context-based vulnerability and intrusion rules in OCL that can be included in PNMT.

Chapter 5

Tests and Results

A recent article published in Computerworld [68], written by Andre Lee, the president of NFR Security [51], mentions that a novel approach to intrusion detection would be to develop IDS that could work at a higher level than the ones available today. He explained that IDS could be improved with the help of three techniques: passive operating system fingerprinting, suppressing a flood of alerts and meta-alarms correlation.

Based on our results, we are confident that PNMT is able to help resolve at least two of the three problems addressed by the techniques proposed by Lee: passive network context gathering and meta-alarms correlation. In this thesis, we interpreted meta-alarms correlation as the correlation between different information types such as security events and network configurations. It can also be shown that this prototype is not restricted to passive operating system fingerprinting to capture the network context. Based on our first set of results, we

believe that PNMT has the potential to tackle most false positive problems of current IDS by using multi-packet signatures, passive information gathering, context-based alerting, meta-alarms and correlation of events. Moreover, our declarative signature approach using OCL on the packet model and the network model allows the network administrator to write intrusion detection rules and information gathering rules instead of programming them.

In this chapter, we describe some of the OCL rules for network context gathering and context-based intrusion rules that have been implemented in PNMT. This chapter is split into two sections. First, we explain the procedure used to test the programmed version of the OCL rules. Then, we present the OCL rules that we implemented and tested on a real network.

5.1 Test Procedure

To test the implementation of each OCL rule described in the next section, we developed a test strategy. The resulting test procedure had to meet a number of requirements. First, the data used for testing had to reflect real network situations that would trigger each implemented rule. Second, the test procedure had to be easily repeatable if we changed the OCL rule implementation. It was also very important to test the implemented version of these OCL rules on a quiet and controlled network environment to manipulate the network traffic and stimulate properly each component of the network to get the proper behavior. In particular, these components had to mimic a real network so their behaviors could reflect

real network situations.

To meet these requirements when testing the implementation of each OCL rule, we developed a controlled virtual network with VMware 5.0. This virtual network enabled us to test and capture exploit traffic traces as well as normal network traffic to study their behaviors. Thus, we were able to reproduce network traffic that reflects real network situations. Moreover, the virtual machines that compose the network behave exactly as if they were computers of a real network. This allowed us to control the network traffic between computers and to create a complete network within a computer. This virtual network contained clients and servers, which were used as targeted computers, and attack systems, which were used to execute exploits. Different scripts were used to create normal network traffic and attack scenarios. To make the test procedure repeatable, we captured the traffic traces generated by those scripts in TCPDUMP format. Then, we developed a Sniffer test driver that was able to read those files and simulate that these packets were coming from a real network. Thus, we were able to repeat each test after any code modification using these traffic trace files.

Although it is fairly simple, this test procedure was very effective to test our implemented OCL rules. It is important to note that because this version of PNMT is still a prototype and because its only purpose for now is to verify our proposed concept for IDS, only few rule tests were needed. Thus, we only tested whether the implemented rules could be triggered on all traffic traces that contained the proper information. We also tested whether the temporal

properties between packets described in the rule were found in the implementation. It is important to note that the traffic traces, the test driver and the virtual network developed for this test strategy could also be used with the next version of this prototype, as explained in the next chapter.

To test the implementation of each OCL rule, we used the following procedure. First, we generated the appropriate traffic to trigger the tested rule and to capture the corresponding traffic traces. The traffic traces were then included in a test suite corresponding to the rule under test. If the rule only required one packet, two test cases were conducted, one with a traffic trace with the proper packet and one without. If the rule required more than one packet to be triggered, we developed test cases to check what happened when all the packets were present in the traffic trace and other test cases to check what happened when only one packet was missing. Thus, a rule that required three packets have four test cases. Then, for rules that have a temporal restriction between packets, we derived other test cases based on temporal aspects between the packets included in the traffic traces. The test cases varied the packets arrival time values between packets to verify if each test case inferred the proper information in the network model, based on the arrival time of each packet at the Sniffer. These test cases were developed in JUnit. We also tested context-based intrusion detection rules using intrusion traffic traces against a vulnerable and against a non-vulnerable system. These results were compared to Snort to verify if we were able to reduce the number of false positives when using intrusion detection rules with network context, compared to a situation

where standard intrusion detection rules were used.

5.2 Implemented Passive Network Monitoring Rules

In this section, we show how OCL rules can provide PNMT with formal rules to tackle false positives. For the sake of brevity, we focus on passive network context gathering and on context-based alerting for vulnerability detection and intrusion detection.

5.2.1 Passive Network Context Gathering

Commercial and open source IDS are not able to provide network administrators with a language that is flexible enough to allow the network administrator to specify how to infer the network context and to capture intrusions from a sequence of packets at the same time. In this section, we present different passive network context gathering rules developed in PNMT using the framework presented in section 3.4.2. These rules were used to populate the network model. They provide the network model with basic network context information such as open port, open session, products, operating systems. This basic network context information can then be used by other OCL rules such as vulnerability and intrusion detection rules so they can incorporate more network context, thus allowing to reduce the number of false positives.

5.2.1.1 Open Session Rule

Preprocessors such as Flow and Flowbits [7] respectively provide Snort with stateful intrusion mechanisms for TCP sessions and applications. However, even if these preprocessors are indispensable for intrusion detection, they are not flexible enough to ensure proper intrusion detection for different reasons. First, they are programmed as plug-ins and cannot be modified and adjusted to meet the needs of network administrators. In fact, they are only able to capture behaviors that are within the same communication session and they cannot provide a mechanism to capture attacks in different sessions. Second, while Flowbits is able to capture the behavior of application session, it only provides this functionality in an ad hoc manner by setting boolean flags to keep track of the scenario state at a given time.

We believe that the Snort language can be improved by using a declarative specification language such as OCL to express the behavior of these preprocessors. This would allow solving the issues with Snort preprocessors discussed above. In particular, the stateful TCP session contexts can be specified using OCL rule, as described in Figure 5.1. This OCL rule describes that for all packets $p1$, $p2$, $p3$ and $p4$, we have an open TCP session, if $p1$ is a SYN packet, if $p2$ is a SYN_ACK packet, if $p3$ is an ACK packet and if we did not receive a packet $p4$ that is a FIN or a RST packet with the proper relationships between those packets.

There are several relationships that are needed for this rule to be true. First, packet $p1$ and $p2$ need to be in opposite TCP flow (see Figure 3.5) and packet $p2$ needs to arrive within 2 seconds of packet $p1$. The same relationship has to be expressed between $p3$ and

Context Packet

```

inv : Packet.allInstances()->forAll(p1, p2, p3, p4 |
    p1.syn() and p2.synAck()
    and oppositeTCPFlow(p1,p2) and oppositeTCPFlow(p2,p3)
    and p2.occurredWithin(p1,2000) and p3.occurredWithin(p2,2000)
    and not (p4.time > p3.time and (p4.fin() or p4.rst()))
    and (oppositeTCPFlow(p3,p4) or oppositeTCPFlow(p2,p4)))
implies
    Session.allInstances()->exists(sourceAddress.ipAddress =
    p3.ip.destinationAddress and
    destinationAddress.ipAddress = p3.ip.sourceAddress and
    sourcePort.number = p3.tcp.destinationPort and
    destinationPort.number = p3.tcp.sourcePort))

```

Figure 5.1: Open TCP Session Rule

p2. Second, for the session to be open, we must not see a packet *p4* that is a FIN or an RST packet with a time value bigger than *p3* that is contained in the opposite TCP flow of *p3* or *p2*. If all those predicates are true, we can imply that there is an open TCP session in the network model between the source and destination IP Address of packet *p3* by using the source and destination TCP port of packet *p3*. It is important to note that this information could also have been derived using information contained within packet *p2* and packet *p1*.

5.2.1.2 Operating System Inference Rules

It is important for the IDS to be able to passively discover the operating system version of each computer it needs to protect. For example, an exploit on Samba could affect only

FreeBSD systems. This way, the IDS would be able to put the Samba attack in the proper network context at the time of the attack. To implement operating system fingerprinting rules, we developed two operating system inference rules: one for the SYN packet and one for the SYN SYN_ACK packet combination.

We based these operating system fingerprinting rules on the work conducted in [14]. In this work, more than 200 different operating systems were studied to capture their behavior on more than 12 different tests. In particular, we implemented two of these tests in PNMT to capture operating system network context: the SYN packet test and the SYN SYN_ACK packet test. Fingerprinting techniques can be used to detect operating system network context because different people programmed the different TCP/IP of each operating system, thus their implementation is distinct, allowing us to capture a fingerprint for each operating system.

The SYN packet test looks at different fields of the TCP SYN packets. The combination of the values of these fields constitutes a signature that can be associated with a particular group or family of operating systems. More precisely, the SYN packet test we used looks at four fields of a TCP SYN packet to capture its behavior: the TCP option fields, the TCP window size, the IP time to live field and the IP do not fragment flag. Figure 5.2 describes the value needed for a SYN packet to infer that the source IP address of a SYN packet is running either version 4.5, 5.0 or 5.1 of FreeBSD.

The packet *p1* has to be a SYN packet, its window size must be equal to 65535, it has

Context Packet

```

inv: Packet.allInstances()->forAll(p1 | p1.syn() and
    p1.tcp.options = "M@1460NW@1NNT" and
    p1.tcp.window = 65535 and
    p1.ip.timeToLive = 64 and
    p1.ip.dontFragmentFlag = true
implies
    IPStack.allInstances()->exists( ipAddress = p1.ip.sourceAddress and
    configuration->exists(prodname = "FreeBSD" and
    type = Product.OPERATING_SYSTEM and
    (version = "4.5" or version = "5.0" or version = "5.1")))

```

Figure 5.2: SYN Packet OS Fingerprinting Rule for FreeBSD 4.5 to 5.1

contain a TCP option field equal to "M@1460NW@1NNT", the time to live must equal 64 and the do not fragment flag must be true. The string describing the option field is used to capture a set option contained in the option field. This string describes that this option field has a maximum segment size equal to 1460, followed by a no option, followed by a window scale equal to 1, followed by two no options and by a timestamp value. More information on this syntax is provided in [14]. If all the values of packet *p1* match the values of this OCL rule, this implies that there is an *IPStack* object in the network model with an IP Address equal to the source IP address of the packet *p1* that contains in its configuration set a product representing a FreeBSD operating system. The version of this FreeBSD System could be 4.5, 5.0 or 5.1. This SYN packet operating system inference rule is thus able to populate the network model with the dynamic associations between the *IPStack* objects and

Context Packet

```

inv: Packet.allInstances()->forAll(p1, p2 | p1.syn() and p2.synAck() and
      oppositeTCPFlow(p1,p2) and
      p1.tcp.options = "M@1460TW" and p2.tcp.options = "M@1460" and
      p2.tcp.window = 0x4470 and p2.ip.timeToLive = 64 and
      ip.dontFragmentFlag = true
implies
      IPStack.allInstances()->exists( ipAddress = p2.ip.sourceAddress and
      configuration->exists( prodname = "FreeBSD" and
      type = Product.OPERATING_SYSTEM and
      (version "3.0" or ...or version = "4.3"))))

```

Figure 5.3: SYN_ACK Packet OS Fingerprinting Rule for FreeBSD 3.0 to 4.3

the *Products* objects.

Most of the information related to network context needs to be expressed using a sequence of packets such as the Open TCP Session rule. Passive Operating System Fingerprinting tools such as POF [69] and Etercap [43] do not use the SYN packet that has generated the SYN_ACK packet in its context when defining the fingerprint of an operating system using a SYN_ACK packet. In cases such as this one, only using the SYN_ACK packet leads to 27 operating system choices running on the computer that sent the SYN_ACK packet, compared to 11 if the SYN packet field is also used to describe the behavior of the operating system.

In particular, Figure 5.3 specifies how an IDS can find that a particular computer is possibly running a certain version of FreeBSD using SYN and SYN_ACK packets. This test

is similar to the SYN packet test. It looks at the same fields in the SYN_ACK packet *p2* as the SYN packet test does, such as the TCP option field, the TCP window size, the IP time to live field and the IP do not fragment flag. However, it also looks at the relationships between the SYN packet *p1* and the SYN_ACK packet. First, the packet *p1* needs to be a SYN packet, the packet *p2* needs to be a SYN_ACK packet and *p1* and *p2* need to be in opposite TCP flows (see Figure 3.5). The important aspect of this rule is that the *p2* TCP option field is influenced by the *p1* TCP option field. In this case, if the *p1* TCP option field is equal to "M@1460TW", the *p2* TCP option field has to be equal to "M@1460" to match this rule. If all the fields of *p1* and *p2* match this rule, then there is an *IPStack* object with an IP address equal to the source IP address of packet *p2* that contains in its configuration set a product representing a FreeBSD operating system. The version of this FreeBSD System could be from 3.0, to 4.3. In particular, this operating system information can be used by the network administrator to query the network model and then to automatically identify all the computers located in the network that have given vulnerabilities, based on their operating system.

5.2.1.3 Product Detection Rules

Commercial and open source IDS only provide a language for intrusion detection. Usually, IDS are not able to provide network administrators with a language that is flexible enough to specify intrusion detection signatures and vulnerability rules. But, by using the static and

Context Session

```
def: sessionOpen(sip : String, dip: String,  
    sport: Integer, dport: Integer): Boolean =  
    Session.allInstances()->exists(sourceAddress.ipAddress = sip and  
    destinationAddress.ipAddress = dip and  
    sourcePort.number = sport and  
    sourcePort.type = Port.TCP and  
    destinationPort.number = dport and  
    destinationPort.type = Port.TCP)
```

Figure 5.4: Session Utility Rules

dynamic part of the network model, it is possible to express rules in OCL to monitor potential vulnerabilities. In this case, the dynamic network context is acquired by PNMT using the passive network context gathering rules presented in previous sections and correlated with static information, such as its associations with known vulnerabilities.

Thus, we developed a parser that automatically populates the static information classes of the network model presented in Figure 3.2. In particular, the *Vulnerability*, *Product*, *Vendor* and *Reference* classes were populated based on the information provided in the Bugtraq database about the vulnerabilities relevant to the intrusion detection rules of Snort. Based on this information, dynamic network context information can be associated with products to identify potential vulnerabilities.

Figure 5.4 describes a *Session* utility rule that is used by other rules to verify if a session is open. This rule returns a boolean expression describing if a particular session is open in

the network model at the time this information is requested.

The utility rule *sessionOpen* verifies if there is an open session between two computers in the network. This rule returns true if and only if there is a *Session* object in the network model with a source IP address *sip*, a destination address *dip*, a source port *sport*, a destination port *dport*, a source port type *TCP* and a destination port type *TCP*.

Context Packet

```

inv: Packet.allInstances()->forAll( p1 |
    p1.data.contains("Microsoft FTP Service.") and
    Session::sessionOpen(p1.ip.destinationAddress, p1.ip.sourceAddress,
    p1.tcp.destinationPort, p1.tcp.sourcePort)
implies
    IPStack.allInstances()->exists(ipAddress = p1.ip.sourceAddress and
    ports->exists(number = p1.tcp.sourcePort and
    state = Port.OPEN and type = Port.TCP and
    daemon.procname = "IIS" and
    daemon.type = Product.FTP and daemon.vendor.name = "Microsoft"))

```

Figure 5.5: Microsoft IIS FTP Server Detection Rule

Figure 5.5 describes a simplified version of the OCL dynamic information gathering rule that allows to identify that a server is using Microsoft IIS FTP server. By using the information inferred from this rule in combination with static information acquired from Bugtraq, we are able to identify that this computer could have several vulnerabilities.

The OCL rule specifies that a server is using Microsoft IIS FTP server if PNMT is able to see in the payload of a packet *p1* the string "Microsoft FTP Service". If this packet is

contained in an open session, PNMT infers that there is in the network model an *IPStack* that has an IP Address *p1.ip.sourceAddress* with a port in its port list that has a number based on *p1.ip.sourcePort*, a state *OPEN*, a type *TCP*, a daemon product name *IIS*, a daemon type *FTP* and a daemon vendor name *Microsoft*. Thus, this OCL dynamic information gathering rule enables PNMT to discover a subset of the relationships defined in Figure 3.2 between *IPStack* and *Product* objects. By using this relationship PNMT is able to identify a potentially vulnerable system.

Thus, the rule in Figure 5.5 enables PNMT to discover all Microsoft IIS FTP Servers in the network. Using this information and the static network context contained in the *Vulnerability* class and the *Product* class, the system is then able to provide the network administrator with all the potential vulnerabilities associated with this type of system and to identify which systems can be vulnerable. Other information such as operating systems context can also be used to discover potential vulnerabilities. In particular, to populate the static network context, we used the information available from Bugtraq, Snort and Nessus databases.

5.2.2 Context-Based Intrusion Detection

In this section, we present the OCL intrusion detection rules with network context that have been implemented in PNMT, using the framework discussed in section 3.4.2. These OCL rules are tested using traffic traces that contain intrusions and attacks against systems that

are vulnerable to these intrusions and attacks, as well as traffic traces that contain intrusions and attacks against systems that are not vulnerable to them.

To assess the potential of PNMT for reducing the number of false positives, we compared its intrusion detection rules with their corresponding Snort rules. We chose Snort 2.3.2 (the latest version available at the time of our study) to test and compare our implementation of OCL intrusion detection rules. To create the traffic traces used for this comparison, we drew on exploits available in the Bugtraq database.

5.2.2.1 Network Context in Intrusion Detection Rules

Product information is one of the key network context information that needs to be added to intrusion detection rules to make intrusion attempt scenarios more specific and thus reduce the number of false positives. One of the important missing information in standard IDS rules is the fact that an exploit will only be successful if the products used by the targeted system are vulnerable to this intrusion attempt.

If we refer to the network model presented in Figure 3.2, an *Exploit* is associated with *Vulnerability* objects that are then associated with *Product* objects. Thus, the *Product* information is crucial for an exploit to succeed when it is used against a targeted system. However, standard IDS rules such as Snort are typically designed to only identify the behavior of the exploit. We show in this section that because of their association through the *Vulnerability* class, integrating exploit identification with product identification can reduce

the number of false positives generated by IDS.

In particular, the operating system and the network services are the most common product information associated with a particular vulnerability. To include operating system versions as network context information in the intrusion detection rules of PNMT, we used the operating system fingerprinting signatures provided in [14]. As presented in the previous section in Figures 5.2 and 5.3, we specified these operating system fingerprinting tests using OCL rules. To derive product information from traffic traces and include them in PNMT, we used information from the Nessus scripts and transformed it in OCL rules, as presented in Figure 5.5.

However, before we could include product information into OCL intrusion detection rules, we first had to modify OCL to allow it to specify such rules. In particular, the standard methods provided for *String* objects in the OCL specification are not powerful enough to express some matching patterns needed for intrusion detection rules. Thus, we introduced the method *match* that verifies if a *String* that has invoked this method matches the regular expression passed to this method as an argument. For example, the predicate *s.match(a*b)* is true if and only if the string *s* starts with a series of *a* followed by only one *b*. The specification used for the regular expression constructs is the same as the one used by Snort, so they could be Perl Compatible Regular Expressions (PCRE) [26] and easily transformed into our framework programmed in Java.

Context IPStack

```

def: hasOS(ip: String n: String, v: String): Boolean =
    IPStack.allInstances()->exists(ipAddress = ip and
    configuration->exists(type = OPERATING_SYSTEM and
    prodname = n and version = v))

def: hasDaemonOnPort(ip: String, i: Integer, t: Integer,
    n: String v: String): Boolean =
    IPStack.allInstances()->exists(ipAddress = ip and
    ports->exists( number = i and
    type = t and daemon.prodname = n
    and daemon.version = v))

def: hasDaemonOnPortBetween(ip: String, i: Integer, t: Integer, n: String
    v1: String, v2: String): Boolean =
    IPStack.allInstances()->exists(ipAddress = ip and
    ports->exists( number = i and
    type = t and daemon.prodname = n and
    daemon.version >= v1 and
    daemon.version <= v2))

def: hasPortInState(ip: String, i: Integer, s: Boolean, t: Integer): Boolean =
    IPStack.allInstances()->exists(ipAddress = ip and
    ports->exists( number = i and type = t and state = s))

```

Figure 5.6: IPStack Utility Rules

5.2.2.2 Utility Rules

Before we describe our OCL intrusion detection rules, we define the OCL utility rules that are used in our intrusion detection rules to simplify their specifications. Figure 5.6 and Figure 5.7 present these OCL utility rules.

Figure 5.6 describes *IPStack* utility rules that are used by other rules to verify the network

context. These rules return a Boolean expression describing whether a given information is contained in the network model at the time this information is requested.

The rule *hasOS* verifies whether a computer on the network has a specific operating system running. This rule is true if and only if there is an *IPStack* in the network model with an IP address equal to *ip* and with a product in its configuration that corresponds to the operating system with product name *n* and version number *v*.

The rule *hasDaemonOnPort* verifies whether a specific port on a given computer is running a certain daemon. This rule is true if and only if there is an *IPStack* in the network model that has an IP address with a port that has a number *i*, a type *t*, a daemon name *n* and a version number *v* in its port list.

The rule *hasDaemonOnPortBetween* also verifies whether a specific port on a given computer is running a certain daemon. This rule is true if and only if there is an *IPStack* in the network model that has an IP address with a port that has a number *i*, a type *t*, a daemon name *n* and a version number bigger than *v1* and smaller than *v2* in its port list.

The rule *hasPortInState* verifies whether a specific port on a given computer is in an open or closed state. This rule is true if and only if there is an *IPStack* in the network model that has an IP address *ip* with a port that has a number *i*, a type *t* and a state *s* in its port list.

In Figure 5.7, the *logAlarm* utility rule is defined to be used by intrusion detection rules to populate the network model with intrusion alarms. This rule is true if and only if there

Context Alarm

```
def: logAttack(sip: String, dip: String, sport: Integer,
  dport: Integer, t: Long, descr: String): Boolean =
  Alarm.allInstances()->exists(sourceAddress.ipAddress = sip and
  destinationAddress.ipAddress = dip and
  sourcePort.number = sport and
  sourcePort.type = Port.TCP and
  destinationPort.number = dport and
  destinationPort.type = Port.TCP and
  time = t and
  exploit.description = descr)
```

Figure 5.7: logAlarm Utility Rule

is an *Alarm* object with the IP address of its *IPStack sourceAddress* attribute equal to *sip*, with the IP address of its *destinationAddress* attribute equal to *dip*, with the number of its *Port sourcePort* attribute equal to *sport* and which corresponds to a TCP port, with the number of its *Port destinationPort* equal to *dport* and which corresponds to a TCP port, with its *time* equal to *t* and its exploit description equal to *descr*.

These utility rules as well as the *openSession* utility rule described in Figure 5.4 are used in the next sections to verify network context in the OCL intrusion detection rules.

5.2.2.3 Microsoft IIS FTP Connection Status Request Denial of Service

Here we examine one of the vulnerabilities of the Microsoft IIS FTP Server that corresponds to vulnerability 10934 in the Nessus database and to vulnerability 4482 in the Bugtraq

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 21
(msg: "FTP EXPLOIT STAT ? dos attempt"; flow:to_server,established;
content:"STAT"; nocase; pcre:"/^ STAT\s+[\n]*\x3f/smi";
reference:bugtraq,4482; reference:cve,2002073; reference:nessus,10934;
classtype:attempteddos;
sid:1778; rev:7;)
```

Figure 5.8: Snort "FTP EXPLOIT STAT ? dos attempt" Rule

database. An exploit of this vulnerability can cause a denial of service (DOS) on the targeted computer. A distinctive feature of this vulnerability is that it can only be effective on specific versions of the IIS FTP Server. A known exploit of this vulnerability is to use the FTP connection status request concatenated with enough information to cause a buffer overflow, which causes a DOS on the service. Figure 5.8 describes the Snort rule used to discover this DOS intrusion.

In what follows, we briefly describe Snort signature fields to help readers unfamiliar with Snort rules get a broad understanding of the Snort rules specification described in this section. For more detailed information about Snort rules, the reader is referred to [7, 23].

The first part of a Snort rule describes the type of the rule and the source and destination of the information required to trigger the rule. For example, the string *alert TCP \$EXTERNAL_NET any -> \$HOME_NET 21*, used in Figure 5.8, specifies that this rule is an alert rule that looks for TCP packets with a source IP address contained in the *\$EXTERNAL_NET* variable, sent from any source TCP port to any IP address contained in the

Field Name	Description
msg:	Describes the message sent to the user if this rule is triggered.
content:	Describes what string needs to be contained in the packet payload to trigger this rule.
distance:	A content modifier which ensures that there is a minimum number of characters between pattern matches using the content field.
uricontent:	Describes what the content in the URI portion of the packet needs to be.
nocase:	Used to deactivate the case sensitivity in the content field of a rule.
sid:	Describes the identifier of this Snort rule.
pcre:	Describes a Perl Compatible Regular Expression that the payload of the packet has to match to trigger this rule.
flow:	Describes the type of TCP flow that this packet needs to be part of to trigger this rule. The tags <i>to_server</i> and <i>established</i> describe that the packet has to be sent from the client to the server and that it has to be part of an open session between the client and the server.
type:	Describes the class type of the attack.
reference:	Describes the reference to the other information sources that this rule refers to.
rev	Describes the revision number of the rule.

Table 5.1: Snort Signature Field Description

\$HOME_NET variable on destination TCP port 21. The variables have a \$ prefix and are used to describe a list of IP addresses and ports. The list of the Snort signature fields used in the following sections is presented in Table 5.1.

Figure 5.8 describes the Snort rule 1778 that looks for TCP packets coming from the external network from any port that are sent to any home network computer on port 21. If one packet with these characteristics is also part of an open session between a client and a

server, if it is sent from the client to the server and if this packet payload field contains a string that matches the PCRE described by the signature field *pcre*, then Snort sends the message "FTP STAT ? dos attempt" to the network administrator. The reference part of the rule specifies that this rule corresponds to Bugtraq 4482 and Nessus 10934. We should note that we relied on these databases to add more network context to our intrusion detection rule. Using these references thus allowed us to assert that this vulnerability is only present in some versions of IIS FTP Server.

However, Snort will raise an alarm if the attacker tries to use the same attack against an FTP Server running on Linux or FreeBSD operating systems, even if these systems are not affected by this attack. This situation can raise a lot of false positives if the intruder randomly targets several thousands of computers. An IDS that could interpret the context of attacks using its knowledge of the network at the time would give the attack described above a lower priority in cases where it targets systems other than the affected Windows operating systems, compared to a situation where the attack would target a Microsoft IIS FTP Server running on the proper Windows operating system.

In Figure 5.9, a simplified example of an IDS signature, inspired by Snort rules 1778, is described in OCL. Information about the proper network context was added to the rule to capture the specific network context for which this attack can actually occur and thus to reduce the number of false positives generated by this signature.

This rule describes that if PNMT finds a packet *p1* with a string in its data field that

Context Packet

```

inv: Packet.allInstances()->forAll(p1 |
    p1.data.match("/^ STAT\s+[\^ \n]*\x3f/smi") and
    p1.tcp.destinationPort = 21 and
    Session::sessionOpen(p1.ip.sourceAddress,
    p1.ip.destinationAddress, p1.tcp.sourePort,
    p1.tcp.destinationPort) and
    (IPStack::hasDaemonOnPort(p1.ip.destinationAddress,
    p1.tcp.destinationPort, Port.TCP, "IIS", "5.0") or
    IPStack::hasDaemonOnPort(p1.ip.destinationAddress,
    p1.tcp.destinationPort, Port.TCP, "IIS", "5.1"))
implies
    Alarm::logAttack(p1.ip.sourceAddress,
    p1.ip.destinationAddress, p1.tcp.sourePort,
    p1.tcp.destinationPort,
    p1.time, "FTP EXPLOIT STAT ? dos attempt"))

```

Figure 5.9: PNMT OCL "FTP Exploit STAT ? dos attempt" Rule

matches the specified regular expression, if its TCP destination port is 21, if this packet *p1* is part of an open session and if the destination IP address of this packet *p1* is running an IIS FTP Server version 5.0 or 5.1, then this implies that we have an alarm logged in the network model based on the characteristics of the packet *p1* and its message value is equal to "FTP STAT ? dos attempt".

The Snort rule version and the implementation of the OCL intrusion detection rule presented in Figure 5.9 were tested on traffic traces generated by two exploits used against a vulnerable IIS FTP Server 5.0 running on Windows 2000 Server and against a non-vulnerable IIS FTP Server 6.0 running on Windows Server 2003 standard (Table 5.2 describes the

Abbreviation	Description
D	Detected
ND	Not Detected
FP	False Positive
FN	False Negative

Table 5.2: Abbreviations: Result Tables

	Windows 2000 Server with IIS FTP Server 5.0 (vulnerable)			Windows Server 2003 with IIS FTP Server 6.0 (not vulnerable)		
Exploit	Successful	Snort	PNMT	Successful	Snort	PNMT
msftp_dos.pl	Yes	D	D	No	FP	ND
msftp_fuzz.pl	Yes	D	D	No	FP	ND

Table 5.3: Microsoft IIS FTP Status Request Denial of Service Results

abbreviations used in the tables that describe our results).

The results of these tests are presented in Table 5.3. As we can see, the two exploits were successful against the vulnerable IIS FTP Server 5.0 and not successful against the non-vulnerable IIS FTP Server 6.0. Snort was able to identify these intrusions on the vulnerable system, but also identified them on the non-vulnerable system and thus sent the same intrusion message to the network administrator in both situations. On the other hand, PNMT identified this attack and issued an alarm only when it was used against the vulnerable system.

It is debatable whether or not the network administrator would be interested in also seeing the event that describes the attempts against the non-vulnerable system. To address

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 3372
(msg: "DOS MSDTC attempt"; flow:to_server,established;
dsize:>1023; reference:bugtraq,4006; reference:cve,2002-0224;
reference:nessus,10939; classtype:attempted-dos;
sid:1408; rev:10;)
```

Figure 5.10: Snort "DOS MSDTC attempt" Rule

this issue, PNMT is able to provide information about unsuccessful attempts, and it can also assess the success of such attempts, based on the network context. In this study, to simplify the procedure, we assume that the network administrator only wants to know when an intrusion attempt has a chance of being successful. Thus, the network administrator will consider events generated by exploits used on non-vulnerable system as false positives, not intrusions.

5.2.2.4 Microsoft MSDTC Service Denial of Service

Other denial of service vulnerabilities are present on several standard services used by the Windows operating system. For instance, the Microsoft distributed transaction service is vulnerable to large packets sent by intruders. This attack creates a buffer overflow that could trigger a denial of service for this service. Figure 5.10 describes the Snort rule 1408 that is used to identify this attack on the Microsoft distributed transaction service.

This Snort rule looks for TCP packets coming from any computer on the external network

from any port that are sent to any home network computer on port 3372. If one packet with these characteristics is also part of an open TCP session from the client to the server and if the size of this packet is bigger than 1023 bytes, then Snort sends a "DOS MSDTC attempt" (a denial of service attempt on the Microsoft distributed transaction service) to the network administrator. This Snort rule has a reference to Nessus script number 10939 and to Bugtraq vulnerability 4006. Based on these references, we are able to add more network context to this rule to reduce the number of false positives. From the Bugtraq vulnerability database, we were able to find out that only some versions of the Windows operating system, such as Windows NT and Windows 2000, are vulnerable to this attack.

A simplified version of this Snort rule with network context written in OCL is provided in Figure 5.11. The difference between this rule and the one specified in Snort is that the OCL rule will only raise an alarm if it is able to confirm that the destination of the attack has a Windows 2000 or Windows NT operating system, using information from the network model. Thus, this OCL rule describes that an alarm is added to the network model if the payload of a packet *p1* is bigger than 1023 bytes, if there is an active session between the source IP address and the destination IP address on port 3372 of the packet *p1* and if the operating system of the destination IP address of *p1* is either Windows 2000 or Windows NT.

We tested this vulnerability using two different scenarios. First, we tested a scenario on a vulnerable Windows 2000 Server operating system. Then, we used the same exploit

Context Packet

```

inv: Packet.allInstances()->forAll(p1 |
    p1.data.size() > 1023 and
    p1.tcp.destinationPort = 3372 and
    Session::sessionOpen(p1.ip.destinationAddress,
    p1.ip.sourceAddress, p1.tcp.destinationPort,
    p1.tcp.sourcePort) and
    (IPStack::hasOS(p1.ip.destinationAddress,
    "Windows", "NT") or
    IPStack::hasOS(p1.ip.destinationAddress,
    "Windows", "2000")
implies
    Alarm::logAttack(p1.ip.sourceAddress,
    p1.ip.destinationAddress,
    p1.tcp.sourcePort, p1.tcp.destinationPort,
    p1.time, "DOS MSDTC attempt")

```

Figure 5.11: PNMT OCL "DOS MSDTC attempt" Rule

	Windows 2000 Server (vulnerable)			Windows XP Pro (not vulnerable)		
Exploit	Successful	Snort	PNMT	Successful	Snort	PNMT
bid4006.txt	Yes	D	D	No	FP	ND

Table 5.4: Microsoft MSDTC Service Denial of Service Results

on a non-vulnerable Windows XP Pro operating system. In both cases, Snort raised an alarm. However, this exploit is only successful when it is used against a vulnerable Windows 2000 operating system. When we used PNMT, an alarm was raised only in the first case. Table 5.4 presents these results.

We should note that the OCL intrusion detection rule presented in this section and the

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS
(msg: "WEB-IIS ISAPI .printer access"; flow:to_server,established;
uricontent:".printer"; nocase; reference:arachnids,533;
reference:bugtraq,2674;
reference:cve,20010241; reference:nessus,10661;
classtype:webapplicationactivity;
sid:971; rev:9;)
```

Figure 5.12: Snort "WEBIIS ISAPI .printer access" Rule

one presented in Figure 5.9 could be improved by verifying if another computer is able to communicate on this port after this event. If not, it means that the denial of service was successful. In fact, all denial of service intrusion signatures could be completed using this information.

5.2.2.5 Microsoft IIS Server 5.0 .printer ISAPI Extension Buffer Overflow

One of the vulnerabilities of the Microsoft IIS Server 5.0 is the ".printer ISAPI Extension Buffer Overflow", where an unchecked buffer in a dll file could provide code execution. This vulnerability corresponds to the Bugtraq vulnerability 2674 and can be found using Snort rule 971. This vulnerability only affects version 5.0 of the IIS Server running on Windows 2000. For this attack to be successful, an attacker has to send an HTTP .printer request containing approximately 420 bytes in the "Host:" field that will allow the execution of arbitrary code. Figure 5.12 describes the Snort rule used to identify this intrusion.

This rule looks for TCP packets coming from computers located on the external network on any TCP port that are sent to any HTTP Server on any HTTP port. If one packet with these characteristics is also part of an open session from the client to the HTTP server and if it contains the string ".printer" in the URI field, then Snort sends the message "WEB-IIS ISAPI .printer access" to the network administrator. The rest of the signature describes the references to other security information sources, the type of the attack and the revision number. However, Snort may still raise an alarm if this exploit is tried against a non-vulnerable version of IIS Server such as version 5.1 running on Windows XP. Once again, this situation can raise a lot of false positives if this attack is tried randomly on the network. In Figure 5.13, we propose a simplified OCL intrusion detection rule inspired by Snort rule 971 that includes the proper network context for this intrusion to arise in the network.

This rule describes that if PNMT finds a packet *p1* containing in its data field the string ".printer", if the TCP destination port of this packet is 80, if this packet is part of an open session, if there is in the network model a computer with the destination IP address and destination port of the packet *p1* with a daemon running a product with a name equal to "IIS" and a version equal to "5.0", then this implies that we have an alarm object in the network model that is characterized by information from the packet *p1* with its message value equal to "WEB-IIS ISAPI .printer access". To test this rule, we captured traffic from different exploits that were used against a vulnerable IIS Server 5.0 running on a Windows 2000 Server operating system and against a non-vulnerable IIS Server 5.1 running on a

Context Packet

```

inv: Packet.allInstances()->forAll(p1 |
    p1.http.url.contains(".printer") and
    p1.tcp.destinationPort = 80 and
    Session::sessionOpen(p1.ip.sourceAddress,
    p1.ip.destinationAddress, p1.tcp.sourcePort,
    p1.tcp.destinationPort) and
    IPStack::hasDaemonOnPort(p1.ip.destinationAddress,
    p1.tcp.destinationPort, Port.TCP, "IIS", "5.0")
implies
    Alarm::logAttack(p1.ip.sourceAddress,
    p1.ip.destinationAddress,
    p1.tcp.sourcePort, p1.tcp.destinationPort,
    p1.time, "WEB-IIS ISAPI .printer access")

```

Figure 5.13: PNMT OCL "WEB-IIS ISAPI .printer access" Rule

	Windows 2000 Server with IIS Server 5.0 (vulnerable)			Windows XP Pro with IIS Server 5.1 (not vulnerable)		
Exploit	Successful	Snort	PNMT	Successful	Snort	PNMT
iishack2000.c	Yes	D	D	No	FP	ND
jill.c	Yes	D	D	No	FP	ND
iiswebexplt.pl	Yes	D	D	No	FP	ND
iis5hack.zip	Yes	D	D	No	FP	ND
iis50_printer_overflow.pm	Yes	D	D	No	FP	ND
sol2k.c	Yes	D	D	No	FP	ND
iis_printer_bof.c	Yes	D	D	No	FP	ND

Table 5.5: Microsoft IIS 5.0 .printer ISAPI Extension Buffer Overflow Results

Windows XP Pro operating system.

Table 5.5 presents the results of this experience. As we can see, all exploits were successful

against IIS Server 5.0 and not successful against IIS Server 5.1. Snort rule 971 and the implemented version of our intrusion detection rule presented in Figure 5.13 were used with PNMT on all the traffic traces generated by all the exploits against the vulnerable and non-vulnerable systems. Snort identified all exploits against the vulnerable and non-vulnerable systems, but was not able to find that the attack against IIS Server 5.1 would not have been successful, since this version is not vulnerable. On the other hand, the PNMT rule only triggered an alarm only when the exploits were used against the vulnerable IIS Server 5.0 system.

5.2.2.6 Sendmail Header Processing Buffer Overflow

One of the vulnerabilities of Sendmail is the Sendmail Header Processing Buffer Overflow. This vulnerability allows an attacker to gain control of the affected server. In particular, version 5.2 to version 8.12.7 are affected by this vulnerability. This vulnerability corresponds to the Bugtraq vulnerability 6991 and can be identified using Snort rule 2087, 2259, 2260, 2261, 2263, 2265, 2267 and 2269. For the exploits used in this study, only rule 2087 was triggered. However, the other Snort rules are very similar to Snort rule 2087 so to keep this study simple, we only focused on this rule. It is however very easy to extrapolate the ideas presented in this section and apply them to test Snort rules 2259, 2260, 2261, 2263, 2265, 2267 and 2269. Figure 5.14 describes the Snort rule 2087 used to identify this Sendmail intrusion.

```
alert tcp $EXTERNAL_NET any -> $SMTP_SERVERS 25  
(msg: "SMTP From comment overflow attempt";  
flow:to_server,established;  
content:"From|3A|" ; nocase;  
content: " <><><><><><><><><><><><><><><><><><>" ;  
distance:0; content"|28|" ; distance:1; content:"|29|" ;  
distance:1; reference:bugtraq,6991; reference:cve,2002-1337;  
reference:url,www.kb.cert.org\vuls\id\398025;  
classtype:attempted-admin; sid:2087; rev:8)
```

Figure 5.14: Snort "SMTP From comment overflow attempt" Rule

This rule looks for TCP packets coming from the external network from any port that are sent to any SMTP server on port 25. If one packet with these characteristics is also part of an open session from the client and to the SMTP server, if this packet payload matches the pattern specified using the *content* and *distance* Snort fields, then Snort sends the message "SMTP From comment overflow attempt" to the network administrator. However, versions produced after Sendmail 8.12.7 are not vulnerable to this attack, such as version 8.12.8 running on FreeBSD 4.8 or Microsoft SMTP Server 5.0 running on Windows 2000. Figure 5.15 presents a simplified OCL intrusion detection rule inspired by Snort rule 2261 that includes the proper network context for this attack to be successful in a network.

This rule describes that if PNMT finds a packet $p1$ with the proper pattern in the packet data field, if its TCP destination port equals 25, if this packet $p1$ is part of an open session and if for the destination IP address and the destination port of the packet $p1$ there is,

	FREEBSD 4.3 with Sendmail 8.11.3 (vulnerable)			FREEBSD 4.8 with Sendmail 8.12.8 (not vulnerable)		
Exploit	Successful	Snort	PNMT	Successful	Snort	PNMT
linux86_sendmail.c	Yes	D	D	No	FP	ND
bysin.c	Yes	D	D	No	FP	ND

Table 5.6: Sendmail Header Processing Buffer Overflow Results

vulnerability 6991 and of the implemented version of the OCL rule presented in Figure 5.15. These exploits were successful against the vulnerable Sendmail version running on a FreeBSD 4.3 operating system and had no effect against the non-vulnerable Sendmail version running on a FreeBSD 4.8 operating system. Snort was able to identify all the intrusions but it was not able to identify the failure of these exploits against the non-vulnerable system. The PNMT rule identified correctly the intrusions against only the vulnerable system. As mentioned before, since other Snort rules are triggered by these two exploits, network context information could also be used to reduce the number of false positives generated by those rules for this intrusion.

5.2.2.7 Microsoft DCOM RPC Interface Buffer Overflow

One of the most widespread vulnerability discovered on Windows operating system is the DCOM RPC Interface Buffer Overflow. Many viruses have used this vulnerability to infect Microsoft operating systems. Most of the Windows operating systems are affected by this vulnerability, such as NT, 2000, XP and 2003. It seems that only the 95, 98 and ME version

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 135
(msg: "NETBIOS DCERPC ISystemActivator path overflow attempt
little endian unicode"; flow:to_server,established;
content:"|05|"; within:1; byte_test:1,&,16,3,relative;
content: "|5C 00 5C 00|"; byte_test:4,>,256,-8,little,relative;
flowbits:isset,dce.isystemactivator.bind; reference:bugtraq,8205;
reference:cve,2003-0352; reference:nessus,11808;
reference:url,www.microsoft.com/technet/security/bulletin/MS03-026.mspx;
classtype:attempted-admin; sid:2351; rev:10;)
```

Figure 5.16: Snort "NETBIOS DCERPC ISystemActivator path overflow attempt little endian unicode" Rule

are not affected. This problem results from an insufficient bound check on the DCOM object activation request. This can allow code execution on the system targeted by an intrusion. This vulnerability corresponds to the Bugtraq vulnerability 8205 and it can be identified using Snort rule 2351. Figure 5.16 describes the Snort rule 2351 used to identify this intrusion.

This rule looks for TCP packets coming from the external network on any port that are sent to any home network computer on port 135. If one packet with these characteristics is part of an open session from the client to the server, if this packet payload matches the pattern defined by the *content* and *byte_test* Snort fields and if the *dce.isystemactivator.bind* is set, then Snort sends the message "NETBIOS DCERPC IsystemActivator path overflow attempt little endian unicode" to the network administrator. The rest of this rule describes the references to the other information sources, the type of this attack and the revision

number. This rule relies on the FlowBit plug-in to derive the proper network context at the NETBIOS application layer. However, this rule does not rely on operating system network context to validate the success of this intrusion, even if this vulnerability is dependent on the running operation system. We thus compared this Snort rule using the FlowBit plug-in with our PNMT version using operating system network context. We should note that we had to develop a new method in the *Packet* class called *snortMatch* to specify and implement this Snort rule using OCL. This method requires the Snort fields that are used to match information in the packet payload because we discovered that more pattern matching techniques were needed for this specific intrusion detection rule than those provided by regular expressions. To simplify our work and to be able to test these types of rules with network context, we decided to use the Snort fields for this project. In Figure 5.17, we propose an OCL intrusion detection rule inspired from Snort rule 2351 that includes the proper network context for this intrusion to be successful in a network.

This rule describes that if PNMT finds a packet *p1* matching the pattern found in the Snort rule, if its TCP destination port is 135, if this packet *p1* is part of an open session and if the destination IP address of the packet *p1* is running Windows 2000, Windows NT, Windows XP or Windows Server 2003, then that implies that we have an alarm object in the network model that is characterized by the packet *p1* with a message value equal to "NETBIOS DCERPC ISystemActivator path overflow attempt little endian unicode". To test this rule, we captured traffic from several exploits used against a vulnerable Windows

Context Packet

```

inv: Packet.allInstances()->forAll(p1 |
  p1.data.snortMatch(content:"|05|"; within:1;
  byte.test:1,&,16,3,relative;
  content: "|5C 00 5C 00|";
  byte.test:4,>,256,-8,little,relative;) and
  p1.tcp.destinationPort = 135 and
  Session::sessionOpen(p1.ip.sourceAddress,
  p1.ip.destinationAddress, p1.tcp.sourcePort,
  p1.tcp.destinationPort) and
  (IPStack::hasOS(p1.ip.destinationAddress,
  "Windows", "NT") or
  IPStack::hasOS(p1.ip.destinationAddress,
  "Windows", "2000") or
  IPStack::hasOS(p1.ip.destinationAddress,
  "Windows", "XP") or
  IPStack::hasOS(p1.ip.destinationAddress,
  "Windows", "2003"))
implies
  Alarm::logAttack(p1.ip.sourceAddress,
  p1.ip.destinationAddress,
  p1.tcp.sourcePort, p1.tcp.destinationPort,
  p1.time, "NETBIOS DCERPC ISystemActivator path overflow attempt
  little endian unicode")

```

Figure 5.17: PNMT OCL "NETBIOS DCERPC ISystemActivator path overflow attempt little endian unicode" Rule

2000 Server operating system and against a non-vulnerable Windows 98 SE operating system.

Table 5.7 presents the results. All the exploits were successful on the vulnerable operating system and not successful on the non-vulnerable operating system. Snort rule 2351 and the implemented version of the OCL rule presented in Figure 5.17 were used against all the

	Windows 2000 Server (vulnerable)			Windows 98 SE (not vulnerable)		
Exploit	Successful	Snort	PNMT	Successful	Snort	PNMT
dcomrpc.c	Yes	D	D	No	ND	ND
dcom.c	Yes	D	D	No	ND	ND
DComExpl.UnixWin32.zip	Yes	D	D	No	ND	ND
07.30.dcom48.c	Yes	D	D	No	ND	ND
30.07.03.dcom.c	Yes	D	D	No	ND	ND
0X82dcomrpc_usemgret.c	Yes	D	D	No	ND	ND
oc192dcom.c	Yes	D	D	No	ND	ND
kaht2.zip	Yes	D	D	No	ND	ND
rpc!exec.c	Yes	D	D	No	ND	ND
msrpc_dcom_ms03_026.pm	Yes	D	D	No	ND	ND

Table 5.7: Microsoft DCOM RPC Interface Buffer Overrun Results

traffic traces generated by these exploits against the vulnerable and non-vulnerable operating systems.

In this case, PNMT and Snort were able to identify all the attacks against the Windows 2000 Server and were also able to identify that the attacks against Windows 98 SE were not successful. The results of using these rules are thus the same, but for different reasons. First, in the case of Snort, the plug-in FlowBit also used two other Snort rules that set the bit *dce.isystemactivator.bind* that is needed to trigger the alarm. These other rules look at the previous steps in the RPC session to put this rule in the proper application session context. Thus, the session application context was enough to discover that the attacks against Windows 98 SE were not successful. In the case of the PNMT rule, it was the operating system context that was used to discredit the attack against Windows 98 SE. However, the application session state that FlowBit uses to characterize the intrusion should

also be included in our OCL rule. It would also be possible to capture this network context through an intrusion detection rule by specifying the RPC application state in the same way we captured the TCP session state in Figure 5.1. To do that, we would have to add an RPC packet header in our packet model and to define the proper application state using an OCL network context information gathering rule. Then, this network context could be added to our OCL intrusion detection rule.

We believe that this Snort rule, even with FlowBit, can still raise false positives if we ran the same exploits against future versions of the Windows operating system. This is because it seems that FlowBit allowed this Snort rule to be successful only because in the second part of the application state, the Windows 98 SE was not able to process the first request correctly. The likely explanation for this is that the requested feature is not available in previous Windows operating systems. We think that future Windows operating systems will still offer this feature, even though they will not be vulnerable to these exploits. Thus, the operating system network context should still be a key component to describe attacks against future non-vulnerable Windows operating systems.

In a number of situations, network context other than the application state is very important to capture false positives. In fact, some exploit attempts can be used against many types of systems. In some of these situations, the system reaction would not provide any information to the IDS allowing to determine whether the attempt was successful. One solution to this problem is to know on what type of system the exploit was executed and to check

if this specific system is vulnerable to it. Our next OCL intrusion detection rule illustrates such a situation, where the exploit attempt can be used against a variety of systems. In this case, the network context can provide information to determine whether the attempt was successful. It is probably in such cases that network context is most useful to reduce the number of false positive generated by these intrusions.

5.2.2.8 Wu-ftp File Globbing Heap Corruption

One of the well-known vulnerabilities against the WU-FTP server is the Wu-ftp File Globbing Heap Corruption vulnerability. This vulnerability results in several flaws in the code used for the file globbing. The file globbing code contains a heap condition vulnerability that allows attackers to remotely execute arbitrary code on a server. This vulnerability corresponds to the Bugtraq vulnerability 3581 and it can be identified using Snort rules 1377 and 1378. These two Snort rules are very similar. One looks for the "[" character and the other one looks for the "{" character in the packet data field. Figure 5.18 describes the Snort rule 1377 that is used to identify this vulnerability.

This rule looks for a TCP packet coming from the external network from any port that is sent to any home network computer on port 21. If this packet is part of an open session from a client to the server and if this packet contains the string "~" and the string "[", where these strings are at least one character apart from one another, then Snort sends the message "FTP wu-ftp bad file completion attempt [" to the network administrator. However,

```
alert TCP $EXTERNAL_NET any -> $HOME_NET 21
(msg: "FTP wu-ftp bad file completion attempt ["; flow:to_server, established;
content: "~"; content: "["; distance:1;
reference: bugtraq,3581; reference: cve, 2001-0550
classtype:misc-attack; sid:1377; rev:14;)
```

Figure 5.18: Snort "FTP wu-ftp bad file completion attempt [" Rule

Snort does not check the version and the type of the FTP server. If an exploit that triggers this rule was used against a non-vulnerable WU-FTP server or another FTP Server such as an IIS FTP Server, Snort would still send an alarm to the network administrator. In Figure 5.19, we present an OCL intrusion detection rule inspired from Snort rule 1377 and 1378 that includes the proper network context for such an intrusion to be successful.

This rule describes that if PNMT finds a packet $p1$ with a data field that matches the regular expression " $\sim.*[$ ", if this packet $p1$ is part of an open session and if, based on the destination IP address and the destination port of this packet, a computer is running a daemon with a product name equal to "WU-FTP" and a version number equal to 2.5.0, 2.6.0 or 2.6.1, then this implies that we have an alarm object in the network model characterized by information contained in the packet $p1$ and its message value equals to "FTP wu-ftp bad file completion attempt [". To test this rule, we used an exploit against a vulnerable 2.6.1 WU-FTP Server running on RedHat Linux 7.1 and a non-vulnerable IIS FTP Server running

Context Packet

```

inv: Packet.allInstances()->forAll(p1 |
    p1.data.match ("~.*[") and
    p1.tcp.destinationPort = 21 and
    Session::sessionOpen(p1.ip.sourceAddress,
    p1.ip.destinationAddress, p1.tcp.sourcePort,
    p1.tcp.destinationPort) and
    (IPStack::hasDaemonOnPort(p1.ip.destinationAddress,
    p1.tcp.destinationPort, Port.TCP, "WU-FTP", "2.5.0") or
    IPStack::hasDaemonOnPort(p1.ip.destinationAddress,
    p1.tcp.destinationPort, Port.TCP, "WU-FTP", "2.6.0") or
    IPStack::hasDaemonOnPort(p1.ip.destinationAddress,
    p1.tcp.destinationPort, Port.TCP, "WU-FTP", "2.6.1"))
implies
    Alarm::logAttack(p1.ip.sourceAddress,
    p1.ip.destinationAddress,
    p1.tcp.sourcePort, p1.tcp.destinationPort,
    p1.time, "FTP wu-ftp bad file completion attempt [")

```

Figure 5.19: PNMT OCL "FTP wu-ftp bad file completion attempt [" Rule

	RedHat Linux 7.1 with wu_ftp 2.6.1 (vulnerable)			Windows 2000 Server with IIS FTP 5.0 (not vulnerable)		
Exploit	Successful	Snort	PNMT	Successful	Snort	PNMT
bid3581.txt	Yes	D	D	No	FP	ND

Table 5.8: Wu-Ftpd File Globbing Heap Corruption Results

on Windows 2000 Server.

Table 5.8 presents the results. As we can see, this exploit was successful against the vulnerable system and unsuccessful against the non-vulnerable system. At first, both Snort

and PNMT were not able to identify this exploit. It turns out that the minimum distance of one character between "~" and "[" does not seem to be required for this attack to be successful. Thus, we modified the relevant Snort and PNMT rules accordingly. The modified Snort rule was able to identify this vulnerability, but it also identified it against the non-vulnerable system. PNMT identified the intrusion only on the vulnerable system. This exploit is a good example of a very simple exploit that is very dangerous and that can generate a lot of false positives.

Other vulnerabilities are still being tested and analyzed using Snort rules. Based on the results presented in this chapter, we have decided to continue this work with a larger set of Snort rules. However, to improve the efficiency of testing and development of OCL rules, some modifications to PNMT are needed. In the next chapter, future work aiming to develop a larger rule set and an automatic code generator for OCL rules is discussed.

Chapter 6

Future Work

In this chapter, we present some of the work that we plan to conduct to expand the work completed in this thesis. More specifically, two projects are planned: the specification and automatic generation of information gathering rules and the design and development of a first-order past linear temporal logic monitoring engine. Since more information gathering rules are needed to capture network information and to provide intrusion rules with network context, we plan to increase the library of rules included in PNMT and to develop a network infrastructure to test them rapidly. Also, a compiler and an engine are needed to allow PNMT to compile the OCL rules into executable code that could be used by a monitoring engine. This solution implies getting rid of the framework presented in Figure 3.6 and replacing it with a compiler and a first-order past linear temporal logic monitoring engine.

6.1 Information Gathering Rules

In Section 4.1 we used the static information provided by Snort, Nessus and Bugtraq databases to derive the missing dynamic information needed to passively gather the network context. As a result of this work, another library of approximately 25 network configuration inference rules is under development. These rules allow PNMT to gather information on different aspects of network components. They are classified into different groups: Host Discovery, Operating System Identification, Protocol Discovery, Service Discovery, Switch and Router Discovery, and IP Host Configuration. In the case of Operating System Identification rules, we have so far implemented only 2 of the 12 rules proposed in [14]. This report lists 12 rule types for operating system fingerprinting, capturing over 200 different operating systems. Thus, we plan to try to incorporate the 12 rule types in a new library of rules for OS fingerprinting. This information is essential to allow for the vulnerability and intrusion detection rules of PNMT to verify the network context.

We are also investigating the possibility of developing a tool that could generate automatically all the possible intrusion rules with their appropriate network context, based on the information provided by Snort, Nessus and Bugtraq. This tool would automatically generate a subset of the intrusion rules available in Snort rule set. We also plan to investigate whether we could test all these rules using modified condition coverage to ensure their accuracy and to still be able to add network context based on information provided by Nessus and Bugtraq. The new rules written in OCL would be used and tested on the next version

of our virtual network testbed, developed using VMWare 5.0.

As we explained in the previous chapter, we developed a virtual machine network to test and capture exploit traffic traces to study their behaviors. We plan to study more exploits and develop tools to automate the process of running, capturing and testing traffic traces using Snort and PNMT. The proposed modifications of our testbed will include at least two attacker virtual machines, a Windows 2000 and a Redhat Linux 8.0 virtual machine, with a larger library of exploit programs. These virtual machines will be used to attack a much larger set of virtual machines running vulnerable and non-vulnerable operating systems. Thus, this will give us more data to run the traffic traces generated by those exploits against Snort and other IDS to verify if PNMT is still able to reduce the number of false positive against non-vulnerable systems. Thus, we plan to use this network on a larger scale to test our OCL intrusion detection rules.

6.2 Proposed Modification to PNMT

Many modifications have to be made to PNMT so network administrators can use it in a production environment. In this section, we describe a new version of PNMT that is still at the design stage. This new version of PNMT will still use OCL as its rule specification language. However, it will use the Past Linear Temporal Logic (PLTL) as its paradigm for the monitoring algorithm. With regard to the monitoring engine, we found, as explained later in section 6.2.2, that the real-time implementation of a PLTL algorithm could provide an

efficient monitoring algorithm (because performance is a key aspect of network monitoring) and a low-level declarative language to express network context gathering using multi-packet rules. However, first-order temporal logic is needed for passive monitoring to link events with one another. Thus, we plan to develop a first-order PLTL to be able to monitor the rules expressed in OCL. In the next sections, we present the new PNMT system architecture and a brief description of PLTL and of the proposed monitoring algorithm for PNMT.

6.2.1 System Overview

Our proposed future approach still comprises five components: the packet capturing engine, the packet model, the rule specification language, the network model and the monitoring engine. The main difference with our current approach is at the design and implementation level. More specifically, the difference between the version presented in Figure 3.6 and this new proposed version lies in the OCL rules compiler and in the monitoring engine implementation. In particular, we propose a full implementation of the approach presented in Figure 3.1. This approach implements the *OCLRuleCompiler* object as a compiler to derive, from the OCL rules, first-order past linear temporal logic formula that can be monitored by the *MonitoringEngine* object, which represents the implementation of a PLTL monitoring engine.

Thus, before the system starts capturing packets, the OCL rules will be translated into first-order PLTL rules and compiled so they can be monitored by the monitoring engine in

PNMT. This will be done because OCL provides only a specification language; the OCL rules are high-level rules independent from the monitoring engine and until now all OCL rules had to be programmed by hand for used in PNMT. In our proposed project, these rules will be compiled into low-level PLTL rules so they can be closer to the monitoring paradigm. We will use a PLTL monitoring engine because capturing packets on a network segment reflects exactly the type of models used in temporal logic. Moreover, temporal operators can express within the logic the discovery of patterns from a sequence of packets. By combining OCL and PLTL, we will have a formal high-level signature language based on objects (through OCL) and monitoring will be provided by an algorithm using PLTL. In the following section, we provide a brief description of past linear temporal logic and we describe the proposed algorithm to be used in PNMT.

6.2.2 Past Linear Temporal Logic

From all available monitoring paradigms that could be used to develop a monitoring engine, we decided to select temporal logic based on the survey described in the related work section on multi-packet monitoring engine. However, we still had to choose between different versions of temporal logic. Temporal logic models differ from one another in many aspects. They may use a linear or a branching model; they may use past and/or future temporal operators; and it may be a first-order logic or a propositional logic.

At some moment in history, a wide debate took place on whether one should use linear

versus branching time temporal logic [18, 19]. Since we are only checking a single path, branching temporal logic was not useful in our case. A second difference between temporal logic models is whether their temporal operators allow a reference to future [34, 36], past, or both past and future events [20]. From an expressiveness point of view, it has been shown in [20] that adding past-time temporal operators to future-time temporal logic is useless. On the other hand, it has also been shown in [64] that limiting ourselves to pure-past temporal logic has great computational advantages as the memory size and the execution time needed to monitor a formula are respectively constant and linear in terms of the monitored trace size. From an expressiveness point of view, although we are not aware of any formal demonstration, limiting ourselves to past-time temporal logic should not be a problem since the only verifiable assertions we are interested in are past-time assertions. To verify future-time assertions, you have to wait, and once you are able to decide whether these assertions are true or not, the first thing you notice is that they actually assert something about the past. In this case, because we are modeling a real-time system and because we can only rely on past captured packets to infer the network context, a past linear temporal logic is sufficient to describe the monitoring engine.

A third difference between temporal logics is whether they are propositional or first-order. Most of the temporal logics introduced in the model checking literature are propositional, while most of the temporal logics introduced in the AI literature are first- (if not higher) order. Temporal logics coming from the model checking literature present the advantage

of not only having well-defined syntax and semantics, but also of having been extensively compared to one another by a wide population of researchers both in terms of expressiveness and computational complexity. However, these comparisons are most of the time made only for the propositional case. In fact, temporal logics are usually presented with the propositional case in the literature. Although first-order past linear temporal logics are pretty rare in the model checking literature, there are still some that have been introduced. The most influencing of them, in our case, are TRIO [21] and Eagle [6]. For our project, what we need is a first-order temporal logic. For example, we do not want to state that *attacks should occur during an active session*. What we want to state is that *attacks should occur within all active sessions*. In order to do that, we have to be able to compare the TCP ports and the ACK numbers of concerned packets, and first-order variables allow doing that.

Therefore, we chose to specify the low-level rules of a first-order past linear temporal logic based on the standard propositional past linear temporal logic. First, we decided to use past temporal operators because a real-time monitoring system is only able to rely on past events. Consequently, future temporal operators are not needed. Secondly, a first-order logic is needed because the events within a rule could be linked with one another to express properties. This allows the logic to model the fact that different events must have something in common to satisfy the formula.

To facilitate passive network context gathering, an information gathering temporal operator is added to our first-order PLTL. This allows capturing the intervals for which cer-

tain network context information is true. The temporal operator $[,]$, used in the formula $[\phi_1, \phi_2]$, where ϕ_1 and ϕ_2 are first-order PLTL formulas, describes that this formula is true between the occurrence of ϕ_1 and ϕ_2 . For example, $[sessionOpen, (p.fin \vee p.rst) \wedge sameTCPFlow(sessionOpen, p)]$ describes that a session is active between the time it is opened and the time that a packet p , that is either a FIN or an RST packet, arrives from the same TCP flow as the *sessionOpen* object. Since this temporal operator allows maintaining a real-time knowledge-base (inserting on ϕ_1 and deleting on ϕ_2), we call this operator *information gathering operator* (IGO) and we include it in first-order PLTL. Thus, our logic is now more intuitive and suitable for network context gathering. It is important to mention that an operator similar to our information gathering operator has been used in [25]. The only difference between their information gathering operator and ours is that these authors used inclusive inequalities instead of exclusive inequalities. In fact, these authors demonstrate that the PLTL logic, with only their version of the IGO as a temporal operator and a border operator, is equivalent to the PLTL.

The monitoring algorithm used in our new version of PNMT is inspired from the monitoring algorithm presented in [25] and [64]. Their algorithm is only able to verify formulas written in propositional PLTL and functions on a finite event trace. Despite these restrictions, we decided to use this algorithm as a basis for our monitoring engine because it is able to compile the formulas we want to monitor and it has been proven to be close to the theoretical minimum complexity to monitor PLTL formulas.

Even if OCL is more expressive than first-order PLTL, it is possible to take advantage of both paradigms in the case of passive network monitoring. In fact, the type of OCL formulas needed to express properties for a passive network monitoring system such as PNMT can be translated into first-order PLTL using translation rules because of the way we modeled the packet sequence. By combining both paradigms, we should provide network administrators with a formal high-level signature language based on objects by using OCL and with an efficient monitoring algorithm by using PLTL.

Chapter 7

Conclusion

Intrusion Detection Systems use different techniques to reduce the number of false positives they generate. One solution they use is to incorporate some communication session context into the intrusion detection signatures. Currently, several vendors propose a stateful approach to intrusion detection at the TCP session layer and at the application layer. However, given all the characteristics and components of the targeted systems (such as the proper operating system and the proper service) that must exist for an intrusion to succeed, a more global approach than the inclusion of the communication state. However, a stateful approach can only reach its full potential if these Stateful IDS are able to correlate the intrusion information with additional network information gathered before the intrusion.

In this thesis, we presented a proof of concept IDS called Passive Network Monitoring Tool (PNMT) that is able to passively acquire network context information, allowing the

addition of such context to network intrusion detection rules. To achieve this goal, we first did a survey of the technology available today for passive network monitoring. With the development of PNMT, we made five contributions to the field of intrusion detection: a network model, a packet model, a framework to implement OCL passive network context gathering and intrusion detection rules, an analysis of the relationships between the Snort Nessus and Bugtraq databases and a library of OCL rules. We analyzed information sources to populate the static part of the network model and to derive OCL rules to populate the dynamic part of the network model. Based on this analysis, we developed a library of passive network context gathering rules and intrusion detection rules with network context. These rules were tested on real traffic traces using real exploits and the results were compared with those of Snort, a well-known IDS.

Based on the results we obtained, we showed that intrusion detection rules with network context can help reduce the number of false positives. We also showed that the PNMT intrusion detection rules using network context were able to identify false positives for several exploits used against several types of systems. Reference links from Snort, Nessus and Bugtraq databases were used when available to infer the proper network context to be added to intrusion detection rules and to derive passive network context gathering rules. However, we believe that considerable improvements in terms of information formatting and references will have to be made in these databases to be able to automatically derive intrusion detection rules with network context from them or to use them with event correlation systems.

Two issues must be addressed to develop a more flexible version of PNMT that could be used in a production environment. First, a larger set of rules for passive information gathering and intrusion detection needs to be defined and tested. As a result, we plan to develop a larger list of passive network context gathering rules, a new virtual network testbed and a parsing program that uses information from Snort and Bugtraq to be able to rapidly derive intrusion detection rules with network context. We would also like to write traffic profiling rules in OCL to derive behavior from TCP sessions, such as traffic characteristics for certain types of services. Traffic characterization is currently being conducted at CRC using lightweight metrics to characterize normal traffic such as HTTP, FTP, SMTP and IMAP. We believe that this information could be useful to network administrators when they try to examine information flows in networks where they suspect hidden malicious activity is disguised as ordinary packets.

A second issue that needs to be addressed is the development of an OCL rule compiler to remove the implementation procedure from the hands of the network administrator. This would accelerate testing and increase the usability of PNMT. As a result, an approach using PLTL as an OCL rules compiler and a monitoring engine for our tool is being investigated.

Other issues such as the performance of the system are very important for real-time network monitoring, in particular when these systems need to analyze several thousands of packets per second. Thus, the network monitoring engine also needs to be very efficient. Furthermore, additional investigation about the management of the network-context

knowledge-base is also very important to the system performance. In this study, we only used a basic network-context knowledge database without any temporal aspect. In particular, the unavailability of network context required by the intrusion signature when the intrusion part of the signature is triggered has to be considered. Thus, efficient techniques to ensure network context availability are required. The sensor distribution and distributed attack scenarios also need to be considered. For instance, the way network sensors are distributed could help reduce the number of false positives. Distributed attack scenarios also need to be modeled as intrusion detection rules to reduce the number of false positives because attacks can come from a variety of sources. Thus, we have to investigate whether our network model, packet model and our usage of OCL are flexible enough to express these intrusion detection rules in a distributed environment.

Currently, the development of PNMT is still in progress and plans are being made to test it on a large network when it is completed. We believe that this prototype and the proposed passive network context monitoring framework presented in this thesis constitute key elements that could be integrated in commercial IDS and open source IDS to reduce the number of false positives. Furthermore, we provide a standard approach to incorporate the network context into network monitoring systems for the future.

Bibliography

- [1] D. Anderson, T. Frivold, A. Tamaru, and A. Valdes. NIDES: Software Users Manual — Beta-Update Release. December 1994.
- [2] D. Anderson, T. Frivold, and A. Valdes. NIDES: A Summary. May 1995.
- [3] H. Anderson. Introduction to Nessus. <http://www.securityfocus.com/infocus/1741>, October 2003.
- [4] H. Anderson. Nessus, Part 2: Scanning. <http://www.securityfocus.com/infocus/1753>, December 2003.
- [5] H. Anderson. Nessus, Part 3: Analysing Reports. <http://www.securityfocus.com/infocus/1759>, February 2004.
- [6] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-Based Runtime Verification. In *Proceedings of Fifth International Conference on Verification*, Venice, January 2004.
- [7] J. Beale and J.C. Foster. *Snort 2.0 Intrusion Detection*. Syngress Publishing, 2003.

- [8] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, 1999.
- [9] L.C. Briand, W. Dzidek, and Y. Labiche. Using Aspect-Oriented Programming to Instrument OCL Contracts in Java. Technical Report SCE-04-03, Ottawa: Carleton University, February 2004.
- [10] L.C. Briand, Y. Labiche, and J. Leduc. Towards the Reverse Engineering of UML Sequence Diagrams for Distributed Real-Time Java Software. Technical Report SCE-04-04, Ottawa: Carleton University, September 2004.
- [11] P. Charles. Network Packet Capture Facility for Java. <http://sourceforge.net/projects/jpcap/>, September 2004.
- [12] M. Couture. Détection d'intrusions et analyse passive de réseaux. Master's thesis, Université Laval, Département d'informatique et de génie logiciel, 2005.
- [13] M. Couture and F. Massicotte. Systèmes et langages de détection d'intrusions. Ottawa: CRC Technical Report CRC-RP-2005-001, July 2005.
- [14] A. DeMontigny-Leboeuf. A Multi-Packet Signature Approach to Passive Operating System Detection. Ottawa: CRC/DRDC joint Technical Report CRC-TN-2005-001 / DRDC-Ottawa-TM-2005-018, December 2004.

- [15] A. DeMontigny-Leboeuf and F. Massicotte. Passive Network Discovery for Real-Time Situation Awareness. In *NATO/RTO Adaptive Defence in Unclassified Networks*, Toulouse, France, April 2004.
- [16] R. Deraison, R. Gula, and T. Hayton. Passive Vulnerability Scanning- An Introduction to NeVO. <http://www.tenablesecurity.com/papers.html>, August 2003.
- [17] S. Eckmann, G. Vigna, and R. Kemmerer. STATL: An Attack Language for State-Based Intrusion Detection. Technical report, Santa Barbara : University of California, 2000.
- [18] E. A. Emerson and J. Y. Halpern. Sometimes and Not Never Revisited: on Branching versus Linear Time Temporal Logic. *Journal of the ACM*, 33(1):151–178, 1986.
- [19] E. A. Emerson and C. L. Lei. Modalities for Model Checking: Branching Time Logics Strikes Back. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 267–278, Cambridge, June 1986.
- [20] D. M. Gabbay. The Declarative Past and Imperative Future: Executable Temporal Logic for Interactive Systems. In Behnam Banieqbal, Howard Barringer, and Amir Pnueli, editors, *Conference on Temporal Logic in Specification*, volume 398 of *Lecture Notes in Computer Science*, pages 409–448. Springer, 1989.

- [21] C. Ghezzi, D. Mandrioli, and A. Morzenti. TRIO: A Logic Language for Executable Specifications of Real-Time Systems. *Journal of Systems and Software*, 12(2):107–123, 1990.
- [22] R.P. Goldman, W. Heimerdinger, S.A. Harp, C.W. Geib, V. Thomas, and R.L. Carter. Information Modeling for Intrusion Report Aggregation. In *Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX II)*, pages 329–342. DARPA, 2001.
- [23] C. Green and M. Roesch. Snort User Manual 2.1.0 - The Snort Project, December 2003.
- [24] R. Gula. Correlating IDS Alerts with Vulnerability Information. <http://www.tenablesecurity.com/papers.html>, December 2002.
- [25] K. Havelund and G. Rosu. Synthesizing Monitors for Safety Properties. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 342–356, 2002.
- [26] P. Hazel. PCRE - Perl Compatible Regular Expressions. <http://www.pcre.org/>, 2004.
- [27] C. Hornig. RFC 894: A Standard for the Transmission of IP Datagrams over Ethernet Networks, April 1984.
- [28] K. Ilgun. USTAT: A Real-Time Intrusion Detection System for UNIX. In *Proceedings of the 1993 IEEE Symposium on Research in Security and Privacy*, pages 16–28, Oakland, CA, 1993.

- [29] V. Jacobson, C. Leres, and S. McCanne. Tcpdump man page. http://www.tcpdump.org/tcpdump_man.html.
- [30] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained - The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.
- [31] S. Kumar. *Classification and Detection of Computer Intrusions*. PhD thesis, Purdue University, Department of Computer Sciences, 1995.
- [32] S. Kumar and E. Spafford. An Application of Pattern Matching in Intrusion Detection. Technical Report 94-013, Purdue University, 1994.
- [33] S. Kumar and E. Spafford. A Software Architecture to Support Misuse Intrusion Detection. In *Proceedings of the 18th National Information Security Conference*, pages 194–204, 1995.
- [34] O. Lichtenstein and A. Pnueli. Checking that Finite State Concurrent Programs Satisfy their Linear Specification. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1985.
- [35] U. Lindqvist and P. A. Porras. Detecting Computer and Network Misuse Through the Production-Based Expert System Toolset (P-BEST). In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 146–161, Oakland, May 1999. IEEE Computer Society Press.

- [36] Z. Manna and A. Pnueli. Verification of Concurrent Programs: The Temporal Framework, in the Correctness Problem in Computer Science. *International Lecture Series in Computer Science*, 1981.
- [37] Mitre. CVE Dictionary. <http://www.cve.mitre.org/>.
- [38] B. Morin and H. Debar. Correlation of Intrusion Symptoms: an Application of Chronicles. In *6th International Conference on Recent Advances in Intrusion Detection (RAID 2003)*, volume 2820 of *LNCS*, Pittsburg, September 2003. Springer.
- [39] B. Morin, L. Mé, H. Debar, and M. Ducassé. M2D2 : A Formal Data Model for IDS Alert Correlation. In *5th International Conference on Recent Advances in Intrusion Detection (RAID 2002)*, volume 2516 of *LNCS*, pages 177–198, Zurich, October 2002. Springer.
- [40] OMG. OMG homepage. <http://www.omg.org/>.
- [41] OMG. UML 2.0 OCL specification. <http://www.omg.org/docs/ptc/03-10-14.pdf>, 2003. Draft Adopted Specification ptc/03-08-08.
- [42] OMG. UML 2.0 superstructure specification, 2003. Final Adopted Specification ptc/03-08-02.
- [43] A. Ornaghi and M. Valleri. Ettercap Project Homepage. <http://ettercap.sourceforge.net/>, December 2004.

- [44] V. Paxson. BRO: A System for Detecting Network Intruders in Real-Time. *Computer Networks*, 31(23–24):2435–2463, 1999.
- [45] D. C. Plummer. RFC 826: An Ethernet Address Resolution Protocol, November 1982.
- [46] J. Postel. RFC 768: User Datagram Protocol, August 1980.
- [47] J. Postel. RFC 791: Internet Protocol, September 1981.
- [48] J. Postel. RFC 792: Internet Control Message Protocol, September 1981.
- [49] J. Postel. RFC 793: Transmission Control Protocol, September 1981.
- [50] M. Roger and J. Goubault-Larrecq. Log Auditing through Model Checking. In *Proceedings of 14th IEEE Computer Security Foundations Workshop (CSFW'01), Cape Breton, Nova Scotia, Canada*, pages 220–236. IEEE Computer Society Press, June 2001.
- [51] NFR Security. NFR homepage. <http://www.nfr.com/>.
- [52] Tenable Network Security. Nessus Scripts. <http://www.nessus.org/plugins/>.
- [53] Tenable Network Security. NeVO homepage. <http://www.tenablesecurity.com/nevo.html>.
- [54] Tenable Network Security. Tenable homepage. <http://www.tenablesecurity.com/>.
- [55] SecurityFocus. Bugtraq Homepage. <http://www.securityfocus.org/>.

- [56] SecurityFocus. Bugtraq Vulnerability Database. <http://www.securityfocus.org/bid>.
- [57] R. Sekar, Y. Guang, S. Verma, and T. Shanbhag. A High-Performance Network Intrusion Detection System. In *ACM Symposium on Computer and Communication Security*, 1999.
- [58] SourceFire. RNA homepage. <http://www.sourcefire.com/products/rna.html>.
- [59] SourceFire. Snort homepage. <http://www.snort.org/>.
- [60] SourceFire. Snort rules. <http://cvs.sourceforge.net/viewcvs.py/snort/snort/doc/signatures/>.
- [61] SourceFire. SourceFire homepage. <http://www.sourcefire.com/>.
- [62] Internet Security Systems. ISS homepage. <http://www.iss.net/>.
- [63] Internet Security Systems. RealSecure Network. http://www.iss.net/products_services/enterprise_protection/rsnetwork/sensor.php.
- [64] P. Thati and G. Roşu. Monitoring Algorithms for Metric Temporal Logic Specifications. In *Runtime Verification (RV04)*, 2004.
- [65] F. Valeur, G. Vigna, C. Kruegel, and R. Kemmerer. A Comprehensive Approach to Intrusion Detection Alert Correlation. *IEEE Transactions on Dependable and Secure Computing*, 1(3):146–169, July-September 2004.

- [66] G. Vigna. A Topological Characterization of TCP/IP Security. Technical Report TR-96.156, Politecnico di Milano, 1996.
- [67] G. Vigna and R. A. Kemmerer. NetSTAT: A Network-Based Intrusion Detection Approach. In *Proceedings of the 14th Annual Computer Security Application Conference*, pages 25–34. IEEE Press, December 1998.
- [68] A. Yee. Making False Positives Go Away. <http://www.computerworld.com/securitytopics/security/story/0,10801,89122,00.html>, January 2004.
- [69] M. Zalewski. Passive OS Fingerprinting. <http://lcamtuf.coredump.cx/p0f.shtml>, 2003.