A SET OF APPROACHES TO EVALUATE AND ADDRESS THE ACCURACY PROBLEM IN INTRUSION DETECTION SYSTEMS

by

Frédéric Massicotte

A thesis submitted to the Faculty of Graduate Studies and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Electrical and Computer Engineering

Ottawa-Carleton Institute of Electrical and Computer Engineering (OCIECE)

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario, Canada, K1S 5B6

December 2010

Your file Votre référence
ISBN: 978-0-494-81563-2
Our file Notre référence
ISBN: 978-0-494-81563-2

The undersigned recommend to
the Faculty of Graduate Studies and Postdoctoral Affairs
acceptance of the thesis

A SET OF APPROACHES TO EVALUATE AND ADDRESS THE ACCURACY PROBLEM IN INTRUSION DETECTION SYSTEMS

submitted by

Frédéric Massicotte

in partial fulfillment of the requirements for

the degree of Doctor of Philosophy in Electrical and Computer Engineering

---

Chair, Howard Schwartz, Department of Systems and Computer Engineering

---

Thesis Supervisor, Yvan Labiche

External Examiner, Marc Dacier, Institut Eurécom

Carleton University
December 2010

À mon père,

à ma mère,

à ma famille,

à mes amis,

à Lucie, à Véronique, à Alexandre, à Madeleine, ...

Home is not a place,

it's where your passion drives you

Abstract

Intrusion Detection Systems (IDSs) protect computer networks against attacks and intrusions in combinations with firewalls and anti-virus systems. Many studies have reported that IDSs have several accuracy problems. For example, IDSs can generate thousands of alarms a day that flood network administrators, and many of these alarms are false alarms. As a result, network administrators run the risk of missing good alarms lost in the noise generated by the false alarms. In this thesis, we present three contributions to the domain of IDS testing and evaluation to measure this accuracy problem and we present one contribution to the domain of IDS signature generation to generate automatically IDS signatures.

Acknowledgements

I would like to warmly thank Dr. Labiche for his guidance and for allowing me to work on such a stimulating subject throughout the course of my Ph.D. degree.

I would also like to thank Dr. Esfandiari, Dr. Somayaji, Dr. Chung-Horng Lung, Dr. Jourdan and Dr. Dacier 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 Mathieu Couture, John Robinson and Tim Symchych. Your continued support was greatly appreciated.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

An Intrusion Detection System (IDS) protects computer networks against attacks and intrusions in combination with firewalls and anti-virus systems. One class of IDS is called signature-based network IDSs as they monitor network traffic, looking for evidence of malicious behaviour as specified in attack descriptions (referred to as signatures). A signature-based network IDS, or simply IDS in the remainder of this thesis, comprises a set of signatures, specifying conditions under which the IDS should report attacks, and an IDS engine that monitors the network and evaluates whether conditions in signatures hold.

Many studies have reported that IDSs have problems accurately identifying attacks (*accuracy problem*). For example, IDSs can generate thousands of alarms a day that flood network administrators, and many of these alarms are usually considered to be so called *false alarms* [1, 2]. As a result, network administrators run the risk of missing *good alarms* lost in the noise generated by the *false alarms*.

Several root causes of this *accuracy problem* have been identified in the literature. For instance, it has been reported that some IDSs provide alarms (also called IDS events) related to networking problems such as misconfigured equipments, instead of attacks [3, 4, 5]. Another cause is the difficulty, for IDSs, of distinguishing between normal and attack traffic. In fact, IDSs often provide

alarms on normal, legitimate traffic [6]. This particular accuracy problem makes the IDSs vulnerable to a *squealing* attack where the attacker generates *synthetic attacks* (i.e., traffic that mimics a real attack without doing a real attack) specifically tailored to make IDSs raise many alarms (e.g., to overwhelm the network administrator) and thus prevent it from identifying the real attack [7]. Yet another cause of this *accuracy problem* (but not related to false alarms) is the inability of IDSs to verify the success or failure of attack attempts [8, 9, 10]. For instance, in many situations, IDSs provide the same alarm whether the attack is successful or not. As a result, it is difficult for network administrators to distinguish between alarms related to a compromised target system and alarms related to a target system that is not compromised. This happens for example when an attacker tries to attack many target systems in the network at the same time or when a computer worm tries to infect various target systems. Finally, IDSs are also known to miss attacks. In particular, many techniques exist to evade detection by IDSs, such as packet fragmentation and HTTP request encoding. These techniques are used to slightly modify the attacks to prevent the IDSs from detecting them. Papers such as [11, 12, 13, 14, 15, 16, 17] describe the most popular IDS evasion techniques used by hackers and show how some IDSs cannot detect an attack when such techniques are used.

It is also important to understand the dynamic nature of this accuracy problem. Every day, vulnerabilities in software programs commonly used by computer systems connected on the Internet are identified and documented. For instance, based on the National Vulnerability Database (NVD)[1], more than 4900 *new* software vulnerabilities have been identified every year[2] since 2005.

Moreover, software programs that attempt to exploit these vulnerabilities are developed to take advantage of them. More specifically, malware programs can exploit one or many vulnerabilities to infiltrate vulnerable computer systems to take control of them. Malware programs are among the most important security threats on the Internet. McAfee has identified, over a period of 22 years (from 1986 to March 2008), 10 million unique malware programs.[3] From March 2008 to March 2009,

[1] nvd.nist.gov/home.cfm
[2] web.nvd.nist.gov/view/vuln/statistics
[3] www.avertlabs.com/research/blog/index.php/2009/03/10/avert-passes-milestone-20-million-malware-samples/

the number of malware programs they identified actually doubled (i.e., 20 million). Only in the first half of this year, they have cataloged 10 million new pieces of malware for a total of around 43 million unique malware (including variants).[4] Note however that these numbers can be questioned, especially if McAfee is using the malware MD5 hash to count malware because polymorphic malware are essentially identical to one another but have different hash values. However, researchers still agree that malware are one of the main threats on the Internet today.

The above-mentioned literature shows that many researchers in the IDS research community agree that IDSs have an accuracy problem. These conclusions are now common knowledge within the research community, but are often based on anecdotal experience with IDSs instead of being based on a systematic assessment of the problem. As a result, researchers are proposing new IDS signatures (i.e., the attack specifications) and new IDS engines (i.e., the IDS component that uses the signatures to identify attacks) without necessarily knowing whether they are addressing an important part of the accuracy problem.

We believe that a number of standard systematic practices should be put in place to allow researchers to maintain their understanding of the ever-changing nature of the accuracy problem. They would allow researchers to identify more precisely the key accuracy problems, leading to a better understanding of their root causes. In turn, this would lead to insightful solutions to these key accuracy problems and more importantly, to the identification of the actual impact of the solutions researchers propose.

In the remainder of this introduction, we first propose to precisely model and define the accuracy problem of IDSs in a way that has not been done in the literature. Second, we discuss the method that we followed in this thesis to assess and address the accuracy problem. Third, we present the scope of the accuracy problem addressed in this thesis. Finally, we present an overview of the thesis and of the work accomplished.

[4] www.mcafee.com/us/local_content/reports/q22010_threats_report_en.pdf

1.1 Problem Definition

To precisely and objectively understand and address the accuracy problem, we propose to divide the accuracy problem into at least two sub-problems, the *detection* and the *verification* problems. We will see that this division of the accuracy problem will also provide more precise test and evaluation objectives as well as more precise objectives for improving IDS signatures and IDS engines.

1.1.1 Attack Definition

To facilitate the description of the accuracy problem, we first need to specify how the information IDS engines use to identify attacks (i.e., packets) is mapped with the IDS specification (i.e., their signatures). For all network packets that can be generated, only specific (sequences of) packets can be used to generate an attack. Thus, the IDS signatures have to account for (1) the stateful nature of the protocols used in the attack (e.g., TCP session handshake), (2) the proper sequence of packets that lead to the attack (e.g., the attacker needs to correctly login to a FTP server) and (3) the attack packet themselves (e.g., buffer overflow). IDSs do not only detect attack, but are also used to detect potential policy violations (e.g., usage of MSN or Skype) and potential illegal access to network services. In the scope of this thesis, we restrict our research to attacks against vulnerabilities. Thus, an *attack* against a computer system is a series of packets that *attempt* to exploit a vulnerability on the target system.

Consequently, we denote A the set of attacks, S^d the set of signatures of an IDS d, and A_{S^d} the part of the information used by the IDS d to detect attacks (i.e., the sequences of packets that represent attacks). Thus, a good IDS is one that satisfies $A_{S^d} = A$. In reality, A and A_{S^d} are not equal, which is the outward sign of the IDS accuracy problem. Moreover, A and A_{S^d} are constantly changing over time, hence the dynamic nature of the accuracy problem.

We also define *normal traffic* or *attack free traffic* as the (sequences of) packets that are not attacks. In the remaining of this thesis, we will use the term *normal traffic* to refer to *attack free*

traffic in the sense that this traffic does not contain attacks against a vulnerability.

1.1.2 Detection Problem

We define the *detection problem* as the inability of an IDS to distinguish between normal traffic and attack traffic (on which it should provide IDS events). In other words, it relates to the ability of the IDS to properly detect attacks. Making $A_{S^d} = A$ consists in resolving the detection problem.

Furthermore, we divide the detection problem into the *extra-detection problem* and the *evasion problem*. The *extra-detection problem* is characterized by the set of attacks monitored by the IDS (according to a set of signatures) that are not attacks (i.e., $A_{S^d} \setminus A$). The *evasion problem* is characterized by the set of attacks that are in $A \setminus A_{S^d}$.

1.1.3 Verification Problem

We define the *verification problem* as the inability of the IDS to distinguish between successful and failed attack attempts. The ability to properly detect attacks (only providing IDS events for attacks that are in A) is not sufficient to address the accuracy problem. It is also important to identify whether attacks are successful or not. For example, an attack can only be successful to exploit a vulnerability when the target system or the target service is vulnerable to it and receives the proper (sequence of packets) within the appropriate time window. Thus, detecting that the proper (sequence of) packets has been sent (detection problem) is the first step, but an IDS also needs to be able to verify the success of this attack (verification problem).

1.2 Research Methodology

To assess and address the IDS accuracy problem, and to structure our work and contributions, we propose a *research methodology* that reflects the dynamic nature of the accuracy problem. First, we need to outline some other terms before defining this *research methodology*.

From a software engineering point of view, the IDS signatures play the role of the *specification* of the IDS engine and the IDS engine plays the role of the *implementation* of this *specification*. We acknowledge that most of the time IDS signatures do not also say "what" (i.e., specification), but only "how" to identify attacks. Therefore, they could be considered as being part of the *implementation*. Ideally IDSs would have, for each attack, a high-level specification from which it derives a signature. The real problem is that there is no such specification for IDSs. In the remainder of this thesis, we propose that the IDS signatures play the role of the *specification* since they are the closest to the attack specification. Also, from the IDS engine point of view, the IDS signatures *specify* what to look for (e.g., attacks) in the network traffic.

Thus, *assessing* the accuracy of an IDS amounts to software *verification* and *validation* problems. In our context, we define *software verification* as the process of evaluating whether the implementation (IDS engine) conforms to its specification (IDS signatures) and *software validation* as the process of evaluating whether the IDS (i.e., its implementation and specification) accomplishes its intended requirements (i.e., accurately identifying attacks). Thus, software *verification* and *validation* are precisely two different aspects of assessing the accuracy of an IDS. It is important not to confuse the software verification problem with the accuracy verification problem (previously defined). Our thesis is that systematic software verification and validation techniques should be used to thoroughly study the IDS accuracy problem. Qualitative and quantitative results will then help us address the detection and verification problems and improve the IDS's implementation (engine) and specification (signatures).

Consequently, *addressing* the accuracy problem of an IDS amounts to improving its *specification* and/or its *implementation*. Furthermore, *assessing* and *addressing* the accuracy problem will lead us to make contributions in three distinct research fields: IDS Verification and Validation, IDS Specification and IDS Implementation.

We propose a methodology that takes into account these three research fields and that conveys how contributions and results obtained in one research field influence the other fields. We also think

that to adequately address the accuracy problem, researchers need to see it as a whole, starting from an IDS verification and validation perspective to identify key accuracy problems in current IDSs, and then continuing with the research on improving the IDS specification and implementation to address the identified accuracy problems. Finally, we believe that with the dynamic nature of the accuracy problem, the proposed methodology must start with an assessment of the accuracy problem to provide a clear understanding of this problem and that it must be iterative to ensure that the resulting work on IDS technology is still up-to-date.

Figure 1.1 presents our research methodology to assess and address the detection and verification problems. This methodology is composed of three stages that are named after their corresponding research field: the IDS Verification and Validation, the IDS Specification and the IDS Implementation.

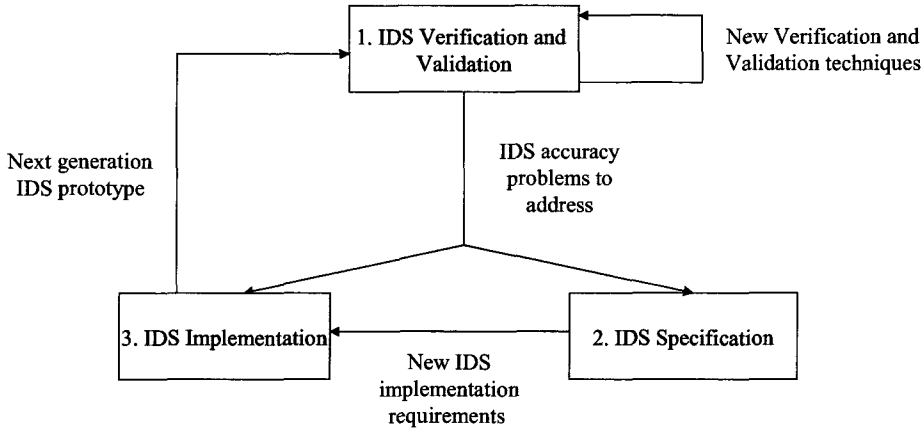

Figure 1.1: Methodology

This iterative process starts by evaluating current IDS technology. This step is crucial to properly understand and measure the accuracy problem of current IDS technology. The research work conducted under IDS Verification and Validation serves two objectives: to precisely identify the key accuracy problems in current IDSs (thus defining interesting research areas in IDS Specification and IDS Implementation) and to identify requirements for improved software verification and validation

techniques that are missing in the current literature on IDS evaluation (loop on IDS Verification and Validation in Figure 1.1).

Then the iterative process turns to either the improvement of the IDS specification (signatures) or the improvement of the IDS implementation (or both) based on the problems identified in the IDS Verification and Validation step. The results obtained in the IDS Specification step may also directly provide new requirements for IDS Implementation. For instance, improved IDS signatures may lead to research on developing new algorithms in the IDS engine, to properly and efficiently use these new IDS signatures.

Finally, when these steps are completed, the process starts over. The new IDS developed through this iteration is used in the next iteration as the IDS to be evaluated with current IDS verification and validation technology. Thus, this new research cycle brings us one step closer to addressing the IDS detection and verification problems. In this thesis, we present the results obtained after executing the first iteration of this iterative process.

1.3 Scope of the Thesis

The purpose of this thesis is to propose new approaches to *assess* and *address* some aspects of the detection and verification problems. However, to clearly define the work we conducted, we first need to refine its scope and to state the strategy we used to address and assess these problems.

In this thesis, we focus our attention on assessing and addressing the accuracy problem in the context of *attack* traffic, i.e., evasion and verification problems, and leaving out the extra-detection problem. The dynamic nature of vulnerabilities (more than 4900 new vulnerabilities a year), and attacks, as well as the increasing number of malware used on the Internet, represent a key research challenge in network security. IDSs are constantly potentially inaccurate when a new vulnerability is identified or a new malware is released on the Internet. We agree with the literature (e.g., [6]), that the IDS accuracy problem on normal traffic (i.e., extra-detection problem) is also an important

problem. However, using normal traffic for IDS research involves policy and privacy issues related to the information contained in the traffic traces (e.g., users's emails and web requests) that go well beyond the scope of the IDS accuracy problem. This is not the case with attack traffic. Both areas offer different and important research challenges. Ideally, a global solution (for normal and attack traffic) would be required to properly assess and address the IDS accuracy problem. However, due to the extent of the research work required, both areas (normal and attack traffic) cannot be addressed in a single thesis.

The strategy at the heart of this thesis is to assess and address the detection and verification problems using *automation*. We believe that to account for the dynamic nature of the detection and verification problems, computer systems (and algorithms) should be used, putting computer systems at the service of computer system's security problems. Consequently, the goal of this research work is not only to make contributions to assess and address the detection and verification problems, it is also to ensure that proposed solutions can be automated using computer systems and algorithms, thus helping to ensure that the IDS accuracy problem can be addressed dynamically, to the maximum extent possible. Since the IDS implementation (IDS engine) is less affected by the dynamic nature of the accuracy problem (it usually does not change when new signatures are added), we concentrate our work in this thesis on the automatic verification, the automatic validation and the automatic generation of the IDS specification. Our work may include assessing and proposing IDS engines, to the extent that new requirements arising from IDS verification, IDS validation or IDS specification, make this necessary.

In conclusion, our focus in this thesis will be on assessing the detection (i.e., evasion) and verification problems by proposing software verification and validation techniques. We will also address these accuracy problems by proposing new IDS specification (signatures) in the context of attack traffic. Moreover, to account for the dynamic nature of the accuracy problem, we promote the usage of automation (for the verification, validation and specification of IDSs).

1.4 Overview of the Thesis

In this thesis, we propose software verification and validation techniques to identify the root causes of the detection (i.e., evasion) and verification problems of IDSs, in the context of attack traffic, by proposing well-defined, systematic procedures to measure and assess these accuracy problems. Based on these results, we clearly derive research objectives to address the root causes of these accuracy problems in a systematic manner. From these research objectives, we also propose new IDS signatures (specification). Consequently, in this thesis, contributions to address the detection and verification problems are made by improving and proposing software verification and validation techniques, and new IDS specifications. More specifically, this thesis is structured as follow.

Chapter 2 describes terms and concepts related to IDSs and discuss the state of the art on IDS technologies.

Chapter 3 presents the related work on IDS verification and validation as well as recent related work on suggested improvements to IDS specification and implementation.

Chapters 4, 5, 6 and 7 are the core chapters of this thesis. Chapters 4, 5 and 6 present the research we conducted and our contributions to IDS verification and validation to identify detection and verification problems. Chapter 7 presents the research we conducted and our contributions to improve IDS specification and IDS implementation. Specifically, the results of Chapters 4, 5 and 6 were used to identify requirements to improve IDS specification; and Chapter 7 shows how we improved IDS specification and implementation given some of these requirements. Also, while our contributions in all these four chapters relate to the field of Intrusion Detection, understanding our solutions requires knowledge of other computer engineering disciplines. For instance, Chapter 4 requires knowledge of network security as well as software testing and experimental research, Chapter 5 requires knowledge of IDS implementation as well as software testing, Chapter 6 requires knowledge of IDS specification and set theory; and Chapter 7 requires knowledge of IDS specification and data mining algorithms. The tone and vocabulary used in each chapter reflects this and as a result is not always primary

focused on intrusion detection terminology. Below is a more detailed description of these chapters.

Chapter 4 presents the Vulnerability Exploitation Program (VEP) testing model to automatically and iteratively assess the detection and verification problems of IDSs. More precisely, our VEP testing model addresses at the same time two different, but complementary, problems which are the verification of the implementation (i.e., the IDS engine) and the validation of the engine specification (i.e., the IDS signatures). We also propose a methodology to keep up with the dynamic nature of the accuracy problem (i.e., new attacks are identified daily). The resulting system, the Automatic Experimentation System / Virtual Laboratory (AES/VLab), is used to automatically generate test cases using a well-defined test criterion and using real attacks generated by VEPs to assess the detection and verification problems of IDSs. We also fulfill a need of the IDS research community by making publicly available on the Internet the documented test cases we generated with our VEP testing model. We present an analysis of the limits of the VEP testing model. We also perform a case study to assess the detection and verification problems using our VEP testing model on two widely used IDSs (i.e., Snort [18] and Bro [19]). We conclude that our VEP testing model can efficiently assess the detection (i.e., evasion) and verification problems of IDSs.

Chapter 5 presents the IDS Engine Stimulator (IDSES) testing model. This testing model clearly maps the IDS engine testing problem to software verification testing principles, thus defining clearly the sub-problems of verifying IDS engines. The IDSES testing model is used to specifically assess the detection problem in the IDS engine (i.e., verification of the IDS engine). Based on the results obtained in Chapter 4 with the VEP testing model, we believe that the software verification and validation problems need to be treated separately to facilitate the identification of the root cause of problems in IDS engines. Indeed, if an IDS does not raise an alarm when tested with a VEP for example, one does not easily know whether the reason is a poor IDS signature database (the signatures are not precise enough or even incorrect–software validation problem) or a poor implementation of the specification (i.e., the specification/signature is correct and precise, but the engine does not implement it correctly–software verification problem), or both. Moreover, techniques

such as VEPs only cover a portion of the specification because a small proportion of vulnerabilities have VEPs. IDSES addresses these problems. Using IDSES, we perform a case study to assess the detection problem using different test criteria on a widely used IDS (i.e., Snort). The IDSES testing model is fully automated to also address the dynamic nature of the accuracy problem. We conclude that our IDSES can automatically identify detection problems in IDS engines that prevent the IDS from detecting attacks.

Chapter 6 presents the IDS Signature Space Analysis (IDS-SSA) approach. It complements the IDSES by verifying the IDS specification (signature database). In Chapter 5, the IDSES testing model is used to specifically assess the detection problem in the IDS engine (i.e., verification of the IDS engine), while in Chapter 6, the IDS-SSA testing model is used to specifically assess redundancy (i.e., inclusion/intersection) between signature specifications that could cause detection problems (i.e., verification of the IDS signature database). We believe that although IDSES can shed light on the detection problems in the IDS signatures (results of Chapter 5), it is not the proper model to asses the problem of redundancy in signature specifications and to facilitate the identification of the root causes of such redundancy. Indeed, while techniques such as IDSES can cover the IDS specification (i.e., provide at least one test case per signature), they only cover a portion of each signature specification because it is nearly impossible to generate a test case for every possible packet monitored by each signature (exhaustive testing is in general impossible). IDS-SSA addresses this problem by modeling signatures and checking whether they have expected properties (in this case, no inclusions, intersections, equalities). IDS-SSA is also fully automated to address the dynamic nature of the accuracy problem. We conclude that our IDS-SSA can automatically identify detection problems in IDS signature database that prevent the IDS from detecting attacks.

Chapter 7 presents an approach to automatically improve the IDS specification and implementation that addresses some of the accuracy problems identified through the IDS verification and validation testing models. More precisely, we propose an approach for the automatic generation of IDS signatures, based on data mining techniques, and create a multi-session IDS engine which is

able to use those signatures to address the accuracy verification problems identified in Chapter 4. Chapter 7 also presents the results obtained when these new IDS engine and signatures are tested against other IDSs (i.e., Bro, Snort and Snort UC Davis [10]). We conclude that our verification rules were generally more or as accurate as the ones found in Bro, Snort and Snort UC Davis.

Finally, conclusions are drawn in Chapter 8.

1.5 Contributions and Publications

The work described in Chapter 4 was published at the Annual Computer Security Applications Conference (ACSAC) [20]. This contribution to the domain of IDS verification and validation has already impacted the researchers working in this field. First, our VEP test cases were distributed to many researchers and organizations (i.e., more than 20 universities, research centres and companies) for projects related to improving IDS verification and validation. Second, the AES/VLab already led to research collaborations and to co-publications [21, 22]. The contributions presented in Chapter 4 are four-fold:

- We propose a new tool, called the Automatic Experimentation System / Virtual Laboratory (AES/VLab), to conduct network security experimentations (e.g., to generate IDS test cases from VEPs). The AES/VLab allows to generate test cases from VEPs that can be automatically updated (i.e, as VEPs become available on the Internet) and documented.

- We develop a VEP data set (i.e., test suite) that is publicly available for the research community to use for evaluating their IDS and is more up-to-date than DARPA's data set. This VEP data set can be used as a common reference point to compare different approaches.

- We propose a new VEP testing model that uses together VEP selection criteria, test criteria, the AES/VLab, as well as the detection and verification metrics to automate the assessment of the detection and verification accuracy of IDSs.

- We perform a case study to illustrate how our VEP testing model can be used to assess the detection and verification accuracy of two widely used IDSs. We also perform an evaluation of the limitations of the VEP testing model.

The work described in Chapter 5 was published in a technical report [23] and at the IEEE International Conference on Quality Software (QSIC) [24]. The contribution of Chapter 5 is four-fold:

- We propose a testing model and a specification-based testing approach that clearly maps the IDS engine testing problem to standard software testing principles. When doing so, as opposed to existing work, we systematically decompose the IDS engine testing problem and evaluate which mature testing techniques can be used (or adapted) and for what purposes. As a result, the testing model leads to systematic testing strategies for IDS engines.
- We illustrate how this testing model can be applied to an IDS.
- We present a tool that can automate our approach.
- We perform a case study to assess the effectiveness of this testing model on an IDS. Results show that our approach is effective at systematically revealing numerous problems in this IDS engine (e.g., problems that prevent the detection of attacks).

The work described in Chapter 6 was published as an extended abstract at the Recent Advances in Intrusion Detection Conference [25] and a full paper will be submitted shortly to a conference. The contribution of Chapter 6 is four-fold:

- We propose a testing model to quantify the signature overlapping problem in an IDS signature database. When doing so, this approach leads to a systematic analysis of the signature overlapping problem.
- We illustrate how this testing model can be applied to a widely-used IDS.

- We succinctly present a tool that can automate our approach.
- We perform a case study to assess the effectiveness of this approach on that IDS. Results show that our approach is effective at systematically revealing overlapping situations in the signature database of this IDS.

The work described in Chapter 7 was published at the Annual Computer Security Applications Conference (ACSAC) [26]. The contribution of Chapter 7 is four-fold:

- We propose an approach that uses a standard data mining algorithm to automatically generate IDS verification signatures, thus reducing the effort and errors related to the manual generation of such signatures.
- We illustrate how this approach can be applied with a widely-used IDS.
- We present a tool that implements our approach.
- We report on a case study that shows that our verification signatures improve the accuracy of IDSs.

Chapter 2

Setting the Context: IDS Concepts

As explained in the introduction of this thesis, our focus is on signature-based network IDSs. However, in this chapter, we introduce some key concepts that relate to IDSs in general to facilitate the understanding of our research work. In particular, Section 2.1 provides a high-level description of IDSs. Section 2.2 describes some basic concepts and provides a classification of the different types of IDS technology used today. Section 2.3 provides definitions for terms used in the IDS research community. Section 2.4 provides a description of two widely-used and maintained IDSs and justifies their usage for this research work. Section 2.5 presents our detection and verification accuracy metrics.

2.1 IDS Description

In the context of computer networks, IDSs are used to detect or prevent attacks against computer systems. They work in combination with other network security equipments, such as firewalls and anti-virus systems, to analyze the activity of computers on a network. An IDS can be defined as a system that dynamically monitors the actions taken in a given environment, and that decides whether these actions are symptomatic of an attack or constitute a legitimate use of the environment [27].

At a high-level, an IDS is designed using four components: the *IDS Engine*, the *Configuration*, the *User Interface* and the *IDS Database*. The *IDS Engine* and the *IDS Database* are the two main components of an IDS. All the other components are *Interface* components for the user and the environment to interact with the IDS.

The *IDS Engine* processes the information from the target system(s) that it is protecting. It uses two sub-components: the *Decoder* and the *Preprocessors*. The *Decoder* decodes the information such as packets, event logs or system calls and the *Preprocessors* reformat (standardize) the information before providing it to the *IDS Engine*. For example, an *HTTP Preprocessor* reformats the URLs to standardize them and to facilitate the detection process because there are many different ways of writing URLs that designate the same URL address (e.g., ASCII, Unicode). The *IDS Engine* uses two other information sources to detect attacks: the *IDS Database* and the *Configuration*. The *IDS Database* contains the information related to the detection techniques (Section 2.2) used to identify attacks. The *Configuration* specifies how the *IDS Engine* has to behave when there are attacks. For example, the *Configuration* keeps track of where and to whom the alarms (i.e., the IDS events) should be sent. The IDS events are given by the *IDS Engine* to the *User Interface*. The IDS can be used in combination with a *Countermeasure Engine*. The IDS events can then be provided to the *Countermeasure Engine* to prevent the attacks from affecting the target system using techniques such as changing the firewall rules to block the attacker or closing the connection between the attacker and the target system.

2.2 IDS Classification

IDSs can be classified using three main criteria [27]: the detection method, the behaviour on detection and the information source. The detection methods can be classified into two classes: behaviour-based and knowledge-based. The behaviour-based method is also referred to as the anomaly-based method because it uses deviation from the normal behaviour of a computer system to detect attacks.

The knowledge-based method uses information about known attacks (e.g., signatures) to detect them. Knowledge-based IDSs are more commonly developed and used because they are easier to build and maintain than behaviour-based IDSs. Moreover, the knowledge-based IDSs are referred to as signature-based when signatures are used to model the known attacks in the IDS.

The second criteria used to classify IDSs, the behaviour on detection, represents the actions taken by the IDS when an attack is detected. The behaviour on detection can be classified into two classes: passive and active. A passive IDS simply logs the attacks and does not take any action against the attacks that occur. The IDS is active when it takes action such as making corrections to the target system (removing the vulnerability), changing the firewall rules to block the attacker or closing the connection between the attacker and the target system. These active IDSs are usually referred to as Intrusion Prevention System (IPS).

The third criteria, the information source, refers to the type of information used by the IDSs to detect the attacks. There are mainly two types of IDSs in this regard: the host IDS and the network IDS. A host IDS uses primarily the computer's internal information, such as its computer event logs or its system calls, to detect attacks. A network IDS uses network packets to detect attacks. The advantages of network IDSs over host IDSs are in their protection coverage. Network IDSs can be located at a central point in a computer network. In this case, only one network IDS is required to protect many computers on the network, as opposed to host IDSs, which have to be installed on each computer on the network. Thus, from aspects such as maintainability, updatability and protection coverage, network IDSs offer a better solution. However, some intrusions are using techniques that make the detection of attacks harder by network IDSs than by host IDSs. In these cases, host IDSs are useful to analyze what the attackers are doing on the computer system. Thus, both approaches are complementary and combining them is preferable.

2.3 Useful Terms and Definitions

In the IDS research community, different terms are used to specify very similar concepts. To clarify the discussion, we outline in this section the terms used in this document and redefine some others to specify the scope of this thesis.

First, to specify the information provided to the network administrators when there is an attack, different words are used (e.g., alarm, alert and event). In this thesis, we use the word IDS event to specify this information.

Second, to refer to information contained in the *IDS Database* to detect attacks the word signature (i.e., IDS signature) and rule (i.e., IDS rule) are the most commonly used. In this thesis, we mainly use the word signature with the exception of Chapter 7. Specifically, we use the word IDS rule when we want to emphasize that an IDS signature is a logical expression (i.e., a rule) that is constructed with logical operators (i.e., $\vee$ and $\wedge$) and predicates.

Third, as explained in the introduction of this thesis, it is important to note that IDSs do not only provide IDS events related to attacks. IDSs also provide IDS events related to other security related issues, where the *Attack IDS events* are only a subset of the IDS events. For example, Snort has signatures and provides IDS events for potential policy violations (e.g., usage of MSN or Skype) and potential illegal access to network services. Thus, IDSs do not only have signatures related to the detection of attacks. Consequently, we need to define what is an attack as well as the concept of an *Attack IDS Event.*

In the introduction of this thesis, we defined an attack against a computer system as a series of packets that *attempt* to exploit a vulnerability on the target system, but we still need to provide a more precise definition. We know that the term *Attack* in the literature is also used to define illegal actions against a target system without exploiting a vulnerability on the target systems. For example, stealing a password from a legitimate user and using it to login to its computer is considered an attack. However, our definition of attacks allows us to concentrate on a specific group of attacks

that are well-defined and for which there is remediation available (patches or fixes) to protect against these attacks. Thus, network scan, session hijacking, tunneling, etc. are not considered as attacks in the context of this thesis. As a result, an attack can be precisely identified because it is associated with a vulnerability. It is also important to know that there are constantly new attacks that can be conducted against computer systems because new vulnerabilities are identified in computer systems every day. Thus, there is the concept of *known attack* (i.e., attack related to known vulnerabilities) and the concept of *unknown attacks* (i.e., attack related to unknown vulnerabilities). In this thesis, we only address the accuracy problem for *known attacks*. Thus, an attack in the context of this thesis is related to a known vulnerability.

Network administrators should be interested in all IDS events. In this thesis, the term IDS event is used to specify an event provided by the IDS when the IDS identifies that an attack against a known vulnerability is in progress. For example, an event related to a buffer overflow attack or a denial-of-service is an IDS event (i.e., *Attack IDS Events*).

2.4 Selection of IDSs

As discussed in the introduction, this research work focuses on addressing accuracy problems for knowledge-based (i.e., signature-based) network IDSs. Most IDSs available today are commercial systems. It is difficult to assess some of the problems we want to address in this thesis because the IDS engine source code and the IDS databases of commercial versions are not available for analysis. In most cases, only black box approaches can be used to infer the information required to assess and address the root causes of the IDS accuracy problem.

However, there are a number of open source IDSs that are widely used and maintained by companies and universities. The two main open source IDSs that are currently available are Snort (maintained by SourceFire) [18] and Bro (maintained by Berkeley University) [19]. Snort and Bro are knowledge-based network IDSs and their behaviour on detection can be passive or active. In

these two cases, we have access to the IDS engine source code and the IDS databases. Such open source IDSs are usually the starting point of research projects related to improving IDS accuracy because they can be efficiently used as components to implement new solutions to the accuracy problem and these proposed solutions can be easily compared with others. For example, different authors used Snort [8, 10] and Bro [9] to develop and present their solution for helping IDS to distinguish between successful and failed attack attempts. For these reasons, we selected these two IDSs (engines and signature databases) for the case studies presented in this thesis. These IDSs are also selected to assess and implement our proposed solutions to the IDS accuracy problem and to facilitate the comparison of our results with existing related work. In the next two sections, we provide a brief description of Snort and Bro. More information about Snort[1] and Bro[2] is available on their respective web site.

2.4.1 Snort

As mentioned previously, Snort [18] is a network IDS, its behaviour on detection can be passive or active, depending on its configuration, and its detection method is knowledge-based. The Snort 2.4.0 signature database (released in July 2005) is currently the last version one can freely download without registration and licensing agreement and Snort 2.4.5 (released in June 2006) is the most recent Snort IDS engine that can be used with these signatures. More recent versions of Snort (engines and signature databases) are available (e.g., the Snort 2.8.4 IDS engine was released in 2009) but their signatures, though currently very similar to the ones of version 2.4.0, cannot be shared with any unregistered user, which would prevent discussing results in details in this thesis.

Snort comes with a signature language that can be used to modify or write signatures that are used to characterize attacks. The Snort signatures can be viewed as logical expressions (i.e., rules) where the predicates (called plug-ins in Snort) verify protocol fields and data (see Section 5.2.2). Snort also offers a plug-in framework to write more plug-ins (e.g., in C) that can be used in the

[1] www.snort.org
[2] www.bro-ids.org

signature language.

2.4.2 Bro

Bro [19] is also a network IDS, its behaviour on detection is active or passive, depending on its configuration, and its detection method is both knowledge-based and behaviour-based.

Bro has two databases for the detection method: the behaviour-based signature database and the knowledge-based signature database. The Bro behaviour-based signatures specify inappropriate behaviours: for instance peer-to-peer file exchange. Bro also has a knowledge-based signature database, as in Snort, to detect attacks. However, only a few signatures are actually specific to Bro and the *s2b* package must be used to convert Snort signatures in the Bro language and populate the Bro signature database with up-to-date signatures. The advantage of using *s2b* is that is adds predicates to Snort signatures, which allows Bro to verify the success of an attack attempt for protocols such as SMTP, POP, HTTP and FTP.

2.5 Accuracy Evaluation

To assess an IDS, we first need to define what is a test case in the context of signature-based network IDSs. A *test case* in this context is a traffic trace (i.e., sequences of packets). More specifically, in this thesis, a traffic trace is a *tcpdump*[3] file (also called a *pcap* file), a standard file format used to store network traffic traces.

Second, to assess IDS accuracy, we also need accuracy metrics to measure the detection and verification problems. The accuracy metrics we developed use a confusion matrix [28] to measure the accuracy of an IDS. Based on the concept of a confusion matrix, we developed two metrics to assess the detection and verification problems of an IDS: the detection metric and the verification metric. Each of these metrics represents the accuracy achieved by the IDS.

[3] `www.tcpdump.org`

The detection metric assesses the ability of the IDS to detect attacks. Figure 2.1 presents the detection confusion matrix (M_D). The rows are associated with the *actual values* of the test cases (i.e., *Normal* traffic and *Attack* traffic) and the columns are associated with the *predicted values* provided by the IDS (i.e., attack *Detected* and attack *Not Detected*). Note that this terminology is the one used in the field of data mining [28]. We are aware that in some other research fields such as software testing, these concepts may be represented differently. Nevertheless, in the remaining of this thesis we used the confusion matrix definitions of [28].

$M_D :=$

	Detected	Not Detected
Attack	TP_D	FN_D
Normal	FP_D	TN_D

Figure 2.1: Detection Confusion Matrix

For example, cell $M_D[Attack, Detected]$ is the number of test cases containing attacks that are detected by the IDS, i.e., the true positive for detection (TP_D). Consequently, for a test case containing *Attack* traffic, the attack is *Detected* (respectively *Not Detected*) when the IDS provides (respectively does not provide) an *IDS Event related to* the attack in the test case.

To identify whether an IDS Event is *related to* an attack we rely on a test oracle. A *test oracle* specifies how we associate a *predicted value* with an *actual value*. Thus, a test oracle is specific to the test model used to generate the test cases (Chapters 4 and 5). In the context of IDSs, we associate the *predicted values* and the *actual values* using the vulnerabilities referred by an IDS event (Section 4.3.1.3) or the signature identifier provided by an IDS event (Section 5.5.1.4). Of course, for a specific test case, an IDS could provide more than one IDS event. The oracle has to be defined accordingly. For instance, in the context of the oracle of the VEP testing model presented in Section 4.3.1.3, an *Attack* test case is classified as *Detected* when at least one IDS event provided by the IDS refers to same vulnerability as the one exploited in the attack test case. Otherwise the attack test case is classified as *Not Detected*. Thus, a *Normal* test case is classified as *Detected* (resp. *Not Detected*) when the IDS provides (resp. does not provide) at least one IDS event related to an attack.

The confusion matrix also allows us to introduce the concepts of false positive, true positive, true negative and false negative. Each of the cells in the matrix represents one of these concepts. TP represents the number of true positives obtained (i.e., it represents the total number of attack test cases for which the IDS has detected the attack). FN represents the number of false negatives, FP is the number of false positives and TN represents the number of true negatives. These definitions being dependent on the type of problem (detection or verification problem), we used as a subscript the letters D or V to specify the problem associated to this measure.

In a confusion matrix, the accuracy is always obtained using the sum of the values in the diagonal, divided by the sum of all values in the matrix [28]. Equation 2.1 presents the accuracy formula for the detection metric. This measure represents the number of occurrences of correctly *predicted values* in relation to the *actual values* of the test cases.

$$A_D := \frac{TP_D + TN_D}{TP_D + TN_D + FN_D + FP_D} \tag{2.1}$$

An accuracy of zero (0) means that the IDS is never accurate and an accuracy of one (1) means that the IDS is always accurate. For example, an IDS providing results such as TP_D=1, FN_D=2, FP_D=3 and TN_D=4, has an accuracy of 0.5 and is thus accurate for 50% of the test cases.

The true positive rate and the true negative rate are also measures used in a square confusion matrix to obtain a more detailed analysis of the accuracy. These measures represent the number of occurrences of a value predicted by the IDS in relation with one of the actual values of the test cases. Equation 2.2 and Equation 2.3 describe the TP_D and TN_D rates.

$$TP_D \; rate := \frac{TP_D}{TP_D + FN_D} \tag{2.2}$$

$$TN_D \; rate := \frac{TN_D}{FP_D + TN_D} \tag{2.3}$$

Similarly, the verification metric assesses the ability of the IDS to distinguish between successful and failed attack attempts against a target system. Thus, this metric is only applicable to the test cases containing attacks that the IDS is able to detect (the test cases that are in TP_D).

Note that different IDSs do not have the same behaviour when they identify successful and failed attack attempts. Thus, our verification metric (and test oracle) has to work with different IDS behaviours. The relationship (specified by the test oracle) between the predicted values and the different types of IDS events is specific to the IDS and varies based on the behaviour of the IDS when verifying attacks. For instance, Snort and Bro remain silent when they are able to verify a failed attack and they provide an IDS event when they believe the attack is successful. As a result, the test cases for which these two IDSs provide an IDS event related to the attack test case are associated with the *Positive* predicted value and the test cases for which the IDS provides no IDS event related to the attack are associated with the *Negative* predicted value. The actual values are *Successful* attack and *Failed* attack. Figure 2.2 presents the verification confusion matrix. Equation 2.4 describes the accuracy formula for the verification metric. Equation 2.5 and Equation 2.6 describe the measures TP_V rate and TN_V rate.

$M_V :=$

	Positive	Negative
Successful	TP_V	FN_V
Failed	FP_V	TN_V

Figure 2.2: Verification Confusion Matrix

$$A_V := \frac{TP_V + TN_V}{TP_V + TN_V + FN_V + FP_V} \tag{2.4}$$

$$TP_V \ rate := \frac{TP_V}{TP_V + FN_V} \tag{2.5}$$

$$TN_V \ rate := \frac{TN_V}{FP_V + TN_V} \tag{2.6}$$

Chapter 3

Related Work

As explained in the introduction of this thesis (Section 1.3), our focus is on assessing the detection (i.e., evasion) and verification problems by proposing IDS software verification and validation techniques, as well as on addressing these accuracy problems by proposing new IDS specification (signatures) in the context of attack traffic. To account for the dynamic nature of the accuracy problem, we propose the usage of automation (for IDS software verification, IDS software validation and IDS specification). The issue of IDS implementation is addressed only to derive some key conclusions that will be useful later in this thesis to support the approaches we propose for IDS software verification, IDS software validation and IDS specification. Consequently, our related work is presented in three parts. Section 3.1 presents the related work on automatic IDS validation and verification. Section 3.2 presents the related work on automatic generation of IDS specification. Section 3.3 presents related work on the different paradigms proposed to implement IDS engines.

3.1 IDS Validation and Verification

Our review of the IDS literature shows that four complementary techniques have been used to validate IDS technologies: (1) using network traffic collected from an emulated network (that mimics

a real network) or a real network, (2) using traffic captured from a honeypot (i.e., a vulnerable computer on the network that attracts attackers and records typical attack attempts), (3) using IDS Stimulators and (4) using Vulnerability Exploitation Programs (VEPs). We observed that, there is practically no software verification technique discussed in the literature. Although we consider these four techniques as primarily software validation techniques, it is important to note that they can be used validate the IDS specification (i.e., signatures) as well as verify the IDS implementation (i.e., engine). Indeed, it is difficult with these techniques to validate the specification without using (verifying) the implementation. In other words, if a failure is revealed while using one of these techniques, investigations is warranted to identify whether the root cause is the specification (validation) or the engine (verification). Remember that in the context of IDSs, we defined software verification as the process of evaluating whether the implementation (IDS engine) conforms to its specification (IDS signatures) and software validation as the process of evaluating whether the IDS (i.e., its implementation and specification) accomplishes its intended requirements (i.e., accurately identifying attacks).

These four validation techniques have one thing in common. They generate their test cases using captured/generated network traffic (test cases) and observe the response of the IDS when these test cases are submitted to it. They do not offer the same quality and diversity of test cases. Although the emulated/real network [29, 30, 31, 32] and the honeypot [10] techniques use real network traffic (either hostile or non-hostile), and therefore evaluate IDSs in their executing environment, they have one main drawback in our testing context: it is difficult, or even impossible, to control the diversity of the test cases (i.e., traffic traces). The traffic contained in the test cases is most of the time entirely controlled by the network users and the attackers.

The third approach is to use an IDS Stimulator. Instead of relying on *real attacks*, IDS Stimulators (e.g. *Snot*[1] and *Stick*[2]) rely on attack specifications (i.e., IDS signatures in our context)

[1] www.securityfocus.com/tools/1983
[2] packetstormsecurity.nl/distributed/stick.htm

to generate *synthetic attacks*.[3] So far, these synthetic attacks have not been used to validate IDS accuracy (with the exception of Mucus-1 [33], which is discussed in Section 3.1.2), but are instead specifically tailored to make IDSs raise many alarms (e.g., to overwhelm the network administrator), thereby performing Denial-of-service (DOS) attacks on an IDS itself, to prevent it from detecting real attacks. Thus, these IDS Stimulators address the validation of the robustness of the IDS rather than the validation of its accuracy.

The fourth approach is to use the vulnerability exploitation program (VEP) testing model to launch attacks against target systems and to observe the response of the IDS. A VEP is an attack program that attempts to exploit a vulnerability on a computer system. The usage of VEPs for IDS validation usually implies building a test bed where the attacks are launched against target systems. The traffic traces (resulting from VEP executions) are in this case the test cases submitted to the IDSs for validation. IDS evasion techniques, such as packet fragmentation and HTTP request encoding, can also be considered as VEPs to further test the accuracy of IDSs. Papers such as [11, 12, 13, 14, 15, 16, 17] describe the most popular IDS evasion techniques used by hackers. The currently known VEP approaches are DARPA [29, 30], NSS [34], Network World Fusion [35], Neohapsis [36], LARIAT [37], Thor [13], Vigna et al. [17] and Debar et al. [38]. They have several drawbacks in the methodology they use to generate test cases. For instance, no specific test criterion was used to generate the VEP test cases, which hinders their comparison and the replications of studies, the process of generating the VEP test cases was mainly manual (not automatic) and only a limited number of VEPs were used against a few target systems.

In summary, two of these approaches are outside the scope of this thesis, i.e., real/emulated network traffic and IDS Stimulator. Moreover, the VEP approach offers more control over the attacks to generate the test cases than the honeypot approach. However, key contributions remain to be done in the case of the VEP approach to accomplish the intended objectives (e.g., automation) of this thesis. Thus, our focus in the case of the software validation problem is to address these

[3]Traffic that mimics a real attack but is not necessary exploiting a vulnerability.

issues for the VEP testing model.

3.1.1 IDS Validation

We identified five limits of the VEP testing model: (limit 1) the public availability and relevance of the data set, (limit 2) the documentation of the data set, (limit 3) the generation process (automatic, flexible and updatable) of the data set, (limit 4) the test criterion used to generate the data set, and (limit 5) the absence of distinction between the assessment of the detection and the assessment of the verification problems of IDSs (i.e., the technique only focus on the detection problem). Each of these limitations is explained below.

First, the main contribution of DARPA [29, 30] to the IDS research community was to make their data sets publicly available (downloadable from the web), as opposed to most of the data sets used in other projects such as NSS [34], Network World Fusion [35], Neohapsis [36], LARIAT [37] and Vigna et al. [17]. These data sets are either not available on the web or are specific to the projects that generated them, making them less amenable to IDS testing and evaluation research in general. As a result, since the DARPA traffic traces represent the only significant data set that is publicly available, it is still used by the network security research community, even if it does not contain recent attacks (last updated in 1999) and even if the techniques used to generate its traffic traces have been openly criticized and its utilization for IDS research projects questioned [39]. Consequently, IDS research based on this data set can suffer from important limitations. To address these problems, one needs to make a dataset publicly available, to automate its creation, to add up-to-date VEPs, and to define clear VEP selection criteria.

Second, documentation is another important problem with the traffic traces from available data sets. To quantify the detection and verification accuracy of IDSs, it is essential for the data set being used to come with at least some minimal documentation. It is important for the users of the data set to know key information for each attack in the data set, such as the targeted system configuration (operating system, targeted service), the success or failure of the attacks and the attack

specification (used VEP, its configuration, targeted vulnerability). Also, the documentation has to be formatted in a way that facilitates automated analysis (e.g., accuracy problem). Since DARPA is the only publicly available data set, it is the only one for which we can evaluate how its traffic traces are documented. Unfortunately, the absence of adequate description of each attack (e.g., lack or absence of information on the success or failure of the attacks, the configuration of the target systems for each attack, the command used to generate the attack) further limits its use in IDS testing and evaluation of the accuracy problem. To address this limitation, one needs a systematic and complete documentation method of the generated VEP data set.

Third, some of the data sets are produced manually or semi-automatically. Manual interventions restrict the diversity and updatability of the data set. For instance, one of the most recent IDS evaluation by NSS (4^{th} edition) [40] was done manually [17]. Similarly, some of the tests conducted in [38] were done by hand. In addition, some techniques [13, 34] used test beds with real target systems, thus, requiring that those systems be reset to the initial conditions, which is either slow [34] (reloading Ghost images of the unaffected system) or not automatic. This limits the number of test cases generated. In fact, the number of VEPs used in these data sets is often small and the variety of targeted systems is limited. The NIST report [41] mentions that the Neohapsis [36], NSS [34] and Network World Fusion [35] data sets used respectively 9, 66 and 27 VEPs against only three different target systems. LARIAT [37], Vigna et al. [17] and Debar et al. [38], used respectively 50, 10 and 60 VEPs and the first two only used nine and five different target systems, respectively. To address this limitation, one needs an automated method to generate the (documented) VEP data set.

Fourth, to the best of our knowledge, no one used specific (test) criteria for selecting the VEPs and generating the test cases. Using (test) criteria to generate the data set is important to measure the coverage of the accuracy problem to precisely understand results, to replicate and compare studies. To address this limitation, one needs specific (test) criteria to generate the VEP data set.

Fifth, in our literature review of the VEP testing models, no one distinguishes between the

detection and verification problems when validating IDSs with VEP data sets. However, the problem (detection or verification problem) being assessed affects the choice of test criterion to use in a case study. For instance, a data set generated with only successful attacks against target system, can be used to assess the evasion problem, but it cannot completely assess the verification problem (as defined in Section 1.1). To address this limitation, one needs test criteria to generate the VEP data set.

3.1.2 IDS Verification

As explained earlier, there is practically no IDS (software) verification technique in the literature to verify IDSs. One can think that software validation technique such as VEPs and IDS stimulator could be used as verification techniques, but we believe that none of these techniques as currently applied is adequate for the following reasons. First, techniques such as VEPs or those that collect (real or simulated) network traffic, do not rely on any specification of the IDS engine to derive tests. They address at the same time the verification of the IDS engine and the validation of the engine specification (i.e., the signatures). We believe that the software verification problem can be treated separately to facilitate the identification of the root cause of problems. Indeed, if an IDS does not raise an alarm when tested with a VEP for example, one does not easily know whether the reason is a poor IDS signature database (the signatures are not precise enough or even incorrect-software validation problem) or a poor implementation of a correct specification (i.e., the engine does not implement it correctly-software verification problem), or both. Second, techniques such as VEPs, even if they are efficient at validating an IDS, only cover a portion of the IDS specification because a small proportion of vulnerabilities have VEPs. Third, as explained before, existing IDS Stimulators, which rely on the IDS specification to derive tests, address a black-box robustness testing problem rather than a black-box functional testing problem, with the exception of [33], which we discuss next.

One IDS stimulation technique, namely Mucus-1 [33], could be considered for verifying IDS

because it partially addresses the issues raised above. Mucus-1 uses IDS signature databases to derive test cases (traffic traces). It takes n IDSs and generates test cases based on each IDS signature database (IDS_{Source_1}, ..., IDS_{Source_n}), and executes them on all IDSs ($IDS_{Target_1}, ..., IDS_{Target_n}$). In doing so, Mucus-1 assumes that the signatures of each IDS are accurate enough to derive realistic, synthetic attacks that the other IDSs should be able to detect. This assumption, unfortunately, does not necessarily hold. For instance, assume IDS_{Source_i} has inaccurate signatures (which often happens [6]) and the IDS_{Target_j} fails to report an attack generated from IDS_{Source_i} signatures. It can be because IDS_{Target_j} signatures are inaccurate too, or its engine is incorrect, or its signatures are better than IDS_{Source_i}. In the latter case, Mucus-1's conclusion, however, is that IDS_{Target_j} failed to detect the (synthetic) attack, whereas it is rather an accuracy problem of IDS_{Source_i}. Mucus-1 cannot prevent such a situation from occurring. As stated by the authors [33], the *"success of this method as a tool for quantitative testing of intrusion detection systems depends heavily on the quality of the signatures present in the input. [...] loosely specified signatures impair IDSs as well as IDS stimulation tools. Obtaining high quality attack signatures is a requirement for achieving the long-term goal of using IDS stimulators to cross-test homogeneous signature-based IDSs."*. Another limitation of Mucus-1 is that it does not rely on an existing software testing body of knowledge to systematically use mature testing criteria for the construction of synthetic attacks from specification (signatures).

Thus, an IDS engine verification technique should first address the problem of verifying that an IDS engine conforms to its specification (signatures) by generating test cases/traffic traces from its signatures. In doing so, this software verification technique should not need to create real attacks, as long as the test cases correspond to its signatures. If an IDS engine does not raise an alarm when executed on a test case derived from one of its signatures, then we can conclude that it does not conform to its specification, regardless of whether this test case is a real attack or not. Second, this software verification technique should rely on an existing software testing body of knowledge to systematically verify the IDS engine.

3.2 IDS Specification

In this section, we present the related work on two aspects of the automatic generation of the IDS specification: the accuracy problem addressed by the IDS specification (i.e., detection or verification problems) and the technique (e.g., data mining) used to automate the process of generating the IDS specification.

3.2.1 Problem Addressed by the IDS Specification

A number of studies have used or examined automation to generate the IDS specification [1, 2, 3, 4, 5, 42, 43]. The authors of [3, 4, 5] focus on distinguishing IDS events related to network problems such as misconfigured equipment from IDS events related to real attacks. For instance, in [4], the authors use a data mining technique (specifically a clustering technique) to learn under which conditions alarms are kept or discarded by network administrators thereby generating rules that automate the classification of IDS events. In [1], the authors use a data mining approach to infer which IDS events are related to the same attack. This process is called *IDS events correlation.* This approach is used to reduce the number of IDS events by grouping those that are in the same attack scenarios. This provides a better understanding of attacks to network administrators.

Other research projects [1, 3, 4, 5] focused more on automating the understanding of the IDS specification (i.e., identifying the IDS events to be kept or identifying the IDS events that relate to one another) than on automating the generation of the IDS specification. We only identified the authors of [2, 42, 43] who suggested approaches to automatically generate IDS specification. These approaches focus on generating IDS signatures that distinguish normal traffic from attack traffic (i.e., detection problem). We did not find any approach addressing the automatic generation of IDS specification for the verification problem of IDSs. Consequently, contributions can be made by proposing an approach to automate the generation of IDS specification for the verification problem. Moreover, as explained in the introduction of this thesis, the improvement (e.g., IDS specification)

of the IDS accuracy on normal traffic is outside of the scope of this thesis. Thus, our focus for the automatic generation of the IDS specification will be on the verification problem, not the detection problem of IDSs.

3.2.2 Automation Process used to Generate the IDS Specification

We analyzed in more detail the algorithms used by the authors of [2, 42, 43] to automate the generation of IDS specification for the detection problem, as these are the closest projects to our work for the verification problem. These authors used different data mining algorithms to automatically generate the IDS specification.

The authors of [2, 42] used algorithms that can all be classified as clustering learning algorithms [28]. These algorithms seek to group objects so that those within a given group (i.e., cluster) are similar. They also used algorithms that can be classified as association learning algorithms [28]. These algorithms seek to identify the implications (i.e., associations) between objects.

The authors of [43] used a classification algorithm [28] (i.e., Ripper [44]). In particular, they used events from the system event logs of the target system and data mining algorithms to generate IDS detection rules that represent normal user behaviour. The generated IDS detection rules are event sequences that represent normal user behaviours. An event log sequence that does not match any rule is interpreted by the IDS as an abnormal behaviour of the user (i.e., potential attack) and an event is raised by the IDS. While this approach is for an anomaly host-based IDS (recall we focus on signature network-based IDSs), we believe that the detection and verification problems are more classification problems [28] than clustering problems, thus requiring classification algorithms. Indeed, for the detection problem, the objective is to classify normal traffic and attack traffic, whereas for the verification problem, the objective is to classify successful attacks and failed attack attempts.

In conclusion, it is therefore relevant to investigate whether classification algorithms can address the verification problem. Thus, in this thesis we investigate how classification algorithms can be used to generate the IDS specification addressing the verification problem.

3.3 IDS Implementation

It is important to note that even if there are many different paradigms proposed to implement IDS engines (i.e., Temporal Logic [45], Petri Nets [46], State Machines [6, 47], Event Calculus [48], Regular Expressions [49] and Ad hoc paradigms such as Snort [18] and Bro [19]), they all model the same problem (i.e., monitoring a sequence of packets). They all enable the IDS engine to check (Boolean) characteristics of packets (e.g. values of protocol fields, strings and bytes in the packet payload), and to check packets individually as well as in sequences (notion of state). Of course, some paradigms offer different functionalities. For instance, Petri Nets allow parallelism and synchronism between packets. However, parallelism and synchronism are not paradigms that are necessary to monitor a linear sequence of packets because attacks can be described without these concepts. Moreover, some paradigms can be translated to one another, making them potentially equivalent in the context of IDS engines. For instance, temporal logic formulas and regular expressions can be translated into automaton which is a form of state machine. Moreover, event calculus is a form of temporal logic specified in first order logic. Furthermore, Snort and Bro use an ad hoc paradigm that mimics the concepts behind a state machine.

Thus, we can reasonably assume, without loss of generality, that IDS engines are algorithms that monitor state machines. Consequently, IDSs have a signature language (specification language) that can be easily abstracted (meta-language) into a state machine where the guard conditions of events triggering state changes are logical expressions made of predicates, and these predicates use boolean functions (i.e., predicate functions) checking packet characteristics. This is precisely the approach followed in [47]. We can further assume that these logical expressions are conjunctions of predicates checking packet characteristics. If a guard is not a conjunction of predicates, we can always transform it in a disjunctive normal form (DNF) and replace the corresponding transition with as many transitions as disjuncts in the DNF. This is a key observation that will be very useful when proposing verification and validation techniques to assess IDS specification and implementation, as

well as when choosing an IDS engine when proposing new IDS signatures.

Chapter 4

Vulnerability Exploitation Program Testing Model

To adequately assess the detection and verification problems presented in Section 1.1, it is important to precisely understand under which conditions IDSs accurately identify attacks, and under which conditions they fail to do so.

In this chapter, we present a new approach to generate IDS test cases using VEPs that addresses the drawbacks identified in the literature review on the VEP approach for the validation of IDSs (Section 3.1.1). The contributions presented in this chapter are four-fold:

- We propose a new approach, supported by a tool infrastructure, called the Automatic Experimentation System / Virtual Laboratory (AES/VLab), to conduct network security experimentations (e.g., to generate IDS test cases from VEPs). The AES/VLab allows to generate test cases from VEPs that can be automatically updated (i.e, as VEPs become available on the Internet) and documented.

- We develop a VEP data set (i.e., test suite) that is publicly available for the research community to use for evaluating their IDS and is more up-to-date than DARPA's data set. This VEP

data set can be used as a common reference point to compare different approaches.

- We propose a new VEP testing model that uses, VEP selection criteria, test criteria, the AES/VLab as well as, the detection and verification metrics (Section 2.5) to automate the assessment of the detection and verification accuracy of IDSs.

- We perform a case study to illustrate how our VEP testing model can be used to assess the detection and verification accuracy of two widely used IDSs. We also perform an evaluation of the limitations of the VEP testing model.

In this chapter, we use the term VEP data set instead of VEP test suite because the data contained in the test suite were not only created for IDS validation. For instance, our VEP data set was used in other research projects related to IDS research, but not related to IDS testing and evaluation [21, 22]. Moreover, we also used the VEP data set to automatically generate IDS signatures (Chapter 7).

The first and second contributions already had an impact on the IDS research community. First, our VEP data set is publicly available, providing an alternative to the DARPA traffic traces (i.e., the only significant data set that was publicly available until recently) that are still used by the network security research community, even if they do not contain recent attacks (the attacks are from before 1999). Our VEP data set has thus generated great interest in the IDS research community. It has been distributed to more than 20 universities and research centers around the world, showing the need for such a data set for IDS testing and evaluation. Second, our AES/VLab was used in other collaborative research projects, showing the need for such a tool in network security research. In particular, we used it to conduct research on IDS verification accuracy [21] and to conduct research on honeypot systems [22].

The rest of this chapter is structured as follows. Section 4.1 presents background information on VEPs for IDS testing and evaluation. Section 4.2 presents our VEP testing model. Section 4.3 presents a case study to assess the detection and verification problems of two widely used IDSs.

Section 4.4 presents an evaluation of some of the limits of the VEP testing model to assess the detection and verification problems. Conclusions are drawn in Section 4.5.

4.1 Background on Vulnerability Exploitation Program for IDS Testing and Evaluation

VEPs can be downloaded from web sites such as SecurityFocus,[1] Milworm[2] and MetaSploit.[3] However, to use VEPs for IDS testing and evaluation, it is important to know the relationships between (1) the VEP and the vulnerability it attempts to exploit and (2) the vulnerability exploited and the IDS event provided by the IDSs.

By using these relationships, it is possible to determine which IDS signatures are supposed to be used to detect a specific VEP attack. Consequently, we also know which IDS events we are supposed to expect from an IDS for a specific VEP attack. These relationships between VEPs and IDS signatures are used in both our case study and in the analysis of the limits of the VEP testing model (Section 4.3 and 4.4).

The former relationships are maintained for the available VEPs on SecurityFocus, Milworm[4] and MetaSploit. There are several vulnerability databases on the Internet. Each of these vulnerability database (e.g., SecurityFocus) provides each known vulnerability with its own proprietary identifier (e.g., BID 2674 for SecurityFocus where the BID is the SecurityFocus vulnerability identifier). As a result, a vulnerability has a different identifier for each vulnerability database and could also have a different representation (i.e., a vulnerability from one database could be represented as three different vulnerabilities in another).

The Common Vulnerability Enumeration (CVE)[5] is a dictionary that addresses this issue by doc-

[1] www.securityfocus.com
[2] www.milworm.com
[3] www.metasploit.com
[4] The relationships between the available VEPs on Milworm and the vulnerability they attempt to exploit is maintained by CVE
[5] www.cve.mitre.org

umenting the relationships between the different identifiers of the different vulnerability databases.

The BID and the CVE identifiers are used in this thesis to refer to the vulnerability exploited by the VEP attack.

Most IDSs maintain the relationships between their signatures, the vulnerability identifiers of different databases and the CVE identifier.

4.2 A Vulnerability Exploitation Program Testing Model

Our VEP testing model attempts to overcome the five limitations identified in the related work (Section 3.1.1): (limit 1) the public availability and relevance of the data set, (limit 2) the (automatic) documentation of the data set, (limit 3) the generation process (which must be as automatic, flexible and updatable as possible) of the data sets, (limit 4) the test criterion used to generate the data set and (limit 5) the absence of distinctions between the assessment of the detection and verification problems. To address limit 1, we made our dataset publicly available, automated its creation, added up-to-date VEPs, and we defined clear VEP selection criteria (more on this in Section 4.2.1). To address limit 2, we propose a systematic and complete documentation method for our VEP data set (see Section 4.2.3.4). To address limit 3, we propose a new method to generate VEP data sets (see Section 4.2.3). To address limit 4, we propose a test criterion to generate VEP data sets (see Section 4.2.2). To address limit 5, we propose a test criterion to generate a VEP data set that can be used to assess the detection and verification problems (see Section 4.2.2). We also perform a case study to assess both the detection and verification problems using our approach (see Section 4.3).

Our approach to automatically evaluate IDS accuracy using the VEP testing model is iterative, as illustrated in Figure 4.1. First, the Web Crawler (step 1) downloads every day new VEPs from well-known web sites such as SecurityFocus, Milworm and Metasploit, and stores them in a database with their description (e.g., the vulnerability they attempt to exploit). Using the source code of the VEPs is essential for security reasons, to precisely know what the VEPs do and for instance

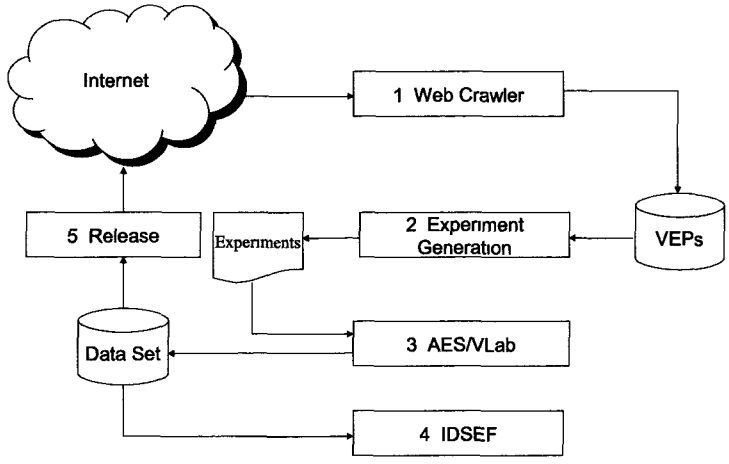

Figure 4 1 Automatic IDS Accuracy Evaluation using the VEP Model

prevent their execution outside the test environment Manual intervention is required to identify the command to execute each attack and to identify what information is provided to the attacker by the VEP for a successful attack and for a failed attack attempt Second, the Experiment Generator (step 2) generates experiments based on a test criterion and on the VEPs available Third, the AES/Virtual Laboratory (step 3) executes these experiments to generate and to document the VEP data set (i e , test cases) Fourth, the VEP data set is provided to the IDS Evaluation Framework (IDSEF) (step 4) to automatically evaluate the IDSs detection and verification accuracy Fifth, the VEP data set is stored on the Internet (step 5) for other researchers to download

This approach is iterative since, when new VEPs become available, new VEP test cases are generated and analyzed by the IDSs and when new IDSs become available, we can easily add them in the IDSEF to be evaluated against other IDSs With the exception of the manual interventions (i e , identifying commands and identifying VEP command outputs on successful and failed attacks) between step 1 and 2, this process is completely automated

Thus, this approach allows us to generate a VEP data set using a test criterion and this data set can be automatically updated, documented and used for IDS testing and evaluation The AES/VLab is completely automated and can generate the VEP data set (i e , more than 10 000 test cases) in a

matter of hours. It is flexible since it can also generate data sets for other security-related projects other than the VEP testing model [21, 22].

In the next sections, we describe in more detail some of the key choices made to develop the VEP testing model. The selection of the VEPs used in step 1 (i.e., downloaded by the Web Crawler) is discussed in Section 4.2.1. The test criterion used to generate the experiments (step 2) which also specifies the test requirements, is discussed in Section 4.2.2. The VEP data set generation and documentation process produced by the AES/VLab (step 3) is presented in Section 4.2.3. Step 4 is described in Section 4.2.4, and since step 5 is relatively straightforward, it is not further discussed in this chapter.

4.2.1 Selection of Vulnerability Exploitation Programs

We rapidly realized that in practice, it is not possible to gather VEPs for each vulnerability (i.e., VEPs have not been developed for each know vulnerability) and that there are too many vulnerabilities (more than 34 000 according to SecurityFocus as of March 2009) to manually compile their corresponding VEPs, identify the VEP commands and configure target systems with a vulnerable product. Thus, to make the VEP data set similar to attacks perpetrated on the Internet and to try to be as general as possible, we configured the Web Crawler to only download the VEPs that attempt to exploit vulnerabilities in the most popular key services (e.g., HTTP, FTP, SMTP, NetBios) and that are running on the most common operating systems (e.g., Linux, Windows, FreeBSD).

As explained earlier, the Web Crawler downloads the new VEPs available from well-known web sites such as SecurityFocus and Milworm on a daily basis. We also manually added VEPs from the MetaSploit project and Nessus,[6] since our Web Crawler is currently not designed to download these VEPs automatically. In the case of vulnerability scanners such as Nessus, one needs to be careful to only select the vulnerabilities for which the vulnerability scanner actually exploits a vulnerability. Most of the time, vulnerability scanners only try to identify whether the target system

[6] www.nessus.org

could potentially have a vulnerability by getting the operating system or server versions. This does not constitute an attack (exploitation of a vulnerability) for an IDS. In the case of Nessus, it does not exploit the vulnerabilities for the majority of the vulnerabilities scanned. Thus, we mainly used VEPs from SecurityFocus, Milworm and Metasploit and we only used the vulnerability scanners when they actually exploited the vulnerability.

From all the downloaded VEPs, we only kept the VEPs that included their source code and the ones that we were able to compile without any major modifications. Most VEPs have minor voluntary errors in the source code to ensure that only experienced programmers can use them. As explained before, for security reasons, the source code was essential to understand the VEP capabilities and functionality.

We downloaded around 500 VEPs using this selection procedure posted from 1998 to January 2005. For the first version of the VEP data set, we used 124 VEPs (covering a total of 92 vulnerabilities). These 124 VEPs were the ones that were the most ready for use (i.e., easy to compile, to identify command line to use them and VEP command output on successful and failed attacks). These VEPs attempt to exploit vulnerabilities on the most common products found on the Internet, they attempt to exploit well-known vulnerabilities, they are publicly available and they are easy to use (e.g., they have none or simple compilation errors), thus we believe that these VEPs should be the ones for which IDSs can easily detect and verify their attacks.

4.2.2 Test Criterion

The VEPs often have different possible parameters that change the behaviour of the attacks they generate. For example, a parameter is often used to specify the targeted operating system version, since it is often required to adjust the attack based on that information to have a successful attack. Thus, every possible *VEP configuration* (i.e., the values used with each parameter) represents a different attack that an IDS should be able to detect and verify. Moreover, as described in [9, 10], vulnerable and non-vulnerable target systems should behave differently when they are targeted by

the same attack. Thus, the execution of every possible VEP configuration against different target systems, offering a service targeted by the VEP (e.g., HTTP and FTP), also leads to different attacks that an IDS should be able to detect and verify. These different combinations of attack scenarios (i.e., different VEP configurations against vulnerable and non-vulnerable target systems) are required to assess IDS accuracy. This way, we produce test cases containing failed attacks against vulnerable and non-vulnerable target systems, and successful attacks against vulnerable target systems. We should not expect successful attack against non-vulnerable target systems, but if it happens, our approach will allow us to identify these situations.

Note that VEPs cannot launch attacks against a target system if it is not able to connect to it (e.g., using TCP or UDP). Thus, the relationship between a target opened TCP/UDP port (e.g., HTTP, SMTP, FTP) and the VEP targeted TCP/UDP ports is an important constraint to satisfy: when a VEP configuration is used against a target system, the VEP must be able to launch an attack.

Let T be a set of target systems. For each target system $t \in T$, we denote P_t the set of open TCP/UDP ports (i.e., offering network services) on t. Let V be the set of selected VEPs. For each VEP in $v \in V$, we denote P_v the set of TCP/UDP ports used by v and C_v the set of possible configurations of v. For the VEP testing model, we define a test requirement as an attack scenario (i.e., VEP + VEP configuration + target system). For each of those test requirements, a test case must be generated. We denote the set of test requirements by TR. Here is the definition of the All Attack Scenario Coverage criterion:

Definition 1

All Attack Scenario Coverage (AASC): For each $t \in T$ and $v \in V$, TR contains a test requirement for each pair (p, c) such that $p \in P_t \cap P_v$ and $c \in C_v$.

The AASC criterion maximizes the diversity of attack scenarios for the IDS under test because every combination of VEP configurations is used against every compatible target systems (i.e., target

system offering a service on the VEP targeted port). However, in practice, it is almost impossible to meet the AASC criterion. For instance, it is difficult, if not impossible, to install and to configure in every possible configuration all possible target systems. Moreover, some VEP parameters can take an infinite or large (e.g., more than 1000) number of values, which could each be considered a separate configuration.

Consequently, we propose another test criterion called Specified Attack Scenario Coverage criterion that is subsumed by the AASC and were T and C_v are user defined parameters to ensure that the criterion is possible in practice and that at least the set of target systems and VEP configuration are well-documented. Using a criterion with user defined parameters is a common solution to this type of problem in software testing [50].

Let T^u be a set of target systems specified by the user and C_v^u the set of user defined possible configuration for v.

Definition 2

Specified Attack Scenario Coverage (SASC): For each $t \in T^u$ and $v \in V$, TR contains a test requirement for each pair (p, c) such that $p \in P_t \cap P_v$ and $c \in C_v^u$.

For T^u, we thus restricted the target system configuration to the ones that we were able to install in a reasonable amount of time. We systematically installed and configured every Windows, RedHat Linux, SUSE Linux, FreeBSD, NetBSD and OpenBSD operating systems available. Every operating system and network services was installed using its default installation and configuration. This corresponds to 108 different operating system versions.

For C_v^u, we only considered the VEP configurations using parameters that have a fixed and limited list (i.e., less than 25) of possible values (e.g., the targeted operating system versions). Thus, in the remaining of this chapter, the term *VEP configuration* refers to those configurations. For the parameters that have a finite, but too large number of choices (i.e., more than 25), we used the default value provided with the VEP when it was available. For example, we used the default value

for the parameters that require a TCP/UDP port number (e.g. to specify the port number on which the VEP opens a shell on the target system after the attack). If a default value is not available we made an arbitrary choice.

Using the SASC criterion, the Experiment Generator generated experiments that are uniquely identified by their VEP, their VEP configuration and the target system used. Every experiment performed by the AES/VLab produces a traffic trace (test case) in the VEP data set. The first version of the VEP data set (i.e., the one presented in this chapter) was generated using 124 VEPs against 108 target systems and contains 10 446 *tcpdump* traffic traces (test cases) that can be replayed to IDSs to test and evaluate their detection and verification abilities. This represents between 2 to 12 times the number of VEPs, and between 12 to 36 times the number of target systems typically used for generating a VEP data set in the literature (Section 3.1.1). Moreover, our VEP data set also includes failed attacks against vulnerable and non-vulnerable target systems (the other VEP data sets do not assess the verification problem), making our VEP data set one of the most complete VEP data set publicly available for researchers. Note that 10 446 does not equal 124 x 108 since different VEPs have more than one configuration and that not all target systems have all the open ports that are targeted by the VEPs.

4.2.3 Automatic Experimentation System/Virtual Laboratory Overview: Generation of the VEP Data Set

In this section, we present the Automatic Experimentation System / Virtual Laboratory (AES/VLab) that was used to generate the VEP data set.

Section 4.2.3.1 presents the requirements and the motivations for the development of the AES/VLab and for using it in our VEP testing model. Section 4.2.3.2 presents the language we developed to specify the experiments. Section 4.2.3.3 describes how an experiment is conducted by the AES/VLab. Section 4.2.3.4 describes the data set documentation process of the experiment results.

The approach behind the AES/VLab goes beyond its usage in the VEP testing model. This

approach is proposed as one of the solutions to address the problem faced by researchers related to the dynamic nature of the accuracy problem.

4.2.3.1 Requirements

As explained in the introduction of this thesis, the current reality in network security is that several vulnerabilities are identified daily in software programs commonly used by computer systems connected on the Internet (e.g., by SecurityFocus, CVE). Moreover, millions of malware programs have now been identified that can infiltrate vulnerable computer systems without the users' consent to control these systems.

Being able to use these malware programs (e.g., VEPs) to reproduce different network scenarios (e.g., a VEP attack against different network services, studying worm propagating in a network) for conducting network security research provides insightful and useful information to researchers. However, researchers often neglect the dynamic nature of network security based on the intense manual process required to test and assess their proposed solutions. We believe that this situation partially explains the small number of VEPs and target system combinations used to assess IDSs (Section 3.1.1).

To address these shortcomings, we developed the AES/VLab.[7] The AES/VLab is designed so that its requirements are more generic than those related to the VEP testing model, while still taking them into account. It is inspired from research fields such as Zoology,[8] where scientists study the behaviour and evolution of animals (e.g., in our cases VEPs, malware) in a specific environment (e.g. in our case, in a computer network); and Microbiology, where scientists use systems to automate their experiments by strategically combining, for example, different proteins located in different test tubes.

The AES/VLab is composed of two modules: the AES and the VLab. The VLab system is the environment where the experiment is conducted and the AES is the system that automatically

[7] Previously called Virtual Network Infrastructure (VNI) in [20].
[8] `en.wikipedia.org/wiki/Zoology`

conducts the experiments within the VLab environment. To create environments to study computer programs, emulators or virtual machines are often used. For instance, they are used by the security community to construct virtual networks as they provide traffic propagation control and reduce computer resources. We selected VMware,[9] among others (e.g. User-mode Linux,[10] Bochs,[11] Virtual PC[12] and VirtualBox[13]), as we already had a positive experience with it [51]. The VLab system provides a database of virtual machines and an environment for the AES to construct a virtual network where the experiment is conducted. The current version of AES/VLab works with either VMware Workstation or VMware ESX. We are aware that some malware do not execute normally (e.g., stop their execution) when they detect that they are executed in a virtual machine. However, in the case of VEPs, this should not cause any problem since all the VEPs used in this study come with their source code.

To generate a VEP data set (i.e., step 3 of our VEP testing model), a controlled network infrastructure executing all the VEPs was developed to allow: (req. 1) recording of all network traffic; (req. 2) control of network traffic noise; (req. 3) control of attack propagation; (req. 4) usage of real and heterogeneous system configurations; (req. 5) fast recovery to initial condition state; and (req. 6) automation and parallelization of VEP experimentation to account for the large number of VEP test cases required by the SASC criterion (Section 4.2.2).

The AES/VLab meets all the requirements specified above to generate test cases for the VEP testing model. The VLab module provides a virtual network environment that allows the capture of all network traffic. These traffic traces can then be used to study attack behaviour (req. 1). Thanks to the virtual network environment, our AES/VLab also allows us to control the network traffic to create clean traffic traces that only contain network traffic relevant to the attack scenarios generated by the executed VEP (req. 2). With VMware, the attack propagation is confined, thus preventing

[9] www.vmware.com
[10] user-mode-linux.sourceforge.net
[11] bochs.sourceforge.net
[12] Microsoft.com/Canada/Virtualization
[13] www.irtualbox.org

infection of the computers running the AES/VLab (req. 3). VMware facilitates the creation of template virtual machines having different software configurations (operating system, services, etc). Thus, it allows the creation of a database of virtual machine templates that can be used to rapidly deploy custom network configurations within a single computer (req. 4). Also, VMware snapshots allow restoration of the test network to the state it was in before each attack attempt. All attack scenarios can then be performed under the same initial conditions (req. 5). Finally, the AES module allows to automatically conduct experiments in parallel (using multiple AES instances on multiple computers) and to automatically document these experiment results for the user to review (req. 6).

4.2.3.2 Experiment Language

As explained before, the AES module is used to automatically conduct the experiments provided by the user, and to document the results. These experiments are generated by the Experiment Generator based on the SASC criterion. These experiments are specified in XML files using our own experiment language. The experiment language requires to specify the network topology and an Experiment Execution Graph (EEG).

The network topology specifies the virtual machines required by the experiment and how these virtual machines are connected together. The virtual machines that are part of an experiment are called *actors*. Each *actor* has a set of network interfaces identified by a unique identifier and are connected on a specific network. Thus, the actors and the network interfaces specify the environment (i.e., network topology) where the experiment will take place.

The Experiment Execution Graph specifies the different steps of the experiment and the order in which they are executed. The EEG is composed of numbered steps, each separated by an operator. The EEG supports two operators: the sequencing operator (i.e., .) and the concurrency operator (i.e., |).

- **Sequencing (.):** The sequencing operator specifies an ordering of the steps. For example, the EEG $S_1.S_2$ specifies that S_1 has to be performed first and then S_2 can be performed. The

sequencing operator also specifies that S_2 cannot be performed before S_1 is completed.

- **Concurrency (|):** The concurrency operator specifies that two steps can be performed in parallel. For example, the EEG $S_1.(S_2 \mid S_3).S_4$ specifies that S_1 has to be performed and completed first, then S_2 and S_3 can be performed concurrently and finally S_4 can be performed when both S_2 and S_3 are completed.

Figure 4.2 shows a simplified example of the experiment used to execute the VEP `jill` against Windows 2000 Server. Lines 2 to 7 specify the network topology. There are three actors: `VMWin2000ServerTarget` (i.e., the target system), `VMLinuxRH80Attack` (i.e., the attacker system) and `VMWin2000ServerDNS` (i.e., the Domain Name System used to resolve computer names). Each actor network interface is connected to the virtual network (i.e., `vmnet`) 1. This network topology is shown in Figure 4.3. Line 1 specifies the EEG is composed of 5 steps and lines 8 to 12 specify each step of the EEG. Each step is associated with an actor (e.g., `actor="VMLinuxRH80Attack"`) that has to execute a specific command (e.g., `cmd="jill 10.92.39.14 80 10.92.39.3 30"`). Steps S_1 and S_2 (lines 8 and 9) are used to check whether the attacker system (i.e., `VMLinuxRH80Attack`) has IP connectivity to the target system and the Domain Name System (DNS). To accomplish this task, the attacker system uses the command `ping` with the target system (S_1) and DNS (S_2) IP addresses. Step S_3 (line 10) instructs the attacker system to start capturing the network traffic that will be generated by the VEP. Step S_4 (line 11) specifies to execute the VEP `jill` against the target system. Step S_5 (line 12) specifies to stop capturing the network traffic.

Each experiment used to generate the VEP data set is similar to this one. The only difference is the target system used and the command used in step S_4 (i.e., the VEP and the VEP configuration used). It is important to note that some optional steps are sometimes added to experiments since they are required by some VEPs to complete the attack and ensure its success. For example, to be sure of the success of the attack, we specified in some experiments that the attacker has to connect, after the attack, to a remote shell on the target system (installed by the VEP during the attack) or

we specified that the attacker has to check if the target system is still available after a DOS attack.

```
1   <experiment execution graph="S1.S2.S3.S4.S5"/>
2   <actor id="VMWin2000ServerTarget">
3       <nic interface="1" VMNet="1" />
4   <actor id="VMLinuxRH80Attack">
5       <nic interface="1" VMNet="1" />
6   <actor id="VMWin2000ServerDNS">
7       <nic interface="1" VMNet="1" />
8   <step id="S1" actor="VMLinuxRH80Attack" cmd="ping 10.92.39.14"/>
9   <step id="S2" actor="VMLinuxRH80Attack" cmd="ping 10.92.36.1"/>
10  <step id="S3" actor="VMLinuxRH80Attack" cmd="START SNIFFER"/>
11  <step id="S4" actor="VMLinuxRH80Attack" cmd="jill 10.92.39.14 80 10.92.39.3 30"/>
12  <step id="S5" actor="VMLinuxRH80Attack" cmd="STOP SNIFFER"/>
```

Figure 4.2: XML Experiment File Example.

4.2.3.3 Experimentation

In this section, we present the sequence of actions that the AES/VLab has to accomplish to fulfill a set of experiments (i.e., generate the VEP data set).

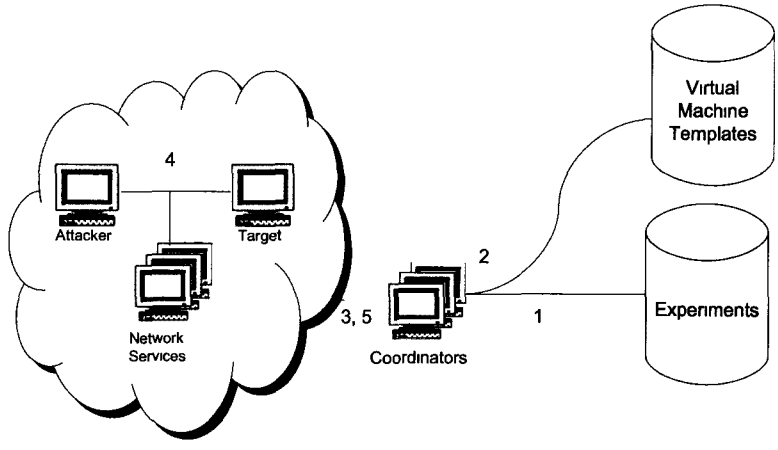

Figure 4.3: Automatic Experimentation System/Virtual Laboratory used for the VEP model

Figure 4.3 presents the AES/VLab when conducting experiments to generate the VEP data set. The coordinators are computer systems running the AES code and using the VLab Each coordinator is able to execute one experiment at a time.[14]

[14]We currently have 30 coordinators running on 30 Pentium 4 with 2 Gigs of memory that can execute 30 experiments in parallel

As explained in Section 4.2.3.2, in the context of the VEP model, the network topology part of each experiment contains an attack system (Attacker), a target system (Target) and network infrastructure services (DNS). The attack systems are used to launch attacks against the target systems by using VEPs.

Each action taken by the AES/VLab to conduct an experiment is indicated in Figure 4.3. All links between the involved entities for actions 1 to 5 is repeated for each experiment. Although the descriptions of the AES/VLab steps below are presented in the context of experiments to generate the VEP data set, the same actions are performed by the AES/VLab for any experiment.

1. *Experiment Negociation.* Each coordinator negotiates the experiment to conduct with the other coordinators to ensure that it does not conduct an experiment that was already conducted or that is currently conducted by another coordinator. When the coordinator has identified an experiment that is available, it *locks* the experiment so no other coordinator can conduct it.

2. *VLab Setup.* Based on the network topology of the experiment, the coordinator builds the virtual network where the experiment is to be conducted.

3. *Opening Communication Channel.* Once the virtual network is ready, the coordinator opens the communication channels with the actors of the experiments. These communication channels (e.g. a hard drive shared via VMware, a virtual parallel port, a virtual serial port, etc.) are the only way the coordinator can communicate the experiment steps to be executed to the actors. This enables to isolate the virtual network (from a networking point of view) while keeping some communication capabilities.

4. *Experiment Execution.* The coordinator executes the EEG by providing commands to the virtual machine in the VLab. As described earlier, in the context of the VEP model, the coordinator first asks the attack machine to check if it is able to communicate with the target system and the DNS (i.e., check the network connectivity). Then, it asks the attack machine to begin recording the network traffic and to attack the target system using the VEP specified

in the experiment. When the attack is completed, the coordinator asks the attack machine to stop capturing network traffic.

5. *Tear Down.* The coordinator gathers from the VLab the experiment results (e.g., the traffic traces, the VEP output) using the communication channel. Then, the coordinator documents and stores the experiment results (e.g. populating the VEP data set) and restores the virtual machines (e.g., the attacker and target systems) to their initial state to avoid side effects (e.g., impact on the next attack).

4.2.3.4 Documention Process

After an experiment is completed by the AES/VLab, its results are documented. In this case, the AES/VLab documents the traffic traces (i.e., test cases) that contain the execution of the VEP attacks. This documentation process is essential for another computer system to automatically use the experiment results (e.g., see the IDSEF description in Section 4.2.4). It is also important to document the experiment to share our results with other researchers (e.g., to make a VEP data set publicly available) so that they can understand and reproduce our experiment. In our VEP data set, each traffic trace is documented by four characteristics: the target system configuration, the VEP configuration, whether or not the target has the vulnerability exploited by the VEP and the success of the attack (see example in Figure 4.4).

The target system configuration description includes the list of installed software (e.g., the operating system, the different network services and their versions), as well as its IP configuration. The VEP configuration defines the options used to launch the attack. The Vulnerable field specifies whether or not this target system uses a product (e.g., Apache, IIS) that is known to have a vulnerability exploited by that particular VEP. To determine whether the attacks have been successful or not, we use the attacker's point of view. Thus, we use the outputs of the VEP command that launches the attacks (e.g., the attack is successful when the attacker is able to get a remote shell on the target system) and the effects, noticeable by the attacker, of the attack on the target system

System Configuration
IP: 10.92.39.14
Name: VMWin2000ServerTarget
Operating System: Windows 2000 Server
Ports:
21/tcp Microsoft IIS FTP Server 5.0
25/tcp Microsoft IIS SMTP Service 5.0
80/tcp Microsoft IIS Web Server 5.0
Vulnerability Exploitation Program Configuration
name: jill.c
reference: BID,2674 CVE,2001-0241
command: jill 10.92.39.14 80 10.92.39.3 30
Vulnerable: yes
Success: yes

Figure 4.4: Test Case Label Example

(e.g., the target system is no longer available in the case of a DOS attack).

4.2.4 IDS Evaluation Framework Overview

To automatically test IDSs using our detection and verification metrics, we developed an IDS Evaluation Framework (IDSEF). This IDSEF consists of five components: the *IDSEvaluator*, the *Data Set*, the *Test Oracle*, the *IDSResultAnalyzer* and the *IDS*. Figure 4.5 presents our IDS Evaluation Framework.

First, the *IDSEvaluator* takes each test case in the *Data Set* and provides it to the tested *IDSs* (steps 1 and 2). We use the words *data set* to represent a group of test cases (traffic traces). More precisely, the *Data Set* is a test suite generated by a testing model. Then, the IDS events generated by the *IDSs* (i.e., the predicted values) are collected and provided to the *IDSResultAnalyzer* (step 3) with the description (i.e., the actual values) of each test case (step 4). The *IDSResultAnalyzer* compares the predicted and actual values for each test case thanks to a test oracle (Section 4.3.1.3). Then it uses the detection and verification metrics to calculate the detection and verification accuracy (step 5).

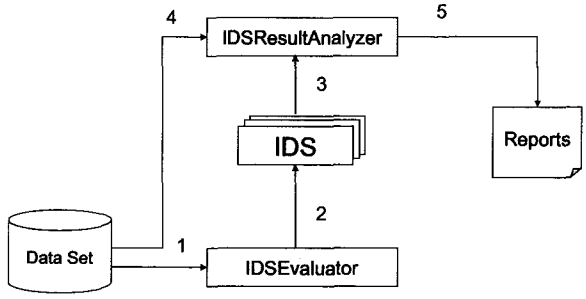

Figure 4.5: IDS Evaluation Framework

4.3 Case Study

In this section, we present the evaluation of the detection (i.e., more specifically the evasion problem) and verification problems using the VEP data set and the IDSEF on two widely used and maintained IDSs (i.e., Snort and Bro). For this case study, we used Snort 2.3.2 (released in March 2005), and Bro 0.9a9 (released in May 2005) since these were the versions available at the time this case study was conducted [20].

In the case of Snort, we also performed this evaluation on older (e.g., 2.2.0, 2.3.0, 2.3.1) and more recent versions (e.g., 2.8.5 with signature databases released in 2009 and 2010) as they became available, to show the effectiveness of our VEP testing model in producing up-to-date analysis of the detection and verification problems. Also, our VEP data set is regularly updated so that these IDSs are retested using the traffic traces generated from new VEPs downloaded from the Internet. Consequently, while the results presented in this section are the ones presented in [20], we continued our analysis of the detection and verification problems with newer VEPs and IDSs. We obtained results similar to the ones presented in this section for more recent IDS versions.

The primary objective of this case study is to show that our VEP testing model can be used to assess detection and verification problems in IDSs, to help identify the root causes of these problems and to identify the key detection and verification problems in IDSs (i.e., research methodology, see Section 1.2). Thus, the objective is not to compare Snort and Bro to decide which IDS is better or worse than the other.

4.3.1 Design of the Experiment

4.3.1.1 VEP Data Set Generation

We used the AES/VLab to generate IDS test cases (i.e., our VEP data set) using the 124 VEPs (covering a total of 92 vulnerabilities) downloaded based on the selection criterion presented in Section 4.2.1. We used these 124 VEPs against 108 target systems using the SASC criterion (Section 4.2.2). Within a week, this resulted in a VEP data set of 10 446 test cases (i.e., documented traffic traces) in which 420 attacks succeeded and 10026 attacks failed. This situation was expected based on the SASC criterion because only a few of the 108 target systems are vulnerable to each VEP. We then only selected the test cases that were generated by VEPs associated with vulnerabilities that were released before or around (i.e., during the same month) the release of Snort 2.3.2 and Bro 0.9a9 to provide a fair evaluation of these IDSs. As a result, this subset of the VEP data used in this case study includes 102 VEPs. We then used this subset of the VEP data set with the IDSEF presented in Section 4.2.4 to assess the detection and verification accuracy of Snort 2.3.2 and Bro 0.9a9.

4.3.1.2 IDS Signature Database

In the case of Snort, we used Snort version 2.3.2 with the signatures included in the release. Bro does not come with its own signature database such as Snort (Section 2.4). Bro 0.9a9 comes with a signature database of Snort signatures translated into the Bro signature language. However, these translated Snort signatures available in Bro 0.9a9 are older than the signatures available in Snort 2.3.2. One approach to add signatures to Bro is to use the *s2b* package to convert Snort signatures into the Bro signature language. We thus used *s2b* to update the Bro signature database with the Snort 2.3.2 signature database, making it more up-to-date. During convertion the Snort signatures are enhanced to reduce the number of FP_V by adding predicates to the Snort signatures that verify the success or failure of an attack. One difficulty we encountered was that the *s2b* converter was

not able to convert all Snort plug-ins such as `byte_test`, `byte_jump`, `isadataat`, `pcre`, `window` and `flowbits`. In these cases, we manually translated the corresponding Snort signatures into the Bro signature language.

4.3.1.3 Test Oracle

The IDSEF uses a specific test oracle to assess the detection and verification accuracy of IDSs. In the case of the detection accuracy, the detection confusion matrix (Section 2.5) requires to classify the test cases as *Attack* or *Normal*. All the test cases generated by the VEP test model are classified as *Attack* because this testing model does not produce test cases with normal traffic. Thus, for a VEP testing model there is no FP_D and TN_D in the assessment of the detection accuracy. The detection confusion matrix also requires to classify the results provided by the IDSs (i.e., the IDS events) into *Detected* (i.e., TP_D) or *Not Detected* (i.e., FN_D). The `reference` field in the test case documentation (Section 4.2.3.4) is used by the oracle for this purpose. A test case is classified as *Detected* when at least one IDS event provided by the IDS has a BID or CVE number equals to the one in the test case documentation (i.e., equals to the vulnerability exploited by the VEP attack contained in the test case). Otherwise the test case is classified as *Not Detected*.

For example, when Snort provides the IDS event specified in Figure 4.6 for the test case documented in Figure 4.4, the IDSEF classifies this as a TP_D since the test case contains an attack and its vulnerability reference (i.e., BID 2674, CVE 2001-0241) equals to the vulnerability reference of the IDS event (i.e., line 3 and 4).

```
1   [**] [1:971:9] WEB-IIS ISAPI .printer access [**]
2   ...
3   [Xref => http://www.securityfocus.com/bid/2674]
4   [Xref => http://cve.mitre.org/cgi-bin/cvename.cgi?name=2001-0241]
```

Figure 4.6: Snort Event 971 (excerpts)

Snort 2.3.2 and Bro 0.9a9 do not provide vulnerability references for all possible IDS events. This is mainly due to improper documentation of the signatures by the signature developers. For instance,

it is possible that one signature is used to detect attacks for many different vulnerabilities and that the signature developers did not include all the vulnerability references or did not include any. Consequently, when a test case was classified as *Not Detected* by the IDS, we manually investigated the resulting IDS events to see if at least one of them could be related to the vulnerability exploited in the test case. If so, we manually added the vulnerability references to the corresponding IDS signature. This manual process was not very time consuming since nearly all the Snort signatures were well-documented.

Moreover, Snort 2.3.2 and Bro 0.9a9 do not provide detailed enough IDS events to precisely classify all the test cases as *Not Detected* because they do not provide IDS events (i.e., they remain silent) when a failed attack attempt is detected. Thus it is difficult to know if they do not provide an IDS event because they cannot detect the attack (i.e., FN_D) or because they identify that the attack failed (i.e., TN_V). To resolve this issue, we used an *optimistic* assessment of their detection accuracy. To accomplish that, we group together the test cases related to the same VEP and then proceed as follow: (1) if one VEP test cases containing a successful attack is classified as *Detected* by the IDS (i.e., TP_D), we consider as *Detected* (i.e., TP_D) all the test cases generated from the same VEP containing failed attack attempts, for which the IDS remains silent; (2) otherwise, (i.e., no VEP test case containing a successful attack is classified as *Detected* by the IDS), the test cases containing failed attack attempts from the same VEP are classified as *Not Detected* (i.e., FN_D).

Using this optimistic assessment of the detection accuracy led to inconclusive results for 18 of the 102 VEPs. These VEPs did not generate successful attacks against any target systems and these attacks were always undetected by both Snort and Bro (i.e., they remained silent). As a result, manual investigation had to be performed to determine whether the IDSs identified that the VEPs failed and remained silent or whether the IDSs did not have any signature in their database to detect those attacks. We are inclined to say that these two IDSs do not have any signature in their database for these attacks because a quick search (thought not exhaustive) in their respective signature databases did not provide any signature related to the vulnerability identifiers associated

with any of the 18 VEPs. As a result, to keep our analysis focused on conclusive results, we decided to remove these VEPs from the rest of the discussion, leaving 84 VEPs for this analysis.

In the case of the verification accuracy, the verification confusion matrix (as defined in Section 2.5) requires to classify the test case as *Successful* or *Failed*. The `Success` field in the test case documentation (Section 4.2.3.4) provides us with this information: expected outcome used by the oracle. The verification confusion matrix also requires to classify the results provided by the IDS into *Positive* or *Negative*. Snort and Bro remain silent when they are able to verify a failed attack and they provide an IDS event when they believe the attack is successful. As a result, the test cases for which these IDSs provide an IDS event equals the vulnerability exploited in the test case (i.e., the IDS provides an IDS event related the `reference` field in the test case documentation) are classified as *Positive* and the test cases for which the IDS remains *Silent* (i.e., the IDS does not provide any IDS event with a BID or CVE number equal to the `reference` field in the test case documentation) are classified as *Negative*.

4.3.2 Results

In this section, we present the results obtained for the detection accuracy (Section 4.3.2.1) and verification accuracy (Section 4.3.2.2) of Snort and Bro using our VEP testing model.

4.3.2.1 Detection Problem

Table 4.1 presents the detection accuracy (A_D) for Snort and Bro. As explained, one limitation of the VEP testing model is that the generated test cases only contain attack traffic and no normal traffic. Thus, we do not assess the extra-detection problem (i.e., the TN_D rate) and the A_D is equal to the detection TP_D rate (Section 2.5). Nevertheless, we can still assess the evasion problem (as defined in Section 1.1). With only test cases containing attacks, we can still derive interesting conclusions about these two IDSs for the detection problem.

It seems that even if the VEPs used to generate this data set are well-known by the IDS developer,

Snort			Bro		
A_D	TN_D rate	TP_D rate	A_D	TN_D rate	TP_D rate
78%	N/A	78%	76%	N/A	76%

Table 4.1: Detection Accuracy for Snort and Bro

some of them are missed by the IDSs. Snort only detects 78% of the attack contained in the VEP test cases while Bro only detects 76% of these attacks. More precisely, for at least 15 out of the 84 VEPs used in this data set, both tested IDSs have detection problems (i.e., both IDSs do not detect or only partially detect the attack generated by these VEPs).

A more in-depth analysis provides additional interesting results. VEPs such as `samba_exp2.tar.gz`, `THCIISSLame.c`, `lsass_ms0411.pm`, `DComExpl_UnixWin32.zip` and `HOD-ms04011-lsasrv-expl.c` evade intrusion detection by both IDSs even if some other VEPs that are related to the same vulnerability are detected by both IDSs.

Some other VEPs were released as early as 1997 and the IDSs do not seem to be able to detect them. In the case of `winnuke._eci.c` and `winnuke.pl`, the signature that detects these attacks is commented in the Snort and Bro signature database since it would have generated a lot of FP_D. The signature specifies that for this attack to happen, a packet has to be sent to any target on port 135 or 139 and the TCP `urgent` flag has to be present with any other TCP flags, which is indeed a behaviour common in normal traffic.

For some VEPs, only one of the two IDSs detects the attack. First, only `msrpc_dcom_ms03_26.pm` is not detected by Snort though it is detected by Bro. How come Bro using the Snort signatures is able to detect an attack and Snort does not? In this case, *s2b* was only able to partially translate (some plug-ins were omitted by *s2b*) the Snort signature that detects the attack against BID 8205 into a Bro signature, and the omitted plug-ins were specifying to the Snort engine that the packets generated by `msrpc_dcom_ms03_26.pm` were not an attack (which is not the case). Without these incorrectly specified plug-ins Bro was able to detect this attack.

Second, Bro did not detect three VEPs that are detected by Snort. These are `kod.c`, `kox.c` and `pimp.c` which all use the IGMP protocol for their attack scenarios (these three VEPs exploit

BID 514). They are detected by Snort, but not by Bro because Bro does not have any module in its packet decoder to analyze IGMP, even if the corresponding signature does not use any IGMP protocol fields. The signatures use the value of the protocol number field in the IP header to identify the IGMP protocol. Even if Bro does not return any error message when this signature is in its signature database, and even if this signature complies with the Bro signature language, the Bro IDS engine is unable to use it. This is a reminder that even when a signature is correctly specified, missing components in the IDS engine or configuration errors could cause FN_D. We propose a test model in Chapter 5 to systematically assess and identify this type of problem.

Third, some of the *s2b* signature enhancements prevented Bro from detecting successful attacks generated by five VEPs. For instance, the successful attacks from `sol2k.c`, `iiswebexplt.pl`, `iis50_printer_overflow.pm`, and `jill.c` evade Bro detection (these four VEPs exploit BID 2674). The enhanced version of this signature by Bro is triggered if no error message is provided by the targeted server after the attack and if the targeted server is a Microsoft IIS HTTP Server. However, in the case of this particular group of attacks, no information is provided by the server to identify it as IIS when the attack is successful. Thus, Bro sees the error message, but it is not able to identify this server as IIS, the signature is never triggered and we have FN_D when the attacks are successful. The VEP named `m00-apache-w00t.c` also evades detection when Bro is used, but for a different reason. Once again, the enhanced version of the signature for detecting a successful attack is triggered if no error message is provided by the targeted server after the attack. However, it is the type of error message given by the targeted server that represents the exploitation of this vulnerability. This VEP is used to find the name of users allowed on the server. If a request is made with a particular user name, the error message `403 Forbidden Access` specifies to the VEP that this user name is a valid user on this server. Thus, Bro once again thinks the attack failed, does not provide any IDS event, and thus does not detect the successful attack.

Table 4.2 summarizes these results and provides the list of VEPs that evade detection for at least Snort or Bro with their corresponding BID.

		Snort			Bro		
VEP	BID	A_D	TN_D rate	TP_D rate	A_D	TN_D rate	TP_D rate
DComExpl_UnixWin32.zip	8205	0%	N/A	0%	0%	N/A	0%
HOD-ms04011-lsasrv-expl.c	10108	0%	N/A	0%	0%	N/A	0%
HOD-ms04031-expl.c	11372	0%	N/A	0%	0%	N/A	0%
lsass_ms0411.pm	10108	0%	N/A	0%	0%	N/A	0%
ms05_039_pnp.pm	14513	0%	N/A	0%	0%	N/A	0%
mssql2000_preauthentication.pm	5411	0%	N/A	0%	0%	N/A	0%
mssql2000_resolution	5311	0%	N/A	0%	0%	N/A	0%
MultiWinNuke.c	6005	0%	N/A	0%	0%	N/A	0%
samba_exp2.tar.gz	7294	0%	N/A	0%	0%	N/A	0%
THCIISSLame.c	10116	0%	N/A	0%	0%	N/A	0%
winnuke._eci.c	6005	0%	N/A	0%	0%	N/A	0%
winnuke.c	6005	0%	N/A	0%	0%	N/A	0%
winnuke.pl	2010	0%	N/A	0%	0%	N/A	0%
zp-exp-telnetd.c	3064	0%	N/A	0%	0%	N/A	0%
msrpc_dcom_ms03_26.pm	8205	0%	N/A	0%	100%	N/A	100%
apache2.pl	2503	94%	N/A	94%	83%	N/A	83%
kod.c	514	100%	N/A	100%	0%	N/A	0%
kox.c	514	100%	N/A	100%	0%	N/A	0%
pimp.c	8205	100%	N/A	100%	0%	N/A	0%
iis50_printer_overflow.pm	2674	100%	N/A	100%	97%	N/A	97%
iiswebexplt.pl	2674	100%	N/A	100%	74%	N/A	74%
sol2k.c	2674	100%	N/A	100%	94%	N/A	94%
jill.c	2674	100%	N/A	100%	89%	N/A	89%
m00-apache-w00t.c	3335	100%	N/A	100%	74%	N/A	74%

Table 4.2: VEP Detection Problem for Snort and Bro

As a conclusion, using our VEP testing model, we identified that Bro is unable to detect the attacks of 18 VEPs and it can partially detect the attacks of 6 VEPs. Thus, Bro is able to completely detect the attacks of 60 of the 84 VEPs. Snort is unable to detect attacks of 16 VEPs and it partially detects attacks of one VEP. Thus, Snort is able to completely detect the attack of 68 of the 84 VEPs.

We could have expected Bro to be as accurate as Snort for the detection problem. Indeed, the Bro signatures used in this case study should have been the same (with enhancement for attack verification) as the ones in the Snort signature database if *s2b* correctly converted the signature specification from Snort to Bro. Problems in the signature enhancement for attack verification added by *s2b* and errors in the interpretation of signatures by the Bro engine lead to FN_D.

Moreover, we believe that Bro should have a lower TN_D rate than Snort (although this is untested in this case study). Some Snort plug-ins (Section 4.3.1) contained in many Snort signatures are not converted by *s2b* into the Bro signature language, making the Bro signatures strictly less restrictive (i.e., less constraints imposed to specify the attack) than the corresponding Snort signatures, and

thus making the Bro signatures more likely to cause FP_D than the Snort signatures.

4.3.2.2 Verification Problem

In this section, we analyze the results provided by Snort and Bro for the verification problem. Table 4.3 presents the verification accuracy (A_V), the TN_V rate and the TP_V rate for Snort and Bro using our VEP data set. One of the main results that emerged from this experiment is that Snort and Bro partially address the verification problem based on our VEP data set. In some situations, they raise IDS events regardless of the success of the attack. This is true for most of the VEPs because Bro does not have an A_V of 100% for 49 of the 66 VEPs it is able to detect and Snort does not have an A_V of 100% for 68 of the 69 VEPs it is able to detect.

Snort			Bro		
A_V	TN_V rate	TP_V rate	A_V	TN_V rate	TP_V rate
16%	11%	100%	45%	42%	91%

Table 4.3: Verification Accuracy for Snort and Bro

The evaluation of the verification accuracy of these two IDSs is classified into three groups of results. First, we present the results for which both IDSs have the same verification accuracy (21 VEPs). For only one of those VEPs both IDSs have a A_V of 100% (i.e., a TN_V and TP_V of 100%). For the other VEPs attacks, the IDSs generated IDS events regardless of the success or failure of the attack: both IDSs provided the same IDS event for successful and failed attacks. In this case, a network administrator cannot know if the attack succeeded or failed (low TN_V rate). Table 4.4 reports on those 20 VEPs for which both Snort and Bro cannot properly address the verification problem.

For the second group, we observed that *s2b* mainly adds predicates in the Snort signatures related to two aspects: the target reaction and the server context information. For example, in the case of an attack over HTTP, additional information on the target reaction is based on the hypothesis that if an attack succeeds, the server replies with a positive response such as message code `200 OK` and if the attack fails, we get an error message back from the server such as `403 Access Forbidden`.

		Snort/Bro		
VEP	BID	A_V	TN_V	TP_V
0x333hate.c	7294	10%	0%	100%
0x82-Remote.54AAb4.xpl.c	7294	3%	0%	100%
0x82-w0000u~happy_new.c	8315	0%	0%	N/A
0x82-wu262.c	8315	0%	0%	N/A
ftpglob.nasl	3581	13%	3%	100%
ms03-04.W2kFR.c	8459	0%	0%	N/A
ms03-043.c	8826	0%	0%	N/A
ms04-007-dos.c	9635	18%	18%	N/A
msasn1_ms04_007_killbill.pm	9633	63%	57%	100%
msdtc_dos.nasl	4006	0%	0%	N/A
msftp_dos.pl	4482	5%	100%	0%
msftp_fuzz.pl	4482	5%	100%	0%
RFParalyze.c	1163	16%	13%	100%
rfposion.py	754	9%	9%	N/A
sambal.c	7294	8%	100%	3%
smbnuke.c	5556	80%	77%	100%
sslbomb.c	10115	70%	100%	67%
wins.c	11763	50%	0%	100%
wins_ms04_045.pm	11763	50%	0%	100%
win_msrpc_lsass_ms04-11_ex.c	10108	18%	0%	100%

Table 4.4: Equivalent Verification Accuracy for Snort and Bro

For instance, added server context information is based on the hypothesis that knowing contextual network configuration such as the type (e.g., IIS, Apache) and version of the attacked server could reduce FP_V because only specific product types and versions are vulnerable to a specific attack. In fact, for 30 VEPs, Bro has improved the A_V of Snort signatures (Table 4.5). However, as explained in Section 4.3.2.1, these improvements to the Snort signatures caused problems for five VEPs.

Third, some Snort signatures also rely on the target reaction (i.e., comparing the server response with the client requests) to assess the success of an attack. This is implement using the Snort `flowbits` plug-ins. The *s2b* tool is not able to translate the `flowbits` plug-ins into the Bro signature language. This enhancement by Snort is effective because for six of the VEPs, Snort improved the A_V compared to Bro (Table 4.6). However, the TN_V rate is still low even with this improvement.

4.3.3 Discussion

Our assessment of the detection (i.e., more specifically the evasion problem) and verification problems, using our VEP testing model and VEP data set, showed that Snort 2.3.2 and Bro 0.9a9 cannot detect more than 22% of the test cases and that Snort and Bro are limited when verifying attacks.

		Snort			Bro		
VEP	BID	A_V	TN_V	TP_V	A_V	TN_V	TP_V
ALL_UNIEXP.C	1806	17%	0%	100%	20%	3%	100%
Apache_chunked_win32.pm	5033	18%	0%	N/A	60%	60%	N/A
DDK-IIS.c	4485	0%	0%	N/A	89%	89%	N/A
decodecheck.pl	2708	18%	0%	100%	91%	88%	100%
decodexecute.pl	2708	8%	0%	100%	92%	90%	100%
execiis.c	2708	17%	0%	100%	100%	100%	100%
fpse2000ex.c	2906	0%	0%	N/A	100%	100%	N/A
iis-zang.c	1806	11%	0%	100%	43%	35%	100%
iis40_htr.pm	307	0%	0%	N/A	39%	39%	N/A
IIS5.0_SSL.c	10115	0%	0%	N/A	100%	100%	N/A
iis5hack.pl	2674	0%	0%	N/A	100%	100%	N/A
iisenc.zip	2708	13%	0%	100%	92%	91%	100%
iisex.c	2708	17%	0%	100%	100%	100%	100%
iisrules.pl	2708	17%	0%	100%	100%	100%	100%
iisrulessh.pl	2708	17%	0%	100%	100%	100%	100%
iisuni.c	1806	29%	26%	100%	100%	100%	100%
iis_escape_test.sh	2708	17%	0%	100%	100%	100%	100%
iis_nsiislog_post.pm	8035	3%	0%	100%	96%	96%	100%
iis_printer_bof.c	2674	0%	0%	N/A	100%	100%	N/A
iis_source_dumper.pm	1578	71%	70%	100%	91%	91%	100%
iis_w3who_overflow.pm	11820	0%	0%	N/A	100%	100%	N/A
lala.c	2708	17%	0%	100%	43%	31%	100%
linux-wb.c	7116	0%	0%	N/A	89%	89%	N/A
msadc.pl	529	0%	0%	N/A	100%	100%	N/A
rs_iis.c	7116	0%	0%	N/A	74%	74%	N/A
unicodecheck.pl	1806	11%	0%	100%	100%	100%	100%
unicodexecute2.pl	1806	11%	0%	100%	100%	100%	100%
windows_ssl_pct.pm	10116	0%	0%	N/A	100%	100%	N/A
wd.pl	7116	0%	0%	N/A	100%	100%	N/A
Xnuxer.c	7116	0%	0%	N/A	69%	69%	N/A

Table 4.5: Bro Verification Enhancement over Snort

		Snort			Bro		
VEP	BID	A_V	TN_V	TP_V	A_V	TN_V	TP_V
0x82-dcomrpc_usemgret.c	8205	56%	11%	100%	50%	0%	100%
30.07.03.dcom.c	8205	13%	6 %	100%	7%	0%	100%
dcom.c	8205	17%	7%	100%	10%	0%	100%
MS03-039-linux.c	8459	8%	8%	N/A	0%	0%	N/A
oc192-dcom.c	8205	42%	9%	100%	36%	0%	100%
rpc!exec.c	8205	17%	6%	100%	11%	0%	100%

Table 4.6: Snort Verification Enhancement over Bro

We identified that the root causes of the detection and verification problems for Snort and Bro are often different. As explained earlier, the goal of this case study was to show that the VEP testing model can automatically identify detection and verification problems related to the accuracy problem of the IDSs. Although the VEP testing model can be used to compare IDSs, the goal was not to compare Snort and Bro.

As explained earlier, the VEP approach does not use normal traffic to assess IDSs. Thus, the VEP model only assesses the evasion problem (defined in Section 1.1). However, this does not affect our ability to address the verification problem because it is only related to attacks that are already detected (defined in Section 1.1).

In the case of the detection problem, we identify that Snort and Bro have problems because they are missing signatures. Bro also has detection problems not present in the original Snort signatures for mainly two reasons: *s2b* incorporates errors in the Bro signatures and *s2b* does not convert all the Snort plug-ins into the Bro signature language. Moreover, we identified that some Bro signatures cannot be used by the Bro engine. This leads us to propose a new testing model presented in Chapter 5 that specifically assesses the detection problem in IDS engines since we believe that the VEP testing model is not the proper approach to systematically identify these types of detection problems.

In the case of the verification problem, Bro and Snort both have mechanisms in their language to verify the success or failure of an attack. For this case study, Bro has a higher verification accuracy than Snort because *s2b* instruments the Snort signatures to verify attacks. However, we identified different problems in the signatures of Bro that lead to FN_V. Furthermore, most of the FP_V are related to missing signatures in the IDS signature database for verifying attacks. This leads us to the development of a new approach (Chapter 7) that automatically generates IDS signatures that verify attacks.

As a conclusion, our VEP testing model allows to identify key detection and verification problems in IDSs and to evaluate thoroughly the accuracy of existing IDSs such as Snort and Bro, making it amenable to use with any IDS that systematically provides a reference to the vulnerability exploited in their IDS events.

4.4 Analysis of the Limitations of the VEP Testing Model

In this section, we evaluate some of the limitations of the VEP testing model. This analysis aims to assess whether an IDS has a signature for each known vulnerability (e.g., based on available information from SecurityFocus and CVE) and whether there is at least one VEP available for each vulnerability monitored by the IDS signatures. To our knowledge, nobody has measured the limitations of the VEP testing model approach in such a way. This analysis thus enhances the understanding of this approach.

We first use two vulnerability databases (i.e., SecurityFocus and CVE) to evaluate the Snort 2.3.2 and Bro 0.9a9 signature coverage of the currently known vulnerabilities. To evaluate this coverage, we identified the known vulnerabilities at the time of the release of Snort 2.3.2[15] (Bro 0.9a9 used the Snort 2.3.2 signatures) and we compared the results to the vulnerabilities associated with IDS signatures (Section 4.1). Consequently, we can estimate the number of vulnerabilities against which Snort and Bro could identify attacks. Second, we once again use vulnerability databases (i.e., SecurityFocus and CVE) to evaluate the coverage of IDS signatures achievable by a VEP testing model. To evaluate this coverage, we identified the IDS signatures that refer to a vulnerability and we checked for which vulnerability it is possible to have access to a VEP exploiting it. Consequently, we can estimate the number of signatures that can be covered using a VEP testing model.

Table 4.7 describes the results we obtained for this evaluation. As explained in our case study, since Bro uses the Snort signatures and since the signature references to the vulnerability database are maintained when using *s2b*, the results for Snort and Bro are the same. To evaluate the IDS signature coverage of the known vulnerabilities, we first identified that there were around 12 600 known vulnerabilities at the time the Snort 2.3.2 signature database (which contains 3306 signatures) was released. Out of these 3306 signatures, 1562 signatures are related to the identification of attacks that attempt to exploit vulnerabilities (the others could look for policy violation for example). More precisely, these signatures identify attacks that refer to 1064 different vulnerabilities. We can thus

[15] Released in March 2005

estimate that Snort 2.3.2 (and Bro when using the Snort 2.3.2 signatures) could only identify attacks against 8% of the known vulnerabilities at the time.

Second, to evaluate the VEP coverage of the IDS signatures, we identified that there are 222 vulnerabilities (corresponding to 544 signatures) for which it is possible to have access to a VEP.[16] We can thus estimate the maximum coverage of IDS signatures by the VEP testing model to 35% of the signatures that refer to vulnerabilities (i.e., 1562 signatures) and 17% of the signatures in the Snort signature database.

It is possible to develop VEPs for vulnerabilities without a VEP, but this is a time-consuming task and for several of these vulnerabilities not enough information is provided to rapidly develop a VEP. Testing an IDS using VEPs is therefore limited as it is currently not feasible to exercise 100% of an IDS' signatures (i.e., the IDS specification) with a VEP testing model. Nevertheless, we showed in Section 4.3 that our VEP model, using a subset of the available VEPs, was still useful to assess the detection (i.e., more specifically the evasion problem) and the verification problems.

	Nb. of Vulnerabilities	Nb. of Signatures
Release of Snort 2.3.2	$\sim$ 12 600	3306
Coverage of Vulnerability of Snort 2.3.2	1064	1562
VEPs available	222	544

Table 4.7: Evaluation of the VEP Testing Model

The first observation we can make from these results is that there are so many vulnerabilities that an IDS only has signatures that refer to a small subset of them. We believe that the large number of known vulnerabilities prevents the IDS developer from developing all the required IDS signatures. Moreover, there are so many products and versions available (vulnerable or not) that can be installed on a target system (10 000 products are listed on SecurityFocus) that it is almost impossible to test all product versions and use all available VEP configurations against them. Thus, this limitation of any VEP testing model affects its ability to assess the detection and verification problems because it limits the number of attack scenarios that can feasibly be tested and evaluated with an IDS.

[16] We used SecurityFocus, Milworm and Metasploit as our sources of VEPs.

Secondly, we observe that the number of VEPs available (which can be downloaded) for a specific IDS signature database is small compared to the number of vulnerabilities an IDS can monitor. Even if there are at least 222 VEPs (some vulnerabilities have more than one VEP) available to cover 544 signatures of Snort, we also need to consider the time required to understand how these VEPs work and to identify the number of *unusable* VEPs (i.e., we are unable to compile the code and make it launch an attack) since there is no guarantee that VEPs can all be compiled and executed. For instance, Windows XP Professional (without any service pack) has more than 1280 vulnerabilities:[17] VEPs are available for at least 250 of them (as of March 2009). Despite our best effort, we were only able to investigate 108 of these 250 vulnerabilities so far: we generated test cases for 24 of them, 46 cannot actually be used (e.g., some do not compile) and 38 are planned to be used to generate test cases in our future work. Thus, an even smaller number of VEPs than the VEPs available can be easily used with a VEP testing model to test and evaluate IDSs. Nevertheless, our VEP data set contains a greater variety of VEP attacks than what is typically used in the literature, thus reinforcing the idea that our VEP testing model is useful, though with limitations, to assess the detection and verification problems.

4.5 Conclusion

In this chapter, we addressed some of the limitations identified in other VEP testing models. To our knowledge, our VEP data set represents the most complete documented and publicly available data set to test and evaluate IDSs using VEPs. We maximized the diversity of test cases to assess the detection (i.e., more specifically the evasion problem) and the verification problems using a specific testing criterion (i.e., SASC). We developed an approach to reduce the effort required for building test cases (i.e., using the AES/VLab). We used a VEP selection criterion to ensure that our VEP data set represents well what is happening on the Internet in terms of attacks used by average

[17] www.securityfocus.com

attackers.

Moreover, we evaluated the limitations of the VEP testing model approach based on the available VEPs, the known vulnerabilities and the vulnerabilities monitored by IDS signatures. We also proposed a *new methodology* to conduct network security research by proposing an AES/VLab to conduct network experimentations. This AES/VLab was used to address, for the VEP testing model and for other research projects [21, 22], some of the difficulties caused by the dynamic nature of the accuracy problem.

Our VEP testing model allows to identify key detection and verification problems in IDSs and to evaluate thoroughly the accuracy of existing IDSs such as Snort and Bro, making it amenable to use with any IDS that systematically provides a reference to the vulnerability exploited in their IDS events.

Through a case study, our VEP testing model allowed us to identify (1) missing signatures in the IDS database in the case of the detection problem. In the case of the verification problem, the VEP model allowed us to identify two problems: (2) missing signatures that verify the success or failure of an attack and (3) incorrect signatures that led to FN_V. The first problem is currently left for future work while problems (2) and (3) are addressed in Chapter 7 of this thesis. Before we embark on this, we propose in Chapter 5 a second testing model that specifically assesses the detection problem in the IDS engine (i.e., verification of the IDS engine) and we also propose in Chapter 6 a third testing model that specifically assesses some of the detection problems in the IDS signature (i.e., verification of the IDS signature database).

Chapter 5

IDS Engine Stimulator Testing Model

5.1 Introduction

In Chapter 4, we proposed a technique which showed that to accurately assess the detection and verification problems, it is important to precisely understand under which conditions IDSs accurately identify attacks, and under which conditions they fail to do so. With the VEP testing model, we showed, based on quantitative and qualitative data, that the tested IDSs, to various extents, do fail to detect and verify attacks. We identified that errors or missing modules in the IDS engines may prevent an attack from being detected, even when the corresponding signature is correctly specified in the IDS signature database. However, even if the VEP testing model can identify such detection problems, to complement this technique, a software verification technique is required since verification techniques are more adequate to assess detection problems specifically related to the IDS engine (Section 3.1.2).

However, as explained in Section 3.1.2, there is practically no software verification technique

in the literature to verify IDSs. Remember that we defined software verification as the process of evaluating whether the implementation (IDS engine) conforms to its specification (IDS signatures). One can think that software validation techniques such as VEPs and IDS stimulators could be used as verification technique, but we showed in Section 3.1.2 that these validation techniques are not adequate for verification of an IDS engine.

To address all these issues, we propose in this chapter a new IDS verification technique called IDS Engine Stimulator (IDSES) that verifies whether an IDS engine conforms to its specification (signatures) by generating test cases/traffic traces from its signatures. In doing so, our software verification technique does not need to create real attacks as opposed to software validation techniques such as Mucus-1 (Section 3), as long as the signatures used to generate the test cases are the one of the tested IDS engine. For instance, if an IDS engine does not raise an IDS event when executed on a test case derived from one of its signatures, then we can conclude that it does not conform to its specification, regardless of whether this test case is a real attack or not. Moreover, we define test adequacy criteria for such specifications and we use those criteria to build adequate test suites for IDS engine testing, thereby making our approach systematic as opposed to IDS simulators such as Snot, Stick and Mucus-1.

The contribution of this chapter is four-fold:

- We propose a testing model and a specification-based testing approach that clearly maps the IDS engine testing problem to standard software testing principles. When doing so, as opposed to existing work, we systematically decompose the problem and evaluate which mature testing techniques can be used (or adapted) and for what purposes. As a result, the testing model leads to systematic testing strategies for IDS engines.

- We illustrate how this testing model can be applied to an IDS.

- We present a tool that can automate our approach.

- We perform a case study to assess the effectiveness of this testing model on that IDS. Results

show that our approach is effective at systematically revealing numerous problems in this IDS engine (e.g., problems that prevent the detection of attacks).

Section 5.2 provides some background information on a widely used IDS (i.e., Snort) important to the understanding of our IDSES specification-based testing model. Section 5.3 discusses our adequacy criteria for our IDSES specification-based testing model and illustrates its application on this IDS. Section 5.4 discusses a tool that supports our IDSES testing model. Section 5.5 reports on the results of a case study where we used our IDSES testing model on a widely used IDS. Conclusions are drawn in Section 5.6.

5.2 Background on IDSs for Specification-Based Testing

To help understand our IDSES testing model, we selected an IDS to illustrate its applications. We selected Snort for many reasons. First of all, Snort has an open source IDS engine. Thus, having access to the IDS engine source code makes our work easier in a case study to identify the root causes of detection problems. Second, it is the only IDS that offers an open source IDS signature database that is sufficiently large (more than 3000 signatures) to offer a good diversity of signatures. Third, there are various versions of the Snort IDS engine and IDS signature database to conduct comparative studies between IDS versions to see if problems in the IDS engine have been resolved. Fourth, Snort has a history of being used as a template IDS to propose new IDS concepts and ideas in IDS research.

To illustrate our IDSES testing model, we used the Snort 2.4.0 signature database (released in July 2005), as it is currently the last version one can freely download without registration and licensing agreement, and the Snort 2.4.5 IDS engine (released in June 2006), the latest 2.4 engine available at the time this project was started (2006). In the rest of this chapter, Snort 2.4.5 refers to the Snort 2.4.5 engine used with the Snort 2.4.0 signature database. More recent versions of Snort IDS engines and signature databases are available (e.g., the 2.8.5 engine was released in 2010) but

their signatures, though currently very similar to the ones of version 2.4.0, cannot be shown to any unregistered user, which would prevent discussing results in detail in this chapter.

In this section, we discuss the Snort signatures, the plug-ins used to specify its signatures in Section 5.2.2 and its mechanism to monitor sequences of packets in Section 5.2.3.

5.2.1 Snort Signatures

Snort 2.4.5 uses a database of more than 3500 signatures. A Snort signature is composed of plug-ins (Section 5.2.2). Plug-ins evaluate specific characteristics of packet fields and data and return Boolean values. For a Snort signature to raise an IDS event, all its plug-ins should return `true` (i.e., a Snort signature can be viewed as a conjunction of logical tests). For example, in the signature of Figure 5.1, `uricontent` (line 4) is a plug-in that evaluates the content of the URI field of packets, and `uricontent: ".printer"` is a predicate that verifies whether a packet contains string `.printer` in its URI field.

Figure 5.1 shows Snort signature 971 (`sid:971` on line 11) which specifies how Snort detects attacks that attempt to use one of the vulnerabilities of the Microsoft IIS Web Server 5.0, specifically, the `".printer ISAPI Extension Buffer Overflow"` vulnerability. In Microsoft IIS Web Server 5.0, an unchecked buffer in a dll file could allow attackers to execute arbitrary code. This vulnerability only affects version 5.0 of the IIS Web Server. At a high-level, signature 971 specifies that Snort should raise an IDS event for this vulnerability when an attacker sends an HTTP packet that contains `.printer` in the URI field (line 4).

This Snort signature specifies (line 1) that if a packet sent to any identified HTTP server (variable `$HTTP_SERVERS`) on specific HTTP ports (variable `$HTTP_PORTS`) from any listed client (variable `$EXTERNAL_NET`) is part of an open session and contains string `.printer` in its URI field (line 4), then Snort sends the IDS event `WEB-IIS ISAPI .printer access` (line 2) to the network administrator. Line 3 states that the packet must go towards a server in the list of observed servers (other signatures can observe the reaction of the server) and a connection must be established. Line 5

states that the search for string `.printer` is not case sensitive. Lines 6 to 9 provide references to vulnerability databases where vulnerabilities exploited by the attack at stake are precisely defined. This information is not necessary for intrusion detection but it is used when reporting the IDS event to the network administrator. Line 12 shows the revision number of the signature.

```
1   alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (
2   msg:"WEB-IIS ISAPI .printer access";
3   flow:to_server,established;
4   uricontent:".printer";
5   nocase;
6   reference:arachnids,533;
7   reference:bugtraq,2674;
8   reference:cve,20010241;
9   reference:nessus,10661;
10  classtype:webapplicationactivity;
11  sid:971;
12  rev:9;
13  )
```

Figure 5.1: Snort Signature 971

5.2.2 Snort Plug-ins

We already discussed some plug-ins (e.g., `uricontent`). Other plug-ins allow the manipulation of regular expressions (`pcre`) to be used when one wants to specify the content of a payload as a regular expression instead of a (sub)string, or the manipulation (e.g., bytewise AND) and evaluation (e.g., equality) of byte values (`byte_test`). There is a total of 52 plug-ins in Snort 2.4.5, classified into five categories [18]: 6 header plug-ins, 5 post-detection plug-ins, 20 non-payload plug-ins, 15 payload plug-ins, and 6 meta-data plug-ins. Header plug-ins define the protocol (IP, TCP, UDP and ICMP), the IP source and destination addresses, the direction (source to destination or destination to source) and the transport protocol source and destination ports used by an attack packet (e.g., line 1 in Figure 5.1). Meta-data plug-ins describe the message provided to the network administrator when a signature is triggered (e.g., `msg`), provide an identifier to the signature (e.g., `sid` and `rev`) and link signatures to (vulnerability) databases (e.g., `reference`). Payload plug-ins describe the packet

payload (e.g., `content`, `pcre`).[1] Non-payload plug-ins describe the protocol fields in the packet (e.g., `ip_opts` checks the IP options of an IP packet). Post-detection plug-ins describe what needs to be done when a packet satisfies the signature (e.g., raise an IDS event, block a TCP connection).

5.2.3 Snort `flowbits`

In this section, we illustrate the `flowbits` plug-in used by Snort to monitor sequences of packets. We use the two complex Snort signatures (excerpts), shown in Figure 5.2. Signature 2192 is about a packet being sent to a server (the `to_server` flow on line 3) whereas signature 2350 is about a packet being sent by the server (the `from_server` flow on line 3). These signatures illustrate that the description of the contents of a packet payload can be complex, using plug-ins such as `content`, `within` and `distance` (among others). For example, signature 2192 specifies that, when looking at any byte (say the i^{th}) of the TCP packet payload, the $(i+1)^{th}$ byte should be hexadecimal value `05` (lines 4 and 5) and the $(i+3)^{th}$ byte should be hexadecimal value `0B` (`distance:1` makes the IDS skip the $(i+2)^{th}$ byte).

These signatures also illustrate the `flowbits` plug-in of Snort that manipulates Boolean variables to make signatures raise IDS events only when other signatures have previously been triggered. This mechanism is useful as some attacks involve several packets and the identification of only one of those packets should not necessary lead to an IDS event; whereas identifying a sequence of specific packets does (i.e., triggering specific signatures in a specific order). For instance, signature 2192 in Figure 5.2 sets Boolean variable `dce.isystemactivator.bind.attempt` to 1 (line 10) and remains silent (line 11) with two different uses of the `flowbits` plug-in. This signature specifies that an attempt to bind to a `dce` service has been made. Signature 2350 shows three uses of the `flowbits` plug-in. It requires that Boolean variable `dce.isystemactivator.bind.attempt` be set to 1 (line 7), i.e., this is a predicate that must be true for the signature to be triggered; it also sets Boolean variable `dce.isystemactivator.bind` to 1 (line 8); finally it remains silent (line 9). This signature specifies

[1] *The actual data that the packet is delivering to the destination.*

```
1   alert tcp $EXTERNAL_NET any -> $HOME_NET 135 (
2   msg:"NETBIOS DCERPC ISystemActivator bind attempt";
3   flow:to_server,established;
4   content:"|05|";
5   within:1;
6   content:" |0B|";
7   within:1;
8   distance:1;
9   ...
10  flowbits:set,dce.isystemactivator.bind.attempt;
11  flowbits:noalert;
12  reference:bugtraq,8205;
13  reference:...
14  classtype:protocol-command-decode;
15  sid:2192;
16  rev:10;
17  )
```

```
1   alert tcp $HOME_NET 135 -> $EXTERNAL_NET any (
2   msg:"NETBIOS DCERPC ISystemActivator bind accept";
3   flow:from_server,established;
4   content:" |05|";
5   within:1;
6   ...
7   flowbits:isset,dce.isystemactivator.bind.attempt;
8   flowbits:set,dce.isystemactivator.bind;
9   flowbits:noalert;
10  reference:bugtraq,8205;
11  reference:...
12  classtype:protocol-command-decode;
13  sid:2350;
14  rev:9;
15  )
```

Figure 5.2: Snort Signature 2192 and 2350 (excerpts)

that the attempt has succeeded. These two signatures, observing packets that indicate an attempt to bind to a service and the success of this attempt, are not sufficient to raise an IDS event: the signatures therefore indicates that the IDS should remain silent. However, these two events can be the beginning of an attack, and must therefore be monitored. A third signature (not shown here) expects `dce.isystemactivator.bind` to be equal to 1 and monitor the contents of a third packet to raise an alarm. As illustrated, the essence of the plug-in `flowbits` is to specify state based behaviour.

5.3 A Specification-Based Testing Approach for Testing IDS Engines

IDSs monitor sequences and contents of packets (e.g., source and destination IP addresses, source and destination TCP ports, packet payload) to identify attacks. Thus, IDSs rely on specific characteristics of packets and protocol headers to define intrusion detection signatures. Even if several paradigms are used as IDS signature languages (i.e., Temporal Logic [45], Petri Nets [46], State Machines [6, 47], Event Calculus [48], Regular Expressions [49] and Ad hoc paradigms such as Snort [18] and Bro [19]), any IDS signature language must allow the IDS engine to check the (Boolean) characteristics of packets (e.g. values of protocol fields, strings and bytes in the packet payload), and to check packets individually as well as in sequences (notion of state). Thus, we can reasonably assume, without loss of generality, that IDSs have a signature language that can be easily abstracted into a state machine where the guard conditions of events triggering state changes are logical expressions made of predicates, and that predicates use boolean functions (i.e., predicate functions) when checking packet characteristics. This is precisely the approach followed in [47]. We can further assume that those logical expressions are conjunctions of predicates, checking packet characteristics. If a guard is not a conjunction of predicates, we can always transform it in a disjunctive normal form (DNF) and replace the corresponding transition with as many transitions as disjuncts in the DNF.

We propose an IDSES specification-based testing model, based on this observation and that an IDS specification (i.e., IDS signature database) can be used with test criteria, test strategy and an IDS Stimulator to test its IDS implementation (i.e., IDS engine). The mechanisms used to specify IDS signatures lead us to identify three different levels for testing an IDS engine: (1) we can test the predicates individually, which we refer to as the *predicate level testing* (Section 5.3.1); (2) we can test logical expressions (transitions) individually, which we refer to as the *logic level testing* (Section 5.3.2); (3) we can test sequences of transitions (including guard conditions), which we refer to as the *state machine level testing* (Section 5.3.3). These three testing levels allow to decompose

the IDS engine testing problem into precise sub-problems for which test strategies can be defined. For instance, before verifying that predicates are correctly evaluated within a transition, we suggest to test those predicates individually (predicate vs. logic testing level). This is very similar to the decomposition of software testing activities into unit testing and integration testing [52]. Thus, the testing levels and associated criteria we discuss in this section focus on the parts of state machine that specify what a packet should look like for the IDS to raise an IDS event.

Note that the contribution presented in this section, i.e., our IDSES specification-based testing model, is mainly theoretical since the implementation of all three testing level is quite labour intensive. Nevertheless, we demonstrate in this section how the three testing levels can be applied to a widely used IDS, we implement the logic testing level (Section 5.3.2), we provide a justification for choosing this testing level (Section 5.4) and we conduct a case study for this testing level (Section 5.5).

5.3.1 Predicate Level Testing and Criteria

It is important to systematically and individually test the predicates because at the other two levels, only parts of the predicates' behaviour can be tested, (i.e., only the behaviour actually used in the transitions and transition sequences).

At the predicate level, two questions have to be answered: How to unit-test predicates (Section 5.3.1.3) and how to test the interaction of predicates (Section 5.3.1.4). Before answering these two questions, we have to define what is a predicate for the IDS under test (Section 5.3.1.1) and we need to discuss how some language specific constraints impose test conditions on the predicates that have to be tested (Section 5.3.1.2).

5.3.1.1 Predicate Definition

The IDS signature language is specific to the IDS using it. Thus, some particularity of the signature language can make it difficult to identify what are precisely the predicates. The first thing to do at

the predicate testing level is to define the concept of predicate for the tested IDS engine, based on its signature language.

In the case of Snort, the predicates are the plug-ins because they are Boolean expressions that evaluate specific characteristics of packet fields and data (Section 5.2.2). However, there are two exceptions to this statement.

First, in Snort, a predicate can also be composed of many plug-ins since the Snort signature language imposes that some plug-ins must be used together with other plug-ins. If they are used alone, this leads to an error in the signature. These plug-ins are `nocase`, `rawbytes`, `depth`, `offset`, `within` and `distance`. They must be used with either `uricontent` or `content`, and therefore cannot be unit tested. They can only be tested with `uricontent` and `content`. Thus, they are considered as parameters of `uricontent` and `content` that change the way `uricontent` and `content` behave. As a consequence, the occurrence of `uricontent` (or `content`) with any of those plug-ins is considered as one single predicate.

Second, some plug-ins are not predicates. These plug-ins do not evaluate to `true` or `false`. In particular, we do not consider the Meta-data plug-ins (Section 5.2.2) as predicates since they only add context to the signatures. This is because, in the context of IDS engine testing, we are not interested in testing whether the right references (`reference`) to vulnerability databases are made or whether the signature is given the right attack class (`classtype`), since the IDS engine does not evaluate this information on packets.

5.3.1.2 Test Conditions

Testing predicates is also subject to constraints of the IDS signature language. The main constraint is that predicates cannot be understood by the IDS outside of a signature. Thus, testing at the predicate level must therefore entail (1) the construction of specific signatures where the predicates are under test and (2) the construction of test cases (i.e., packets) for those signatures.

Building a signature for the test of a predicate must follow the IDS signature language con-

straints. Testing that the IDS engine recognizes syntax problems in signatures does not require the construction of packets, and is outside the scope of this thesis.

In the case of Snort, the main condition imposed by its signature language is that all the header plug-ins must be part of a signature (e.g., line 1 in Figure 5.1). The Snort developers made this decision to filter as much as possible the packets to be analyzed in a signature, by selecting them based on their source, destination IP addresses and ports.

5.3.1.3 Unit Testing

Each predicate provides specific functionalities and has to be (unit) tested with specific techniques. The testing techniques are specific to what constitutes a predicate for the signature language of the IDS engine under test.

In the case of Snort, Table 5.1 presents the different testing techniques we suggest to unit test the predicates (i.e., plug-ins). Table 5.1 is an excerpt of a more complete table were all the Snort plug-ins are listed and can be found in [23].

Snort plug-in	Functionality	Testing Technique
`uricontent`	Search for a string in the URI field	Category Partition; [53]
`content`	Search for a string in the packet payload	Category Partition [53]
`flow`	Analyze the state of TCP connection	Testing finite state machines [54]
`pcre`	Search in the packet payload using regular expressions	Testing finite state machines [54]
`byte_test`	Operation on bytes	Category Partition [53]

Table 5.1: Testing Techniques for Snort Plug-ins

For instance, `uricontent` (with arguments) specifies the content of the URI field of a packet, and can be tested with the Category Partition [53] testing technique. The corresponding test cases could exercise distinct situations and some interesting properties of the predicate such as when the expected content is not in the URI field, when it appears one or several times, or when it appears at the beginning, end or in the middle of the URI field. In the case of the `pcre` plug-in, which takes a regular expression as an argument, since a regular expression can be represented as a finite state machine, testing techniques devised for finite state machines are probably more appropriate [54].

5.3.1.4 Interaction Testing

As explained in Section 5.3.1.2, constraints in the signature language impose that some predicates have to be used with others. Moreover, it is also possible that some predicates *influence* the behaviour of others (i.e., the predicates are not *independent*). Thus, we have to define the *influence* relationship between predicates to develop an interaction testing strategy at the predicate testing level.

Assuming P is a set of predicates, we define relation $\rightarrow$ (*influences*) as follows: $p_1, p_2 \in P$, $p_1 \rightarrow p_2$ iff the presence of p_1 may modify the behaviour of p_2 in signatures. And we say that p_1 is *independent* of (i.e., can never *influence*) p_2 iff $\neg(p_1 \rightarrow p_2)$ and $\neg(p_2 \rightarrow p_1)$.

For instance, in the case of Snort, we have `content` $\rightarrow$ `pcre` because `content` modifies the way `pcre` behaves when a specific parameter of `pcre` is used (i.e., `R`, which stands for relative): specifically, the parameter changes the location in the packet payload where the `pcre` plug-in should look for a regular expression, based on what string is used by the `content` plug-in. For example, `content:"string"; pcre:"/t/"` specifies that `string` should appear in the payload and that regular expression `t` should also appear in the payload (e.g., matching the string `string`), whereas `content:"string"; pcre:"/t/R"` specifies that `t` should appear in the payload after the word `string` (i.e., specified parameter `R`). In this case, the regular expression matches the remaining of the payload after what has been found by `content` (e.g., matching the string `stringt`).

The $\rightarrow$ relation among the predicates for Snort is depicted in Figure 5.3: Arrows illustrate relation $\rightarrow$; An arrow to/from a (rounded) rectangle illustrates relation $\rightarrow$ to/from all the elements in the (rounded) rectangle. Only the Payload plug-ins are presented in this figure because the Non-Payload and Header plug-ins are all *independent* from other plug-ins. Plug-ins `content`, `uricontent` (rectangle) and `pcre`, `byte_test`, `byte_jump`, `isdataat` (rounded rectangle) are grouped as they influence one another within the same group.

Predicates have to respect constraints of the language and that the *influence* ($\rightarrow$ relation) between predicates has to be considered at this testing level.

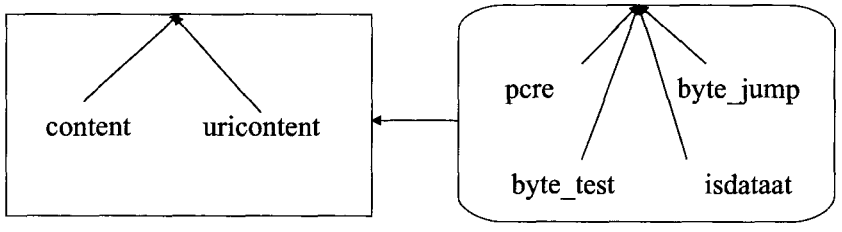

Figure 5.3: Relations between Snort Plug-ins

In the case of Snort, based on its predicate relations presented in Figure 5.3 and on the test conditions identified in Section 5.3.1.2, here is what can be suggested as a possible test order.

Header plug-ins have to be tested first because they are mandatory in signatures, then Payload and Non-Payload plug-ins can be tested independently. Since Non-Payload plug-ins do not influence the behaviour of one another, they can be tested independently. The rationale is different for Payload plug-ins. The plug-ins grouped in the non-rounded rectangle in Figure 5.3 can first be tested independently, and then together. Plug-ins in the rounded rectangle can be tested independently, then together and finally with `content` and `uricontent`. Plug-ins `ftpbounce` and `asn1`, not shown in Figure 5.3 because they are not involved in any relation, can be tested independently of the others.

These dependencies, i.e., the fact that the presence of a plug-in can modify the behaviour of another plug-in, also suggest that interaction testing strategies such as the ones developed by the software testing community can be tailored to our context (e.g., [55]).

5.3.1.5 Discussion

We recognize that most available IDSs rely on specific characteristics of packets and protocol headers to define intrusion detection signatures and that at the predicate level, high-level concepts such as the predicates (Section 5.3.1.1), the constraints imposed by the language (Section 5.3.1.2) and the predicate interaction relationships (Section 5.3.1.4) for the signature language of the tested IDS have

to be identified. Specifically, the predicates for each signature language provide distinct, specific functionalities that have to be (unit) tested using distinct techniques (Section 5.3.1.3). For instance, in the Bro signature language, one predicate specifies the content of a packet payload (using regular expression). In the case of Snort, there are different predicates that provide different functionalities (i.e., search the packet payload using string, regular expression or bytes) to specify the content of a packet payload. These Snort predicates require different testing techniques (see Table 5.1). We believe that at the predicate testing level, identifying a testing technique for a predicate can only be done on a case by case basis that highly depends on the signature language. Thus, we decided to leave the experimentation of the assessment (case study) of the detection problem at the predicate level outside the scope of this thesis. This would require the application of testing techniques specific to the paradigm (e.g., string, byte, regular expression) used by the predicates of the IDS signature language that would be closer to an engineering problem (only specific to the tested IDS) than to a research problem.

5.3.2 Logic Level Testing and Criteria

At the logic level, we test predicates in the state machine representation, i.e., guards that are logical expressions we assumed to be conjuncts. We use the predicate definition to derive the logical expressions (Section 5.3.2.1) and we identify the test criteria (Section 5.3.2.2). Then, we select the relevant test criteria (Section 5.3.2.3) and we identify the test requirements for each criterion (Section 5.3.2.4).

5.3.2.1 Logical Expression

To identify the logical expressions, we use the predicate definition proposed in Section 5.3.1. For instance, in the case of Snort, signature 971 (Figure 5.1) is a conjunction of 8 predicates A∧B∧C∧D∧E∧F∧G∧H (Table 5.2). Note that we grouped `uricontent` and `nocase` in predicate `H` as per our previous discussion (Section 5.3.1). It is important to note that plug-ins `reference`, `classtype`, `sid` and `rev`

are not considered predicates since they do not evaluate to true or false (Section 5.3.1).

```
A=tcp
B=$EXTERNAL_NET
C=any
D=->
E=$HTTP_SERVERS
F=$HTTP_PORTS
G=flow:to_server,established
H=uricontent:".printer";nocase
```

Table 5.2: Predicates for Snort Signature 971

5.3.2.2 Test Criteria

Several criteria have already been defined for testing Boolean expressions [56, 50]. For instance, the simple *All True Predicate Coverage* criterion requires that a combination of Boolean variable values that lead to a true value of the whole Boolean expression be exercised.

In [50], the authors describe test criteria for testing logical expressions. They define eight coverage criteria to which we added the All True Predicate Coverage. Before providing the definition of each coverage criterion, we have to first define what is a predicate, a clause, a logical formula, a major and minor predicate, the determination concept, and a test requirement. Moreover, the authors of [50] seem to use slightly different definitions than the common definitions for predicates and clauses used in first-order logic. Thus, we changed the definition of the nine criteria according to the most common terminology. The authors of [50] define a predicate as an expression that evaluates to a Boolean value, and that is the topmost structure. Commonly, in first-order logic, this concept rather corresponds to the definition of a formula. In first-order logic, a predicate is an atomic formula (also known simply as an atom) because it does not contain any of the logical operators (e.g., and, or). In [50], the authors refer to this concept as the definition of a clause. However, in first-order logic, a clause is defined as a disjunction of predicates. To standardize the definition of the nine coverage criteria, we use the term formula, predicate and clause as is common in first-order logic. First, we need to provide some concepts. Logical expressions (i.e., logical formula) and their predicates are

used to introduce these covering criteria. We define F as a set of logical formulas and P as a set of predicates in the logical formulas in F. For each logical formula $f \in F$, P_f is the predicates in f, that is, $P_f = \{p \mid p \in f\}$. P is the union of each P_f in each logical formula in F.

In some logical expressions (e.g., $(A \vee true) \wedge B$), the value of some predicates (e.g., A) does not change the value of the logical expression. To really test the importance of a predicate, the predicate should *determine* the value of the logical expression. Thus, we introduce the concept of determination, major predicate and minor predicate.

Definition 3 *A predicate p_i in P_f, called the major predicate, determines f if and only if the values of the remaining minor predicate p_j are such that changing p_i changes the value of f.*

The last concepts to define are a test requirement and a test set. The authors of [50] define a test requirement (tr) as a specific element of a software artifact that a test case must satisfy or cover. We denote TR the set of test requirements. The authors of [50] define a test set T a set of test cases.

Here is the description of nine test criteria (the name in parentheses is the original criterion name from [50]):

1. Formula Coverage (Predicate Coverage): For each $f \in F$, TR contains two requirements: f evaluates to true, and f evaluates to false.

2. Predicate Coverage (Clause Coverage): For each $p \in P$, TR contains two requirements: p evaluates to true, and p evaluates to false.

3. Combinatorial Coverage (Combinatorial Coverage): For each $f \in F$, TR has test requirements for each predicate in P_f to evaluate to each possible combination of truth values.

4. General Active Predicate Coverage (General Active Clause Coverage): For each $f \in F$ and each major predicate $p_i \in P_f$, choose minor predicates p_j, j different from i so that p_i determines f. TR has two requirements for each p_i: p_i evaluates to true and p_i evaluates to false. The

values chosen for the minor predicates p_j do not need to be the same when p_i is true as when p_i is false.

5. Correlated Active Predicate Coverage (Correlated Active Clause Coverage): For each $f \in F$ and each major predicate $p_i \in P_f$, choose minor predicates p_j, j different from i so that p_i determines f. TR has two requirements for each p_i: p_i evaluates to true and p_i evaluates to false. The values chosen for the minor predicates p_j must cause f to be true for one value of the major predicate p_i and false for the other.

6. Restricted Active Predicate Coverage (Restricted Active Clause Coverage): For each $f \in F$ and each major predicate $p_i \in P_f$, choose minor predicates p_j, j different from i so that p_i determines f. TR has two requirements for each p_i: p_i evaluates to true and p_i evaluates to false. The values chosen for the minor predicates p_j must be the same when p_i is true as when p_i is false.

7. General Inactive Predicate Coverage (General Inactive Clause Coverage): For each $f \in F$ and each major predicate $p_i \in P_f$, choose minor predicates p_j, j different from i so that p_i does not determine f. TR has four requirements for p_i under these circumstances: (1) p_i evaluates to true with f true, (2) p_i evaluates to false with f true, (3) p_i evaluates to true with f false, and (4) p_i evaluates to false with f false. The values chosen for the minor predicates p_j may vary among the four cases.

8. Restricted Inactive Predicate Coverage (Restricted Inactive Clause Coverage): For each $f \in F$ and each major predicate $p_i \in P_f$, choose minor predicates p_j, j different from i so that p_i does not determine f. TR has four requirements for p_i under these circumstances: (1) p_i evaluates to true with f true, (2) p_i evaluates to false with f true, (3) p_i evaluates to true with f false, and (4) p_i evaluates to false with f false. The values chosen for the minor predicates p_j must be the same in cases (1) and (2), and the values chosen for the minor predicate p_j must also be the same in cases (3) and (4).

9. All True Predicate Coverage: For each $p \in P$, TR contains one requirement: p evaluates to true.

5.3.2.3 Selection of Test Criteria

Choosing a test criterion is often dependent on (1) the cost of using it, (2) its relevance to the logical expressions under test (the IDS signature language) and (3) the relationships between the chosen test criteria.

First, test criterion 3 is very costly (i.e., time-consuming) because it requires to evaluate each predicate with respect to each possible combination of truth values. Thus, this test criterion should only be used in last resort or when its cost is justified (i.e., highly critical and only a few predicates in the logical formula). Consequently, we do not use test criterion 3 for IDS engine testing.

Second, not all test criteria are relevant in the context of an IDS signature language. For instance, the guard conditions derived from the state machines are conjunctions of predicates. Thus, the inactive test criteria 7 and 8 are not relevant because in a conjunction, all the predicates are active and there is no inactive predicate. Moreover, in the case of a conjunction, we can show that criteria 4, 5 or 6 are *equivalent*. To prove this, we need to introduce the concept of *subsumption* and *equivalence*.

Definition 4 *Test criterion i subsumes test criterion j if and only if every test set T that satisfies test criterion i also satisfies test criterion j.*

Definition 5 *Test criterion i is equivalent to test criterion j if and only if i subsumes j and j subsumes i.*

From [50], we know that criterion 6 subsumes criterion 5 and that criterion 5 subsumes criterion 4. Thus, by demonstrating that criterion 4 subsumes in the case of a conjunction criterion 6, we demonstrate that criteria 4, 5 and 6 are equivalent in the case of a conjunction.

Note that the only difference between criteria 4, 5 and 6 is the choice of the value associated with

the minor predicate (Section 5.3.2.2). Thus, to demonstrate that criterion 4 subsumes criterion 6 in the case of a conjunction, it is sufficient to show that in the case of a conjunction, there is only one possible TR (i.e., only one possible choice for each minor predicate) that satisfies criterion 4 and that this TR also satisfies criteria 6. Here is this demonstration.

Based on criterion 4 (and also criteria 5 and 6) there are two test requirements for each major predicate p_i: p_i has to evaluate to true and p_i has to evaluate to false. By definition, a major predicate such as p_i has to determine the value of f. In the case of a conjunction, for p_i to determine f (whether p_i is true or false), all the other predicates have to be true. Otherwise, this violates the definition of a major predicate. Consequently, the test requirement where p_i = true is the same for all p_i in P_f (i.e., p_1 = true, ..., p_i = true, ..., p_n = true) and there is only one test requirement possible for each p_i in P_f when p_i = false (i.e., p_1 = true, ..., p_i = false, ..., p_n = true). Consequently, in the case of a conjunction, there is only one possible set TR that satisfies criterion 4 which contains $n + 1$ specific test requirements (where n is the number of predicates in P_f). Moreover, this TR also satisfies criterion 6 since the values chosen for the minor predicates p_j are the same when p_i is true and when p_j is false (Section 5.3.2.2). Thus, in the case of a conjunction, criterion 4 subsumes criterion 6. The latter implies that criteria 4, 5 and 6 are equivalent in the case of a conjunction. Consequently, only test criteria 1, 2, 6 (same as 4 and 5) and 9 are relevant to use at the logic level.

Third, the subsumption relationships between test criteria [50] can also be used to identify the proper test criteria to choose. From [50], we also know that test criterion 6 subsumes 1, 2 and 9. Thus, by satisfying test criterion 6, the remaining test criteria 1, 2 and 9 are satisfied. However, in the context of this thesis, we use all the test criteria (i.e., criteria 1, 2, 6 and 9) relevant to an IDS signature language to compare their cost and effectiveness.

5.3.2.4 Test Requirement

In this section, we illustrate how we can use a test criterion to derive a set of test requirements TR from logical expressions (i.e. IDS signatures). We also show that for some IDS signature languages,

some test requirements of TR are impossible.

For instance, we use Snort signature 971 presented in Figure 5.1 and its predicates presented in Table 5.2. As mentioned earlier, we selected the Formula Coverage (FC) criterion (i.e., criterion 1), the Predicate Coverage (PC) (i.e., criterion 2), the Restricted Active Predicate Coverage (RAPC) criterion (i.e., criterion 6) and the All True Predicate Coverage (ATPC) criterion (i.e., criterion 9).

Predicate	tr_1	tr_2	tr_3	tr_4	tr_5	tr_6	tr_7	tr_8	tr_9	tr_{10}
A=tcp	T	*F*	*F*	T	T	T	T	T	T	T
B=$EXTERNAL_NET	T	*F*	T	*F*	T	T	T	T	T	T
C=any	T	*F*	T	T	*F*	T	T	T	T	T
D=->	T	F	T	T	T	F	T	T	T	T
E=$HTTP_SERVERS	T	*F*	T	T	T	T	*F*	T	T	T
F=$HTTP_PORTS	T	F	T	T	T	T	T	F	T	T
G=flow:to_server,established	T	F	T	T	T	T	T	T	F	T
H=uricontent:".printer";nocase	T	F	T	T	T	T	T	T	T	F

Table 5.3: Test Cases Snort Signature 971

Table 5.3 lists 10 test requirements (where T=true and F=false) that can be derived from the logical expression A∧B∧C∧D∧E∧F∧G∧H, derived itself from Snort signature 971 in Table 5.2.

Different sets of test requirements (TR) satisfy the various test coverage criteria. For instance, $TR_{ATPC} = \{tr_1\}$ satisfies the ATPC criterion. $TR_{FC/PC} = \{tr_1, tr_2\}$ is the smallest possible set of test requirements (in the case of a conjunction) that satisfies both the Formula Coverage (FC) and Predicate Coverage (PC) criteria. $TR_{RAPC} = \{tr_1, tr_3, tr_4, tr_5, tr_6, tr_7, tr_8, tr_9, tr_{10}\}$ is the only set of test requirements that satisfies RAPC.

For some test requirements, specific predicates cannot be false (i.e., designed by F instead of F in Table 5.3) and there are three reasons for that.

First, in test requirements tr_2 and tr_3, it is not simple to find the meaning of the false value for predicate A (i.e., A = tcp). What does it mean for a packet to not be a TCP packet? One can suggest replacing TCP by another protocol when A is false. If we decide to replace it with ICMP, the source and destination port predicates do not make sense since ICMP does not have a notion of port. However, if we use UDP, the source and destination port predicates still make sense. Identifying a negative value for a protocol is not a simple problem and needs more thought. For this

reason, we leave this problem for our future work and we consider that the `protocol` predicate in Snort cannot be false.

Second, in some cases, such as predicate C, it is impossible for the predicate to be false. For example, there is no value for the opposite (e.g., the false value) of all possible source ports. Thus, predicate C cannot be false. The predicate C does not add any constraint to the signature, but it has to be part of the signature because the Snort signature language requires the value for predicates A to F (the six header plug-ins) to be explicitly specified in every Snort signature (Section 5.2.2 and 5.3.1.2).

The same situation applies for predicates B and E. Recall that IDSs have a *Configuration* component that stores the IDS configuration and that specifies how the IDS Engine has to behave when there are attacks. In the case of predicates B and E, the default Snort configuration (i.e., the configuration that is installed by default with Snort) specifies that the value of the variables `$EXTERNAL_NET` and `$HTTP_SERVERS` is once again `any`. Thus, `$EXTERNAL_NET` and `$HTTP_SERVERS` represents respectively all the possible source (for predicate B) and destination (for predicate E) IP addresses. Since we used in this work the default configuration of Snort, predicates B and E cannot be false. Predicate F also depends on the default configuration of Snort, but the default value of `$HTTP_PORTS` is 80. Thus, predicate F can be false.

Third, a predicate cannot be false when it is implied by another predicate and when this other predicate has to be true for a specific test case. In the case of Snort, this situation can only happen with payload plug-ins because all the other plug-ins verify disjoint bytes (protocol field) of the packet. It is possible that the bytes one plug-in is verifying in the packet payload are also verified by another payload plug-in. Assume a Snort signature that contains `content:"bla"; pcre:/bla[^\n]{432}/i`, where the second predicate checks whether the packet payload contains the string `blab` (`i` is for case insensitive) followed by 432 characters that are not a carriage return (`\n`). This `content` predicate is implied (i.e., includes) the `pcre` predicate. A test case that requires the `content` predicate to be false and the `pcre` predicate to be true is impossible. This situation

often occurs in Snort signatures for performance reasons since the Snort engine is designed to perform better when a `pcre` is used together with (is included in) a `content` plug-in than used alone in a signature.[2]

Consequently, for the $TR_{FC/PC}$, tr_2 has to be modified to meet the above constraints (i.e., would have to be TTTFTFFF instead of FFFFFFFF). In the case of TR_{RAPC}, tr_3, tr_4, tr_5 and tr_7 are impossible. Thus, $TR_{RAPC} = \{tr_1, tr_6, tr_8, tr_9, tr_{10}\}$.

5.3.3 State Machine Level Testing and Criteria

The state machine testing level targets how the IDS monitors sequences of actions (e.g., packets) taken by attackers. First, we need to identify within the IDS signature database the different state machines (i.e., in the case of Snort the groups of signatures that interact with one another to monitor sequences of actions). Then, we need to identify and use test criteria on these state machines.

5.3.3.1 State Machine

As explained earlier, this state-based behaviour is used in many IDS signature languages: Temporal Logic [45], Petri Nets [46], State Machines [6, 47], Event Calculus [48], Regular Expressions [49] and Ad hoc paradigms such as Bro and Snort.

To identify the state machines within the Snort signature database, we are interested in signatures that use the `flowbits` plug-in. For example, from the two Snort signatures presented in Figure 5.2, we can derive part of the state machine presented in Figure 5.4.

In fact, there are other signatures, not shown in Figure 5.4 due to space constraints, that require the variable `dce.isystemactivator.bind` to be set to 1 to raise an IDS event. As one can see, the `flowbits` plug-in, in effect, specifies a state-based behaviour, as summarized in Figure 5.4 (excerpt), where `var1` and `var2` replace respectively the variables `dce.isystemactivator.bind.attempt` and `dce.isystemactivator.bind`, to improve legibility. Signatures 2351, 2352, 3198, and 3197 (not

[2] `vrt-sourcefire.blogspot.com/2009/07/rule-performance-part-one-content.html`

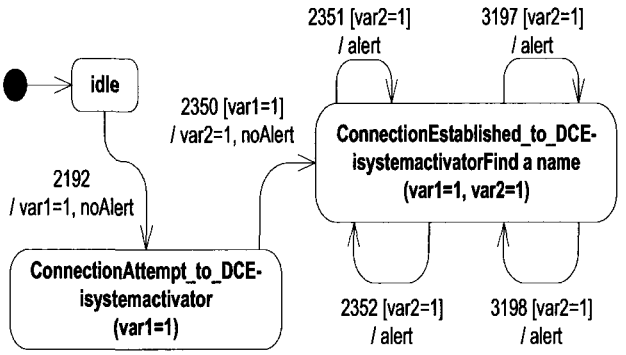

Figure 5.4: Sequence of signatures as a state based behaviour

shown in this chapter) require `var2` to be set to 1 (and therefore that `var1` be set to 1 too) in order to raise an IDS event. In other words, they require signatures 2192 and 2350 to be triggered beforehand, and in that order.

5.3.3.2 Test Criteria

When all the state machines have been identified from the IDS signature database, testing criteria for state machines can be applied [57]. Similarly to the logic level, different test criteria can be considered. This includes test criteria that define test requirements from the behaviour explicitly specified in the state machine (e.g., covering states or transitions), as well as test requirements that ensure that when no transition or action is specified, this is actually the case (i.e., sneak paths [57]). Once again, as with the logic level, a set of test requirements is identified (i.e., a set of sequences of transitions to be triggered) and the test cases can be generated.

The selection and application of these state machine test criteria are currently part of our future work.

5.4 IDS Engine Stimulator Testing Model

In this section, we present how we implement our IDSES testing model using the specification-based approach presented in Section 5.3.

5.4.1 Implementation

We chose to first implement the logic testing level as it shows whether or not an IDS engine is actually able to at least properly use its signature database. This is a crucial issue when addressing the detection problem in IDS engines and should be the minimum expected from any IDS. The logic testing level is the first logical choice for implementation since it tests a specific usage of the predicates (i.e., predicate level) by generating test cases for their current usage and for their most important usage (i.e., in the signatures contained in the IDS signature database). The logic testing level also shares commons concepts with the state machine testing level (such as logical expressions that specify transition conditions between the sates of the state machine) making it a useful first step for implementation.

Thus, the framework that implements our IDSES testing model was developed with two requirements in mind: (1) to potentially be able to handle the three levels of testing discussed previously, (2) to support the implementation of the logic level by automating the generation of test cases using an IDS signature database for the logical expression coverage criteria presented in Section 5.3.2.2.

The IDSES uses a three-phased approach (Figure 5.5) to generate test cases. First, the IDS signatures (step 1) are analyzed and transformed by the *Parser* module into the models used for each testing level (e.g. logical expressions for the logic level, see Section 5.3.2.1).

Second, these models (step 2) are fed to a *Test Case Generator* module to generate the test cases (i.e., packets) according to selected testing criteria (Section 5.3.2.3). For example, an *ATPC Test Case Generator* is dedicated to the ATPC criterion at the logic level. Each *Test Case Generator* derives their corresponding test requirements and removes the impossible test requirements

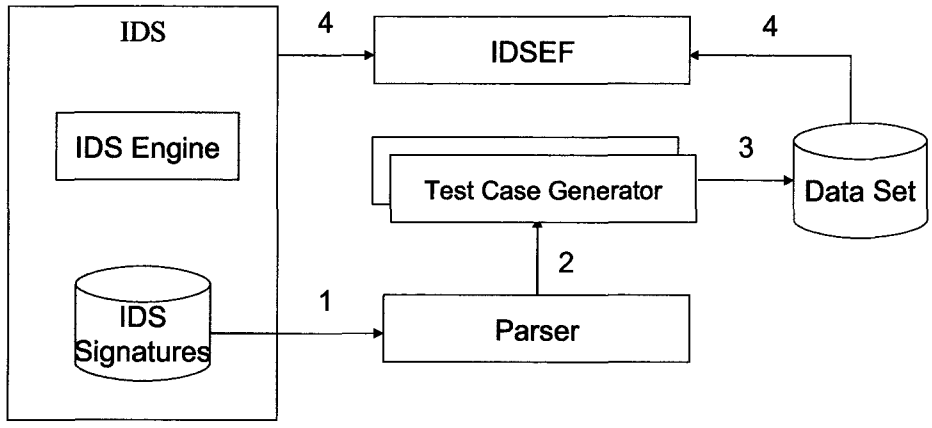

Figure 5.5: IDS Engine Stimulator

(Section 5.3.2.4).

Third, each *Test Case Generator* (step 3) uses the relevant test requirements to generate the packets (i.e., test cases), using *Libnet*,[3] which is commonly used to generates packets. Then it stores these packets into a *tcpdump* file format and documents them.

The IDSEF (step 4) is then used with the generated test cases (and the same IDS used to generate these test cases) to assess the detection problem of its corresponding IDS engine.

5.4.2 Documentation Process

To facilitate future reuse, sharing and the automatic evaluation of detection accuracy with the IDSEF, each test case in the data set is documented: *snort signature used, predicate / signature / state machine testing level considered, testing criterion applied* and the *actual value*. Figure 5.6 presents an example of such documentation.

The *actual value* specifies whether the test case represents an attack or normal traffic. For example, in the logical expression test criterion used at the logic level, if the result of the logical expression is `T`, the test case is classified as attack traffic and if the result is `F`, it is classified as

[3] `www.packetfactory.net/libnet`

Snort Signature
version: 971
revision: 9
Testing Level: Signature
Testing Criterion: ATPC
Actual Value: Attack

Figure 5.6: Test Case Label Example

normal traffic.

5.4.3 Limitations of the Implementation

The current implementation of the IDSES testing model has three main limitations that reduce the number of signatures we can use. The first two limitations relate to the generation of packets from signatures. First, not all the Snort plug-ins are supported by IDSES: plug-ins `ansl`, `ftpbounce`, `ip_opts` and `rpc` are not supported. Second, we observe that the *Libnet* library fails to create packets with destination address 255.255.255.255 (the *Libnet* library has a bug that prevents us to generate test cases requiring this IP address).

The third limitation is related to the fact that the IDSES currently only supports the logic testing level. Moreover, there may be an observability issue since triggered signatures can remain silent (recall the example in Section 5.2.1). If a packet triggers a signature that contains the `noalert` `flowbits` command and Snort is silent, we are not sure whether the signature has indeed been triggered (Snort remained silent as expected) when we have a test case classified as attack traffic or whether the signature has not been triggered, which is a FN_D. A solution to solve this problem is, for testing purposes only, to remove the `noalert` command from the signatures: if Snort remains silent, then this is clearly a failure. However, we did not implement this solution since `flowbits` is related to the state machine testing level. Thus, test cases for the signatures using the `flowbits` plug-ins where not generated. These limitations are relatively minor since the majority of Snort signatures were used to generate test cases (Section 5.5).

5.5 Case Study

In this case study, our main goal is to evaluate the effectiveness of using a systematic testing model (i.e., IDSES) to assess the detection problems of IDS engines at the logic level, using different logical expression coverage criteria. Two experiments were conducted: a first experiment with IDSES using Snort 1.8.6; and an experiment with Snort 2.4.5 (i.e., the latest version available at the time of conducting this case study). We also performed experiments using other versions (i.e., 2.2.0, 2.3.0, 2.3.1 and 2.3.2) for which we observed similar results across versions and therefore we only report on version 1.8.6 and 2.4.5.

5.5.1 Design of the Experiment

5.5.1.1 IDS Signature Database

The Snort signature database for version 2.4.5 has a total of 3576 signatures while the Snort signature database for version 1.8.6 has 1266 signatures. Some of those signatures are grouped into signature description files since they have a common purpose (e.g., to monitor traffic to web sites with inappropriate content). Not all of these files are included in the set of files that Snort uses by default (remember we use the default Snort configuration). We decided to include all the signature description files. In some files, specific signatures are tagged as commented out (e.g., they are deemed to not function properly), or tagged as deleted (e.g., a signature that will disappear from the file in a future release). We did not consider the 385 commented and deleted signatures for Snort 2.4.5 (25 for Snort 1.8.6). The limitations related to the implementation of the IDSES testing model mentioned in Section 5.4.3 further reduce the number of signatures we considered: 740 additional signatures are not considered for Snort 2.4.5 and 29 for Snort 1.8.6. In particular, 612 are related to the observability problem and the usage of the `flowbits` plug-in discussed in Section 5.4.3 for Snort 2.4.5 (0 for Snort 1.8.6 since this version of Snort does not use `flowbits`). In the end, we used 2451 signatures for Snort 2.4.5 and 1205 signatures for Snort 1.8.6 with our IDSES testing model.

5.5.1.2 Incorrect Signature Specification

The implementation of our IDSES testing model uses its own module (Section 5.4.1) to transform the Snort signatures into logical expressions. The IDSES identified errors in the specification of some signatures that do not seem to be identified by the corresponding Snort IDS engine. Specifically, we identified seven incorrect signature specifications for Snort 1.8.6 and one for Snort 2.4.5.

First, no packet can trigger signature 1047 in Snort 1.8.6. This signature is looking for a string of length nine using the `content` plug-in, but this plug-in is restricted to only look at the first seven bytes of the TCP payload because of the usage of `offset:0` and `depth:7` plug-ins in the signature: the signature is trying to identify a string of nine characters by only looking at seven characters.

Second, the $ character is a reserved character for variables in the Snort signature language (see example in Figure 5.1). However, Snort 1.8.6 has signatures with the character $ in the `content` string (e.g., `content:"Admin$"`). The only way to solve this problem is to use the ACSII code of character $ in the `content` string (e.g., `content:"Admin|24|"`).

Third, it is important to note that different plug-ins have similar, but different grammars (e.g., `content` and `pcre`) that could cause errors in the signatures. Signature 663 in Snort 2.4.5 has an incorrect regular expression used by the `pcre` plug-ins. The author of the signature used the symbol | (e.g., `|3a|`) to identify an hexadecimal value in the regular expression. This is the correct way to specify hexadecimal values in the `content` plug-ins, but with `pcre` the prefix `\x` (`\x3a`) should have been used instead because the | symbol expresses an `or` in a regular expression.

It is not clear if there is an error in the IDS engine because it should have identified these errors in those signatures. In any case, all these specification errors prevent the IDS engine from detecting the attack corresponding to these signatures (i.e., evasion problem). As a result, to keep our analysis focused on conclusive results, we decided to remove these signatures from the case study. In the end, we used 2450 signatures for Snort 2.4.5 and 1205 signatures for Snort 1.8.6.

5.5.1.3 Test Criteria

Using our implementation of the IDSES testing model, we automatically generated test cases at the logic level for the selected logical expression test criteria presented in Section 5.3.2.3. We generated one test suite that meets the ATPC, one test suite that meets the PC and FC; and one test suite that meets the RAPC. For instance, a packet (test case) is generated for each of the 2450 signatures using the ATPC criterion. For each generated packet (test case), we then looked at whether or not Snort raised the IDS event of the signature used to create the packets (see Section 5.5.1.4).

For a given test requirement, many different test cases could be generated. For example, if we use test requirement tr_1 (presented in Section 5.3.2.4) to generate a test case, any IP address could be used to generate a test case for predicates B and E (i.e., B and E are true). To choose the value that should be used in the test case we use a deterministic method.

Thus, the value used for a predicate and its Boolean value in a test case is always the same for all test requirements to maintain a certain level of repeatability (i.e., always generate the same test case) and similarity across test cases. For example, the value in the test cases for predicate H is always the same for tr_1 and tr_3 to tr_9 when H is true. Similarly, the value in the test cases is always the same for predicate H for tr_2 and tr_{10} when H is false.

We choose a deterministic method to facilitate the implementation of IDSES and the identification of the root causes of the detection problems in the IDS engine. Thus, the application of a randomize method is outside the scope of this work.

5.5.1.4 Test Oracle

The IDSEF uses a specific test oracle to assess the detection accuracy of the IDS engine. The detection confusion matrix (as defined in Section 2.5) requires to classify the test cases as *Attack* or *Normal*. This classification is provided by the test case documentation field `actual value` (Section 5.4.2). The detection confusion matrix also requires to classify the results provided by the IDSs (i.e., the IDS events) into *Detected* or *Not Detected*. The `version` and `revision` fields in the test

case documentation (Section 5.4.2) are used by the IDSEF to automatically identify whether the IDS has provided the expected IDS event. A test case is classified as *Detected* when at least one IDS event provided by the IDS (the test case may trigger many signatures) has the same identifier (i.e., version and revision number) as the test case identifier (i.e., `version` and `revision` fields). Otherwise the test case is classified as *Not Detected.*

For example, when Snort provides the IDS event specified in Figure 4.6 for the test case documented in Figure 5.6, the IDSEF classifies this as a TP_D since the test case contains an attack (i.e., specified by the `actual value` field) and the test case identifiers (i.e., 971 and 9) equal to the IDS event identifiers (i.e., line 1). To assess the IDS detection accuracy, FP_D and FN_D are further investigated to identify the root cause of the problem.

Note that we also count the number of IDS events provided by a test case. A test case is expected to only have one IDS event since each test case is generated using only one IDS signature. However, we identified that some test cases often raise IDS events from signatures that were not used in the generation of the test case. These results suggest redundancies between signatures. They also suggest situations that can be exploited by an attacker as well as potential problems or errors in the signature database. We further discuss this issue in Section 5.5.2.3 and we propose in Chapter 6 a new testing model that is more appropriate than the IDSES testing model to assess this detection problem.

5.5.2 Results

In this section, we present the results obtained using the IDSES testing model on Snort 1.8.6 and 2.4.5 IDS engines. We first present the comparative analysis of the test criteria, ATPC, PC/FC and RAPC for Snort 1.8.6 and 2.4.5. Then, we describe in detail and identify the root causes of each detection problem in Section 5.5.2.1 to 5.5.2.3.

Table 5.4 presents the results for Snort 2.4.5 while Table 5.5 presents the results for Snort 1.8.6. The tables show the number of test cases generated and used when evaluating each IDS for each

test criterion and the number of TP_D, TN_D, FP_D and FN_D.

Test Criterion	Test Cases	TP_D	TN_D	FP_D	FN_D
ATPC	2450	2418	N/A	N/A	32
PC/FC	4900	2418	2450	0	32
RAPC	11072	2418	8622	0	32

Table 5.4: Snort 2.4.5 Results

Test Criterion	Test Cases	TP_D	TN_D	FP_D	FN_D
ATPC	1205	1075	N/A	N/A	130
PC and FC	2410	1075	1205	0	130
RAPC	4487	1075	3282	0	130

Table 5.5: Snort 1.8.6 Results

The time required to generate and to run the test cases is a couple of hours per IDS when using a standard desktop computer. The number of test cases required for PC/FC compared to RAPC is relatively small (i.e., RAPC requires approximatively twice the number of test cases required by PC/FC for the Snort 2.4.5 and for the Snort 1.8.6 signature databases). This situation is explained by the number of impossible test cases that cannot be generated for the RAPC (Section 5.3.2.4): the number of test requirements for RAPC is much larger but many of them are unfeasible. Thus, the cost of using RAPC compared to PC/FC is quite affordable when testing an IDS engine such as Snort.

Note that the ATPC test suite does not contain test cases with normal traffic because the ATPC test criteria only creates test cases that contain attacks (Section 5.3.2.2). Thus, there is no TN_D and FP_D for the ATPC test suite.

From these results, we can make several observations and draw a few conclusions. First, the IDSES testing model was able to identify several FN_D (i.e., cases where the IDS engine does not raise the expected IDS event) for both IDSs. In this case, we are pretty confident that all the packets have been seen by the IDSs because the IDSs analyzed the packets contained in *tcpdump* traffic traces. The ATPC seems to be sufficient to identify all the FN_D. This was expected for two reasons. A first reason is that the ATPC is sufficient to produce all the test requirements with the actual value `Attack` (Section 5.5.1.4) because the Snort signature language only specifies conjunction of predicates (i.e., the result of the corresponding logical expression can only be `true`

if every predicate is true). A second reason is that the test cases in the PC/FC and RAPC test suite for which the logical formula is true are the same as the test cases in ATPC because the Snort signatures are conjunctions of predicates and because of the deterministic method used to select the values in the test cases.

Second, only test cases documented as `Attack` have identified potential errors in the IDS engines. We have FN_D for both IDSs and we do not have any FP_D for any test criteria for both tested IDSs. As mentioned, the Snort signature language only uses conjunction of predicates. Thus, if only one predicate of the signature is false, the signature corresponding to the test case is not triggered. It seems that for the tested IDS engines, when a test case contains normal traffic (i.e., at least one predicate in the signature is `F`, see Section 5.5.1.4), the IDS engine properly ignores (i.e., does not raise an IDS event) the packets contained in the test case.

Using our IDSES testing model, we identified 32 FN_D for Snort 2.4.5 and 130 FN_D for Snort 1.8.6 that actually prevent the IDS engine from properly understanding some signatures in its IDS database. In any case, these problems prevent the IDS engine from properly detecting potential attacks. There are 32 signatures (i.e., 1.3% of 2450 tested signatures) in the Snort 2.4.5 signature database and 130 signatures (i.e., 11.8% of 1205 tested signatures) in the Snort 1.8.6 signature database for which our IDSES testing model identified at least one FN_D, indicating a potential problem with the IDS engine. In Section 5.5.2.1 to 5.5.2.3, we specify the root causes of each FN_D identified for Snort 1.8.6 and 2.4.5.

5.5.2.1 `http_inspect` Processing

The Snort 2.4.5 IDS engine (but not Snort 1.8.6) uses a preprocessor, namely `http_inspect`, to preprocess HTTP packets before they are used to trigger signatures. Specifically, `http_inspect` standardizes the format of URLs, thereby generating one URL format, with the initial objective of facilitating the analysis of URLs in plug-ins (no need to account for every possible format in each plug-in that analyzes URLs). For instance, it removes keywords HTTP and GET from HTTP

packets, it removes unnecessary consecutive slashes in URLs, translates the Unicode in ASCII, and it removes directory traversal in commands. As a result, signatures that specify that packets should contain HTTP, GET, consecutive slashes, Unicode characters or directory traversals cannot work since the information they require has been removed by `http_inspect`. Although the usage of this plug-in is known to be erroneous[4], the impact of this problem has not been fully investigated in the literature. Thanks to our study, we are now able to quantify the impact of this erroneous behaviour. Specifically, we have identified that 26 signatures are not properly understood by the IDS engine (i.e., evasion problem).

We also identified problems with six signatures that we believe are also due to the `http_inspect` preprocessor because they are similar (plug-ins) to the ones above. The four signatures that require (1) the `uricontent` to look for the hexadecimal value 0x09 or 0x0a, and the two signatures using `uricontent` that have (2) a destination port other than 80 or 8080 are not working as expected. Further investigations are required for (1), but we believe that (2) is related to the fact that `http_inspect` only processes packets on port 80 and 8080 and thus signatures using another port with the `uricontent` plug-in cannot be triggered because the packet is discarded by the `http_inspect` preprocessor. We are not aware that these problems have been documented or identified in Snort.

5.5.2.2 `stream4 detect_scans` Preprocessing

As explained earlier, in Snort, preprocessors are modules that format information contained in a packet before that information is used to evaluate signatures. Some of these preprocessors are also used to identify attacks. In the default Snort 1.8.6 configuration, the `stream4 detect_scans` parameter specifies that the `stream4` preprocessor should first (before signatures are analyzed) detect scans by attackers. However, there are also signatures in Snort 1.8.6 that are used to detect scans.

We identified that when a scan is detected by `stream4`, the signatures that should be triggered

[4] `www.snort.org/archive-3-233.html`

by the packet are not triggered. As a result, twelve signatures that detect a scan are never triggered. The impact on the IDS detection problem can be classified into four outcomes.

First, for seven signatures (signatures 1128, 621, 623, 624, 625, 627 and 630), the `stream4` preprocessor provides another IDS event than the expected one. In this case, the message of the IDS event provided by the `stream4` preprocessor to the network administrator is nevertheless similar to the message of the IDS event that should be provided if the correct signatures were triggered. As a result, it is not clear if this is actually a detection problem in the IDS engine because another part of the IDS engine compensates for the signatures that are not triggered.

Second, for signatures 629, 619 and 1133, the `stream4` preprocessor provides a generic message about the target being scanned. This message is not as precise as the one provided by the signatures. However, it still provides correct information about the attack. Here, we believe that this is an actual detection problem in the IDS engine because the information provided to the network administrator is not as precise as the one that is supposed to be provided by these signatures.

Third, for one signature (signature 626), no message is given to the network administrator by `stream4` and `stream4` does not provide the corresponding packet to this signature. Thus, we have an evasion problem in the IDS engine.

Fourth, for one signature (signature 1257), an IDS event related to a scan is given to the network administrator by `stream4`, while the corresponding signature is used to detect a DOS attack. Thus, in this case, we have an evasion problem in the IDS engine because the `stream4` preprocessor prevents the detection of the attack specified by this signature and it provides incorrect information about the attack.

In all these cases, if we deactivate the `stream4` preprocessor in the Snort configuration (i.e., we did not use the default configuration), we were able to get the expected IDS event (i.e., the right signatures are triggered and the right IDS events are sent to the network administrator).

5.5.2.3 Signature Inclusion and Intersection

Some of the root causes of the FN_D for Snort 1.8.6 can be explained by *inclusions* and *intersections* between signature specifications and to the limitations of the IDS engine. Chapter 6 provides a detailed discussion and precise definitions of the inclusion and intersection of signature problems. We refer the reader interested in more details to this chapter, and, in this section, it is sufficient to know that these problems result in a single packet (derived from one signature) potentially triggering more than one signature (i.e., the characteristics of the packet match the conditions of several signatures).

Snort limits the number of IDS events to log (i.e., a *log limit*) for a given (sequence of) packet [58]. In the case of Snort 1.8.6 the log limit is one and cannot be changed: Snort 1.8.6 only provides one IDS event for the first signature that is triggered.

In the case of Snort 1.8.6, inclusions and intersections (i.e., overlap) between signature specifications lead to FN_D. For example, some signatures (i.e., the including signatures) prevent other signatures (i.e., the included signatures) from being triggered when the including signature is checked first by the IDS. With older version of Snort such as 1.8.6, it is the order of the signatures in the signature database that determines which one is checked first and not the content length such as newer versions [58].

For Snort 1.8.6, the inclusions of signature specifications explain 119 of the 130 FN_D and the intersections between signature specification explain 4 of the 130 FN_D. In Snort 2.4.5, the inclusion/intersection problems of signatures did not cause any FN_D because none of our test cases triggers more than three signatures (the Snort 2.4.5 log limit). Nevertheless, we identified that several test cases raised more than one IDS event with Snort 2.4.5. This suggests redundancies between signatures.

We need to see the inclusion/intersection (overlapping) problems of signatures in a larger context to assess their implications for the evasion problem as they have important consequences on this

issue.

Assume that n signatures for vulnerabilities $v_1, \ldots, v_n$ overlap (i.e., that one packet can trigger them all), and that the IDS is configured to log a maximum of $n-1$ signatures (as per the log limit). Further assume, to simplify our discussion (without loss of generality), that when a packet in the overlap (inclusion or intersection) is received, the IDS (which only logs $n-1$ IDS events) and does not log the IDS event for S_i which detects attacks for v_i. One can then build a packet in the overlap that is an attack for vulnerability v_i, and this attack will not be detected by the IDS: the attacker can evade detection (i.e., evasion problem). Thus, we have a *signature overlapping problem* when the sets of packets characterized by too many signatures (more than the log limit) overlap.

The question is then which value of for the log limit should be used? There is no simple answer to that question since signature overlaps can involve varying numbers of signatures depending on the IDS signature database used. In the case of Snort, given its current design, with one log limit for all the signatures, the only safe solution is to select the maximum (to be identified) number of overlapping signatures for the database it is using. We are not aware of any technique that could calculate or identify this log limit.

IDSES is not the proper testing model to assess the signature overlapping problems as all the possible test cases for a given signature would have to be exercised to ensure that this signature does not intersect with another or that signature is included into another. For instance, in the case of the Snort signature 971 presented in Figure 5.1, several packets can trigger this signature. There are too many possibilities to generate all possible test cases that can trigger this signature. For example, think about all the test cases that can be generated using the string `.printer`. In the case of TCP (protocol used by the attack specified for Snort signature 971) the maximum payload size is 1460 bytes.[5] The questions is then how many possible strings of 1460 or less can you construct that contain the string `.printer` in the packet payload? Certainly a lot. Thus, although IDSES can shed light on the overlap problems of signatures, it is not the proper testing model to asses this

[5]The Ethernet MTU is 1500 bytes. If we removed from this maximum the IP and TCP header (20 bytes each) we have 1460 bytes available for the TCP payload.

problem.

We believe that a more appropriate testing model is required to assess the signature overlapping problems in IDS signature databases. This approach could be used to identify the *log limit*, to identify the overlapping signatures and to refine signatures (i.e., the database) to avoid signature inclusions/intersections. Such a solution is proposed in Chapter 6.

5.5.3 Discussion

As any black-box (specification-based) testing technique, when IDSES reveals a failure, the fault can either be in the implementation, in the specifications (i.e., signatures), or in the test scaffolding (i.e., the infrastructure put together to execute the tests). In the case of IDSES, one can therefore argue that the detection problems identified in Section 5.5.2 are related to incorrect implementations of the signatures since it is possible to modify the signatures to fix the detection problem. Alternatively, one can argue that these detection problems are in the IDS engine since plug-ins are part of the IDS engine (not the signature database) and that the role of the IDS engine should be to notify the network administrator that some signatures could not be properly used. Without knowing the exact intent (design specifications of the IDS engine) of the Snort developers, it is not possible to know which of these arguments hold.

Despite uncertainty about interpretation of the root causes, we still developed a new testing model that automated IDS engine testing and that actually identified detection problems in IDSs. We identified 32 FN_D for Snort 2.4.5 and 130 FN_D for Snort 1.8.6 that actually prevent the IDS engine from properly detecting potential attacks. Moreover, we also identified what we think is a new problem (i.e., signature overlapping problem) in the IDS signature database that prevents the IDS engines from properly using some signatures.

5.6 Conclusion

In this chapter, we presented the IDSES testing model, a systematic approach to test IDS engines, an area in which very little theoretical and empirical work exists. We achieved our overall objective to propose and use a testing model that can identify detection problems in IDS engines. Our IDSES testing model should eventually help improve IDS engines or establish requirements for building new IDSs.

We proposed three main levels for testing IDS engines that we refer to as predicate, signature and state machine level testing. At each level, we identified potential test adequacy criteria from the software testing literature.

We built an infrastructure to support the IDSES testing model and, as a first step, used the logical expression coverage criteria at the logic level to systematically check whether the IDS engines are at least able to trigger their signatures when expected. The proper understanding by an IDS engine of its signatures (i.e., logic level) is crucial when assessing detection problems in IDS engines and should be the minimum expected from any IDSs.

We showed that our IDSES testing model would likely apply, with minor modifications, to other signature-based network IDSs. However, we restricted our analysis to test different versions of Snort, a widely used IDS.

We were able to automatically generate test cases using four test criteria (i.e., ATPC, PC, FC and RAPC) to test Snort 2.4.5 using 2450 of its 3191 uncommented signatures and Snort 1.8.6 using 1205 of its 1241 uncommented signatures. We showed that, for this well-known IDS that has been used for many years, Snort 2.4.5 does not work as expected for more than 1.3% of its signatures and that Snort 1.8.6 does not work as expected for 11.8% of its signatures. We showed that these detection problems caused these two versions of Snort to miss attacks (i.e., evasion problem). These results obtained in a systematic way provided additional insights on what was previously reported in the literature: many more detection problems were identified, thus providing evidence of the

effectiveness of our testing model and showing that more systematic testing models are required.

Our future work will look into using state machine testing criteria, to better assess the detection accuracy of IDS engines. It will also look at extending the infrastructure that supports the IDSES testing model to handle these additional state machine test criteria and conduct a case study. We propose in Chapter 6 a new testing model to assess the detection problems related to the signature overlapping problem on which our IDSES testing model shed some light.

Chapter 6

IDS Signature Space Verification Method

6.1 Introduction

In Chapter 5, IDSES identified that an IDS can miss an attack when many signatures (more than the log limit) specify the same group of packets, a problem that we refer to as the *signature overlapping problem.* One likely reason for overlapping signatures is the necessity for IDS developers and the IDS community to constantly update the signature database to cope with the new attacks and vulnerabilities that are regularly identified. For instance, because of the diversity of network attacks (based on the National Vulnerability Database, more than 4900 new software vulnerabilities have been identified every year since 2005), we can see a quick growth in the number of signatures used by IDSs. For example, while Snort used around 3000 signatures in 2005, there are more than 15 000 signatures in its database in 2010. Thus, the number of signatures has increased 5 times in 5 years (while it only doubled from 2002 to 2005).

Consequently, when the set of added or modified signatures increases, it becomes more and more

difficult to maintain a coherent and consistent set of signatures, and errors are made or inconsistencies are introduced. As discussed later, this is an important problem since overlapping signatures can lead to (1) evasion attacks (i.e., an attack not being detected) and (2) more aggressive squealing attacks (i.e., IDS can be stimulated by synthetic attacks that overwhelm the network administrator with IDS events) that could eventually have a negative impact on the IDS performance (e.g., the IDS begins to miss packets), which can result in a denial of service on the IDS itself.

To date, there has not been any attempt to study this signature overlapping problem. Only anecdotal evidence of the problem has been reported (like in our Chapter 5), and none of the IDS verification and validation techniques proposed so far (Chapter 3) is adequate to specifically study this problem. No solution has so far been proposed to systematically study and quantify the signature overlapping problem in a signature database. Identifying a method to systematically quantify the problem would be helpful from different points of view. First, the method could help explain in qualitative and quantitative ways what practitioners observe. Second, the method could point to problems to be fixed in signature databases and help future modifications of signature databases to avoid overlapping signatures. Third, the method could help improve IDSs, as improvements to signatures could reduce IDS evasion problems and prevent some squealing attacks.

Someone could suggest designing the IDS engine in such a way that it can cope for the overlapping signature problems. We are not sure whether this is possible. We argue that it is always better to understand the extent of a problem (i.e., analyzing the signature database) before proposing a solution (i.e., proposing modification to the IDS engine or to the signature database).

In this chapter, we propose an approach (which is based on set theory and automaton theory) to precisely and systematically study the signature overlapping problems in a signature database. Our approach attempts to identify whether a model (in our case a set of signatures) has expected properties (in our case, no inclusion, no intersection and no equality). We have implemented our solution into a tool and applied it on the signature database of one well-known IDS, namely Snort [18]. Results show the (unexpected) extent of overlapping signatures in this database. To the best of our

knowledge, this is the first time such systematic quantitative analysis is attempted and reported in the literature.

The contribution of this chapter is four-fold.

- We propose an approach based on set theory and automaton theory to quantify the signature overlapping problem in an IDS signature database. When doing so, this approach leads to a systematic analysis of the signature overlapping problem.
- We illustrate how this approach can be applied to a widely-used IDS.
- We succinctly present a tool that can automate our approach.
- We perform a case study to assess the effectiveness of this approach on that IDS. Results show that our approach is effective at systematically revealing overlapping situations in the signature databases of this IDS and we present a real attack that exploits overlapping signatures.

The remainder of this chapter is structured as follows. We present examples of the signature overlapping problem (Section 6.2) and then show its relevance for the IDS community (Section 6.3). We then detail our approach to systematically quantify the signature overlapping problem in IDS signature databases (Section 6.4). Section 6.5 discusses the tool support for our method. The method and tool are used in a case study: Section 6.6. Conclusions are drawn in Section 6.7.

6.2 Signatures Overlapping Examples

The intent of this section is not to formalize the definition of how and when signatures overlap (which is done in Section 6.4), but to provide intuitive examples of this problem. First, we illustrate the signature overlapping problem using simple, abstract signatures and then illustrate it with Snort.

6.2.1 Simple Example

Suppose we have three signatures: S_1 which looks for the string `A`, S_2 which looks for the string `AB` and S_3 which looks for the string `BC`. These signatures only look at the payload of packets. Assume that in this case (which is the case in most IDSs), this means that the IDS is checking if the substring specified by the signature (e.g., `A`) is contained or not in a packet payload (e.g., `PAYLOAD`) to detect an attack (which is the case here, because `A` is in `PAYLOAD`).

In this example, *all* the packets that can trigger S_2 will also trigger S_1 since any packet payload (string) that can be constructed that matches the specification of S_2 always matches the specification of S_1: if the payload contains `AB`, it contains `A`. Moreover, *some* of the packets (not all) that trigger S_2 also trigger S_3 at the same time since it exists packet payloads that match the specifications of S_2 and S_3: i.e., any string containing `AB` and `BC`. Furthermore, this same group of packets can trigger S_1, S_2 and S_3 at the same time. In fact, a packet with the payload `ABC` will trigger the tree signatures.

6.2.2 Snort Example

To illustrate the signature overlapping problem with concrete IDS signatures, we use Snort. As explained in Section 5.2.2, a Snort signature is composed of plug-ins (Snort terminology) and raises an IDS event if all its plug-ins return (evaluate to) `true`. For example, in the signature of Figure 6.1, `content` (line 4) is a plug-in that evaluates the content of the packet payload, and `content:"CWD"` is a predicate that verifies whether a packet contains the string `CWD` (not case sensitive, line 5) in its packet payload. The `pcre` (line 6) is a plug-in that evaluates the content of the packet payload using a regular expression (see below). Lines 7-8 specify meta-data, providing the identification and the revision number of the signature.

Figure 6.2 shows another Snort signature. Both S_{1672} (Figure 6.1) and S_{336} (Figure 6.2) come from the same Snort signature database, specifically the signature database of Snort 2.4.5.

```
1  alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (
2  msg:"FTP CWD ~ attempt";
3  flow:to_server,established;
4  content:"CWD";
5  nocase;
6  pcre:"/^CWD\s+~/smi";
7  sid:1672;
8  rev:11;)
```

Figure 6.1: Snort Signature 1672

```
1  alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (
2  msg:"FTP CWD ~ root attempt";
3  flow:to_server,established;
4  content:"CWD";
5  nocase;
6  content: "~root";
7  distance:1; nocase;
8  pcre:"/^CWD\s+~root/smi";
9  sid:336;
10 rev:10;)
```

Figure 6.2: Snort Signature 336

We argue here that *any* (sequence of) packet that triggers S_{336} also triggers S_{1672}. First, as can be seen from the two figures, lines 1, 3-5 are the same (line 2 does not specify constraints on packets).

Second, the regular expression at line 6 in Figure 6.1 specifies that the packet payload has to start (i.e., ^) with the string `CWD` followed by at least one (i.e., +) space character (i.e., \s) and then followed by the string ~. The regular expression is delimited by two characters `/` and `smi` (read options `s`, `m`, and `i`) are regular expression modifiers [18] (typically used to indicate whether the search is case sensitive and how to handle new lines in the string). Thus, a packet payload that matches the regular expression of line 6, will match the constraints specified in lines 4-5. Consequently, lines 4-5 can be removed without changing which packets are matching this signature. The `content` and `pcre` plug-ins are often looking a the same bytes in Snort signatures for performance reasons (Section 5.3.2.4).

Third, lines 4-7 in Figure 6.2 specify that the string ~root (not case sensitive, line 7) has to be

at a distance of one byte (line 7) after the string `CWD` (not case sensitive, lines 4-5) in the packet payload. Line 8 in Figure 6.2 specifies the same thing as line 6 in Figure 6.1 with the exception that the string $\sim$ has to be followed by the string `root`. Here the regular expression of line 8 could be rewritten to `/^CWD\s~root/smi` (i.e., removing the + sign) without changing which packets are matching this signature since the `distance` plug-in restricts a distance of one byte between `CWD` and `~root`. In fact, this rewritten regular expression can replace lines 4-8.

Thus, the regular expression at line 6 in Figure 6.1 would match any string that meets the specification of the rewritten regular expression of line 8 in Figure 6.2. Specifically, S_{1672} does not require the string `~root` one byte after the string `CWD` (as specified in lines 6-7 of Figure 6.2). As a consequence, *any* (sequence of) packet that triggers S_{336} also triggers S_{1672}.

```
1  alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (
2  msg:"FTP large PWD command";
3  flow:to_server,established;
4  dsize:10;
5  content:"PWD";
6  nocase;
7  sid:1624;
8  rev:6;)
```

Figure 6.3: Snort Signature 1624

Figure 6.3 shows another Snort signature from the same Snort 2.4.5 signature database. We can argue that *some*, but not all (sequences of) packets that trigger S_{1624} also trigger S_{1672} (and vice-versa) since (sequences of) packets exist that can trigger both signatures at the same time: for instance, a packet within an open session with a destination port of 21 that contains the string `"CWD ~PWDxx"` does. One can argue that this packet is not an attack, but an attacker could use it to mislead the network administrator (e.g., squealing attack) or prevent the IDS from detecting attacks (Section 6.6.2.2).

Note that, it is *not* possible to trigger S_{336} and S_{1624} on the same packet since we cannot have a packet that contains a payload that matches both signatures. First, the packet payload has to be of size 10 (line 4 in Figure 6.3). Second, it has to match `/^CWD\s+~root/smi` (line 8 in Figure 6.2)

that requires a string of at least 9 characters minimum (e.g., `CWD ~root`). Third, the packet payload has to also contain the string `PWD` (line 5 in Figure 6.3) which is 3 characters long. Thus, a packet payload matching line 5 in Figure 6.3 and line 8 in Figure 6.2 needs 12 characters which contradicts line 4 in Figure 6.3. Thus, S_{336} and S_{1624} can not be triggered on the same packet.

These signature overlapping examples are simple to identify: e.g., someone familiar with the Snort language would identify them easily if told to look at those signatures. But looking for every signature overlaps in an IDS signature database (with 3 000 to 15 000 signatures) has to be automated. Moreover, the payload plug-ins such as `content` and `pcre` have to be considered together since they may be looking at the same bits in the packet payload (which is the case in Figure 6.1 and Figure 6.2). Manually deciphering what those plug-ins specify is not an easy task. Consequently, a systematic approach is required for analyzing the signature overlapping problem, such as the one presented in this chapter.

6.3 Relevance of the Signature Overlapping Problem

Modern IDSs, such as Snort 2.4.5 (the latest version where signatures can be shown to any unregistered user) and Snort 2.8.5 (released in 2010), include the ability to control which signatures should be triggered in the case that a packet matches multiple signatures (e.g., the order in which the signatures are triggered is based on the priority and impact of the attack) and the ability to log multiple IDS events (i.e., corresponding to the triggered signatures) on a single packet.

In older IDS versions, such as Snort 1.8.6, the overlapping situation had security implications for an IDS. For instance, older Snort versions can only trigger one signature, even if more than one signature is true, given the packet's characteristics. Suppose that a packet satisfying the conditions of S_{336} is observed by Snort, and that it happens (because of implementations details in the Snort engine) that S_{1672} is verified before S_{336}. Then Snort raises an IDS event because the packet satisfies S_{1672}. Snort would be silent about S_{336}: there is an evasion attack (which is common in older Snort

version, see Section 6.6). Fortunately, S_{1672} and S_{336} are related to the same type of attack, i.e., to the same vulnerability. In this case, their respective message provided by the `msg` plug-in are similar, so the network administrator will see an IDS event about the right attack/vulnerability. However, it is still possible for an attacker to exploit this using overlapping signatures that detect different attacks/vulnerabilities, which is the case if the attacker would have exploited the overlap between S_{1672} and S_{1624} (suppose that S_{1624} is verified first). In this case, the attacker misleads the network administrator since these signatures detect different attacks/vulnerabilities. Overlapping signatures that detect different attacks/vulnerabilities are referred to as orthogonal signatures. Otherwise, the signatures are non-orthogonal.

This is probably one of the main reasons why modern IDSs have the ability to log multiple IDS events triggered (from signatures) on a single packet. However, even if they are able to log multiple IDS events, IDSs such as Snort still limit the number of IDS events considered to be logged and the actual number of IDS events logged. In the case of Snort, it limits the IDS events to be considered for logging on one packet or stream.

Note that the notion of stream is important in IDSs such as Snort. Snort reassembles the TCP and UDP packets into one or more larger packets (i.e., a stream) allowing the signatures to match packet payloads that are split across several TCP or UDP packets [58]. Snort limits the IDS events to log using the `event_queue` configuration. For instance, using the configuration `config event_queue: max_queue 8 log 3 order_events content_length` in Snort 2.4.5, we can configure, using `max_queue`, how many IDS events will be considered (will be stored in the event queue) for a single packet or stream (i.e., 8), using `log`, how many IDS events to log for a given packet or stream (i.e., 3) and, using `order_events`, we can specify how the incoming IDS events are ordered in the event queue (i.e., signatures that have a longer string in the `content` plug-in are ordered before signatures with shorter strings). The most important part is that Snort cannot log more IDS events than the number that is specified by `log`. For instance, Snort 2.4.5 and Snort 2.8.5 have a default log limit of three.

Why do IDSs such as Snort still limit the number of IDS events to log? We were unable to find any documentation on this subject. At this point, we can only speculate on the Snort developer intentions. We came up with two reasonable hypotheses: the decision is based either on accuracy or performance assessment of Snort.

First, the Snort developers know that signatures can overlap and that there is an *accuracy impact.* Since Snort is still vulnerable to squealing attacks using synthetic attacks, the developers have decided to limit the number of IDS events that could be logged to mitigate this type of attack. Note that we demonstrated in Chapter 5 that Snort versions such as Snort 2.4.5 are still vulnerable to squealing attacks. An attacker could create a more effective squealing attack by using the overlapping signatures in an IDS signature database. Using the signature overlap, the attacker, with only one packet, can now trigger multiple signatures (e.g., S_{336} and S_{1672}) which is was not considered in the development of tools such as Snot, Stick and Mucus. In fact, based on the results presented in Section 6.6.2, an attacker can easily trigger three IDS events (which is the default Snort log limit) using only one packet. Moreover, it is theoretically possible, using one packet, to trigger n IDS events (where $n \leq 25$) when the log limit is 25. Based on the results obtained using our verification method (Section 6.6.2), and some manual analysis of the Snort 2.4.5 signature database, we identified a packet payload that has triggered 25 signatures when tested against Snort 2.4.5. Table 6.1 shows the list of signatures.

S_{882}	S_{987}	S_{1301}	S_{1594}	S_{1612}
S_{907}	S_{1107}	S_{1455}	S_{1598}	S_{1740}
S_{936}	S_{1113}	S_{1536}	S_{1599}	S_{1741}
S_{928}	S_{1160}	S_{1537}	S_{1601}	S_{1806}
S_{930}	S_{1189}	S_{1593}	S_{1602}	S_{2145}

Table 6.1: 25 Overlapping Signatures in Snort 2.4.5

Consequently, a large number of overlapping signatures makes the squealing attack more efficient than ever since fewer packets (a packet triggers multiple signatures) have to be sent to the IDS to have a similar impact to the original squealing attack (where a packet triggers only one IDS event and the attacker has to send many packets). Note that we could easily have generated a packet that

could trigger more than 25 signatures (when the log limit is bigger than 25). However, without a systematic analysis of the signature overlapping problem in the IDS signature database, it is not possible to know what is the maximum number of signatures that could be triggered on one packet (Section 6.6). The fact that it is possible to trigger many signatures (i.e., at least 25) on a single packet leads us to our second hypothesis.

Second, the Snort developers know that signatures can overlap. There is a *performance impact* on the IDS when many signatures are triggered since it requires I/O operations to store the IDS events in a log file. This hypothesis is also supported by some statements in the Snort official documentation.[1] For instance, the Snort official documentation states that *"If you want Snort to go fast (like keep up with a 1000 Mbps connection), you need to use unified logging and a unified log reader such as barnyard.*[2] *This allows Snort to log alerts in a binary form as fast as possible while another program performs the slow actions, such as writing to a database. If you want a text file that's easily parsable, but still somewhat fast, try using binary logging with the "fast" output mechanism."* To check this hypothesis, we reused the packet payload that triggered 25 signatures. We conducted a rudimentary experiment to confirm our hypothesis that Snort limits logging of IDS events for performance reasons. In this experiment, we generated a traffic trace for which we sent this packet payload (that triggers 25 signatures) an arbitrary number of times (i.e., 16). We then provided this traffic trace to Snort 2.4.5. We used the Snort 2.4.5 default configuration (i.e., log the IDS events in a file). For each test, we changed the log limit (i.e., 1, 3, 5, 10, 15, 20 and 25) to identify the performance impact on the time required by Snort to analyze the traffic trace. Table 6.2 shows that indeed increasing the log limit (i.e., to 25 IDS events) has an impact in some Snort logging configurations when one packet can trigger multiple signatures (i.e., 25 signatures). These tests were conducted on a Dell 1950 with two quad core Intel Xeon E5430 processors (2.66 GHz), 32 GB of RAM, using a DELL MD3000i 15 TB SCSI disk and running Windows 2008 (64-bit).

Consequently, increasing the log limit too much could lead to *accuracy* and *performance impacts*

[1] `www.snort.org/assets/125/snort_manual-2_8_5_1.pdf`
[2] `sourceforge.net/projects/barnyard/`

Log Limit	Processing Time (sec)
1	1.2
3	1.8
5	3.2
10	6.3
15	8.5
20	12.4
25	15.5

Table 6.2: Log Limit Versus Processing Time for Snort 2.4.5

on IDSs such as Snort. However, having a log limit too low (e.g., one) could lead to evasion attacks when many orthogonal overlapping signatures (e.g., S_{1672} and S_{1624}) are triggered on the same packet. With these two hypotheses, we have an idea why IDSs such as Snort have a log limit. However, the essence of the question remains. (1) What is the extent of the signature overlapping problem in a signature database? (2) Moreover, is it possible to exploit a group of orthogonal overlapping signatures (bigger than the log limit) to hide (prevent the IDS from triggering the IDS event related to) a real working attack (not a synthetic attack) and make it trigger orthogonal signatures to the real attack, leaving the attack unnoticed? We call this attack the *sneaky squealing attack.*

In this chapter, we propose a new approach that could answer question 1 and help answering question 2. We propose a new approach to precisely and systematically identify overlapping signatures in a signature database (Section 6.4). We used this verification method to assess the signature overlapping problem in different versions of Snort and show that it is possible to make a *sneaky squealing attack* (Section 6.6).

6.4 A Set/Automaton Theory Approach to the Signature Overlapping Problem

An approach to analyze the signature overlapping problem in any IDS would require that we possess an IDS signature (meta-)language, describing every possible IDS signature language, from which to build our approach. Unfortunately, to the best of our knowledge, this (meta-)language does not

exist. It is not even clear whether building that language is realistic given the peculiarities of IDS languages, and creating such a language is well beyond the scope of our research. Fortunately, network IDS signature languages have a sufficiently large number of common functionalities that we can describe the main modeling capabilities of that (meta-)language.

In Section 3.3 we also explained that we can reasonably assume, without loss of generality, that IDSs have a signature language that can be easily abstracted (meta-language) into a state machine, where the guard conditions of events triggering state changes are logical expressions made of predicates, where predicates use boolean functions (i.e., predicate functions) checking packet characteristics. Thus, the specification of an attack into an IDS language is equivalent to specifying a state machine. Consequently, the IDS manages multiple state machines at the same time (i.e., one for each attack specified). When a new packet is captured by the IDS, a state change occurs on all the state machines that have a guard condition (e.g., a signature) which matches this packet.

In the case of our analysis of the signature overlapping problem, we want to know if certain guard conditions could be missed (no transition occur) by the IDS because of the log limit (which could limit the number of guard conditions that can be triggered).

Consequently, the problem is to analyze the overlap between guard conditions. Guard conditions (i.e., signatures) are made of predicates where predicates specify *constraints* on the protocol fields and the packet payloads. For instance, in Snort, `$EXTERNAL_NET any -> $HOME_NET 21` are five predicates (Section 5.3.2.1) that specify constraints on protocol fields (i.e., source and destination IP addresses, and source and destination TCP/UDP port), and `pcre:"/bla/"` and `content:"foo"` are two predicates that specify constraints on the packet payload. Usually, in IDS signature languages, range of values are used to specify constraints on fixed sets of possible values (e.g., the protocol fields), and substrings or regular expressions (which can all be represented as an automaton) are used to specify constraints on large (possibly infinite) sets of values (e.g., the packet payload). In this case, a *constraint* is a set of values or an automaton. Thus, the signature overlapping problem is a problem of intersecting sets of *constraints*. Consequently, it is natural to rely on set theory (for

the protocol field constraints) and automaton theory (for the packet payload constraints) to devise a method to systematically study the signature overlapping problem. Moreover, set and automaton theory provide a uniform way of reasoning about overlapping constraints since they define the $\cup$, $\cap$ and $\setminus$ operators which can then be used to identify overlapping signatures where signatures are logical expressions of *constraints.*

This set and automaton theory approach entails three different analyses of an IDS signature database: analyzing the signatures for *equality*, *inclusion* and *intersection.* Although this chapter defines the entire approach, our tool and case study focus on the *equality* and *inclusion* analyses. The *intersection* analysis, being more computationally intensive (Section 6.5.1), is left for future work.

In this section, we present how each signature is converted into a set of constraints (Section 6.4.1), we show how Snort signatures can be converted into sets of constraints (Section 6.4.2), and we define set/automaton theory operators in this context (Section 6.4.3).

6.4.1 Signature Conversion

To convert IDS signatures into sets of constraints, a function *convSig*() has to be defined which converts every signature i of an IDS d, which we note ${S_i}^d$, into the set of constraints $C_{{S_i}^d}$ (i.e., $convSig({S_i}^d) = C_{{S_i}^d}$).

To define the function *convSig*(), we need to propose a representation for $C_{{S_i}^d}$. Remember that $C_{{S_i}^d}$ expresses constraints on packets. Packets are well-structured objects since they are generated based on specific protocol specifications, which in turn have well-documented fields. The protocol fields have fixed length (e.g., a maximum of 32 bits) and therefore can be represented into finite sets of values (with the exception of the packet payload). They can therefore be converted into ranges of values, specified with minimum and maximum allowed values, (i.e, sets of values) and associated with a protocol header field. These ranges of values can then be used with the $\cup$, $\cap$ and $\setminus$ operators.

It is inappropriate to convert constraints on packet payload into ranges of values. Modern IDSs

do not only look at the packet payload, but also at the stream (i.e., the combined sequence of multiple packet payloads). IDSs reassemble the TCP and UDP packets into one or more larger packets (i.e., a stream), allowing the signatures to match packet payloads that are split across several TCP or UDP packets. Thus, the sequence of information (of packet payloads) exchanged between the computers in a TCP/UDP session may be extremely long, and the content of the payload(s) can have a very large set of possible values. Consequently, regular expressions are often used to specify constraints on packet payloads. However, regular expressions can not be directly used with operators such as $\cup$, $\cap$ and $\setminus$, which are required to identify signature overlap problems. In this case, we used a finite state automaton (FSA) representation for the packet payloads since algorithms [59] exist to convert a regular expression into a finite state automaton.

Of course, the only information required in ${S_i}^d$ to create $C_{{S_i}^d}$ is all the predicates in the signature that characterize the values of protocol fields and the strings and bytes in the packet payload. For example, in the case of S_{1672} (Figure 6.1), the information on line 1 and lines 3 to 6 is used by *convSig*(). Thus, metadata about vulnerability number, for instance, is not used.

Let F be the set of protocol fields and Π be the set of protocols. For each $\pi \in \Pi$, F^π is the set of fields in π, that is, $F^\pi = \{f \mid f \in \pi\}$. F is the union of each F^π in each protocol in Π. Let V^f be the set of possible values of field f (i.e., all possible values allowed by f based on its protocol specification) and $V^f_{{S_i}^d}$ the set of values that is specified (required) for f by ${S_i}^d$.

Definition 6 *A set of constraints $C_{{S_i}^d}$ is a vector (i.e., a set) of $V^f_{{S_i}^d}$, where $V^f_{{S_i}^d}$ is represented using a range of values when f does not represent the packet payload and an automaton otherwise.*

Consequently, a signature S^d_i, characterizing the protocol fields and the packet payload, is now represented as a set of $V^f_{{S_i}^d}$, one of which is specified as a FSA (for the packet payload) and the others as ranges of values. These ranges of values and FSA can then be used with the $\cup$, $\cap$ and $\setminus$ operators to assess the signature overlapping problem.

Although we describe our approach so it can be applied to any IDS, the function *convSig*() is

specific to each IDS signature language. This is an important aspect of our approach since every single IDS (language) has its own peculiarities that need to be accounted for, thus requiring that we tailor the conversion function *convSig*(). Consequently, in this chapter, the conversion function *convSig*() is described in the context of Snort, but could be easily described in the context of other IDSs similar to Snort (that use a state machine and first order logic as guard conditions).

6.4.2 Conversion of Snort signatures

To calculate $C_{S_i^{Snort}}$ from S_i^{Snort} in the case of Snort, we need to convert the *header*, the *non-payload* and the *payload* families of plug-ins [18] since these are the ones that looks at characteristics of packets.

For instance, the header and non-payload plug-in families can be mapped to a specific protocol header field. For example, S_{1672} (Figure 6.1) specifies the IP Source Address field (i.e., `$EXTERNAL_NET`). Thus, all the protocol header fields managed by the header and non-payload plug-ins of the Snort signature language are converted by *convSig*() into ranges of values.

Using S_{1672} (Figure 6.1) as an example, the possible value of $V_{S_{1672}^{Snort}}^{TCP_{destport}}$ for the destination port of a TCP packet is 21 (end of line 1 in Figure 6.1). In this case, S_{1672} does not have constraint for the source port, thus $V_{S_{1672}^{Snort}}^{TCP_{sourceport}} = [0, 65535]$. In fact, when a signature does not have a constraint for a field f, $V_{S_i^d}^f = V^f$.

There are several payload plug-ins (e.g., `content` and `pcre`) that are used to look into the packet payload, as opposed to the header and non-payload plug-ins where only one specific plug-in is used to look into a specific protocol field. This happens often with the `content` and `pcre` plug-ins in Snort whereby `content` is used to specify a constraint at a high-level of granularity and `pcre` is used to specify a finer grain constraint. For instance, in Figure 6.1, `content` specifies that the payload should contain the string `CWD` and `pcre` specifies that `CWD` should be followed by something else in the string. Such a combination of `content` and `pcre` in Snort signature is often used for performance reasons (Section 5.3.2.4), whereby the second, complex to evaluate (in Snort) constraint (`pcre`) is

only evaluated if the first, simpler to evaluate constraint (`content`) evaluates to true.

Furthermore, the Snort signature language specifies that some payload plug-ins must be used together, and if this is not the case, there is a syntax error in the signature. These plug-ins are `nocase`, `rawbytes`, `depth`, `offset`, `within` and `distance`: if used, they must be used with either `content` or `uricontent`.

In the case of Snort, to convert these plug-ins into one FSA, we proceed as follows. First, we convert each payload plug-in ($Plugin_j$) occurrence in a signature S_i into a regular expression (RE_j).[3] We used a similar approach to *s2b* (i.e, a Snort signature converter that is part of Bro [19]) to convert Snort plug-ins into regular expressions. However, the objective of *s2b* is not to analyze the signature overlapping problem, but simply to convert the Snort signatures into Bro signatures since Bro uses regular expressions to look into the packet payload. Moreover, we improved the concept of *s2b* since *s2b* does not convert some Snort payload plug-ins (e.g., `byte_test`). Then, we used known algorithms [59] to convert each regular expression (RE_j) into a finite state automaton (FSA_j). Finally, we combine (intersect) each FSA_j into $FSA^{payload}$ (i.e., $FSA_1 \cap ... \cap FSA_n = FSA^{payload}$) and we obtain $V^{payload}_{S_i{}^{Snort}} = FSA^{payload}$.

This FSA representation provides a common representation for the Snort payload plug-ins. Thus, different plug-ins can be combined into one FSA for the whole payload using a well-defined FSA intersection algorithm [59]. The regular language accepted by the resulting $FSA^{payload}$ is the *largest* language accepted by all FSAs used in the intersection. This is the intended purpose as it meets the Snort signature specification language.

6.4.3 Definitions

In our approach, a signature S^d_i, characterizing the protocol fields and the packet payload, is now represented as a set of $V^f_{S_i{}^d}$, one of which is specified as a FSA and the others as ranges of values. Thanks to the signature conversion mechanism discussed previously, we can now formalize the

[3] With the exception of `byte_jump`, which is not implemented yet.

notions of signature inclusion, intersection and equality.

Definition 7 *Signature S_i^d (strictly) includes signature S_j^d ($S_j^d \subset S_i^d$) if and only if $C_{S_j{}^d}$ is strictly included into $C_{S_i{}^d}$. In other words: $S_j^d \subset S_i^d \Leftrightarrow \forall f \in F, V_{S_j^d}^f \subseteq V_{S_i^d}^f \wedge \exists f \in F, V_{S_j^d}^f \subset V_{S_i^d}^f$.*

Definition 8 *Signature S_i^d intersects signature S_j^d ($S_i^d \cap S_j^d$) if and only if $C_{S_i{}^d}$ and $C_{S_j{}^d}$ are different and have a non-empty intersection. In other words: $S_i^d \cap S_j^d \Leftrightarrow S_i^d \neq S_j^d \wedge \forall f \in F, V_{S_j^d}^f \cap V_{S_i^d}^f \neq \emptyset$.*

Definition 9 *Signature S_i^d equals signature S_j^d ($S_i^d = S_j^d$) if and only if $C_{S_i{}^d}$ and $C_{S_j{}^d}$ are equal. In other words: $S_i^d = S_j^d \Leftrightarrow \forall f \in F, V_{S_j^d}^f = V_{S_i^d}^f$.*

6.4.4 Signatures Overlapping Examples

If we go back to the examples of S_{1672}, S_{336} and S_{1624} presented in Figure 6.1, 6.2 and 6.3: we have the following relationships: S_{1672} *includes* S_{336} and S_{1672} *intersects* S_{1624}.

It is obvious that S_{1672} *includes* S_{336} since all constraints of both signatures are the same with the exception of the constraints on the packet payload where (we can easily understand by looking at the corresponding plug-ins) that $V_{S_{1672}^{Snort}}^{payload}$, the automaton generated from lines 4-6, accepts all the strings accepted by $V_{S_{336}^{Snort}}^{payload}$ the automaton generate from lines 4-8.

We can use the same argument to explain that S_{1672} *intersects* S_{1624} since once again all constraints of both signatures are the same with the exception of the constraints on the packet payload. Intuitively, two automata intersect if we can build an automaton that accepts the intersection of the languages of the given automata. In this case, $V_{S_{1672}^{Snort}}^{payload}$ and $V_{S_{1624}^{Snort}}^{payload}$, (i.e., the lines 4-6 in Figure 6.1 and 6.3) intersect since there is a string (e.g., `CWD ~rootPWDxx`) that is accepted by both automata (i.e., the intersection of the languages accepted by these two automata is not empty).

However, S_{336} *does not intersect* S_{1624} since the intersection of the languages accepted by the $V_{S_{336}^{Snort}}^{payload}$ and $V_{S_{1624}^{Snort}}^{payload}$ automata is empty. In fact, it is not possible to identify a string of size 10

(i.e., line 4 in Figure 6.3) that matches the regular expression of S_{336} (i.e., line 8) and `content` of S_{1624} (i.e., line 5).

6.5 IDS Signature Space Analyzer

In this section, we present how we automatically analyze the signature overlapping problem in an IDS signature database for overlapping signatures according to our approach (Section 6.5.1). We then discuss current limitations of our tool (Section 6.5.2).

6.5.1 Implementation

Our supporting tool, called IDS Signature Space Analyzer (IDS-SSA) works in two steps. First, (step 1) each IDS signature of an IDS is analyzed and transformed by the *Parser* module into a set (or automaton) representation using the approach described in Section 6.4.2. This *Parser* is necessarily specific to the IDS being tested, but different parsers can easily be created for different IDSs since, as we discussed earlier, network IDS languages have common features. The *Parser* relies on `dk.brics.automaton`[4] to convert regular expressions into FSA. Our first implementation of this framework targets Snort, i.e., we created a parser for Snort signatures (step 1). This parser can handle Snort databases from version 1.8.6 to 2.4.5 (see Section 6.6).

Second, (step 2) these set/automaton representations of the IDS signatures are fed to an *Analyzer* module that searches for overlaps between signatures: the *Analyzer* implements Definition 7, Definition 8 and Definition 9. Again, this relies on library `dk.brics.automaton` for different operators on FSA (e.g., intersection). Note that the *Analyzer* module is independent from the IDS (language).

Observe that since inclusion and equality are transitive relations, inclusion chains (e.g., $S_i^d \subset S_j^d \subset S_k^d$) and equality chains (e.g., $S_i^d = S_j^d = S_k^d$) of length greater than 2 can be computed at relatively low cost once all the pair-wise inclusions/equalities have been computed. Computing

[4] `www.brics.dk/automaton`

inclusion or equality chains can be done in time $O(n^2)$ where n is the number of signatures. We implemented the equality and inclusion algorithms.

However, computing intersection chains (e.g., $S_i^d \cap S_j^d \cap S_k^d \neq \emptyset$) of length greater than 2 is more costly than for equality and inclusion since we cannot take advantage of transitivity. Although we can rely on the intersection operation provided in `dk.brics.automaton`, the lack of transitivity requires that we compute first the intersection of every pair of signatures. This would not be enough for our problem as we would also like to know which triplets of signatures intersect, which 4-tuples of signatures intersect, ... This problem is clearly very computationally intensive. Snort 2.4.5 has 3576 signatures! For instance, there are

$$\binom{n}{2} + \binom{n}{3} + \binom{n}{4} + \cdots \binom{n}{n}$$

(n being the total number of signatures in the analyzed database) subsets of signatures for which we should identify whether there is an intersection or not. This represents the worst case scenario where every signature in the IDS database intersects with one another (i.e., $S_1^d \cap \ldots S_n^d \neq \emptyset$). Obviously, in practice we would not need to determine all those subsets of signatures because there is an optimal log limit (recall Section 6.3). A possible way to proceed could be to identify the

$$\binom{n}{2}$$

subsets which can be done with combinatorial algorithms and identify those that intersect. This is similar to identifying *cliques* in a graph. A *clique* is a subset of nodes in a graph G, such that for every two nodes in this subset, it exists an edge connecting the two. Thus, a clique K_n is a *complete* subgraph of n nodes in G. For instances, if we identified that the following pairs intersect, $S_i^d \cap S_j^d \neq \emptyset$, $S_j^d \cap S_k^d \neq \emptyset$ and $S_i^d \cap S_k^d \neq \emptyset$. We can construct a graph where the edges represent the relationship *intersect with* and the nodes are the signatures. In this case, we do not have to

calculate the

$$\binom{n}{3}$$

possible intersection chains to identify which triplets of signatures intersect since only the cliques K_3 make it possible (i.e., a clique is a necessary condition, but not a sufficient condition) for triplets of signatures to intersect. In this case, we have a clique K_3 where we could have $S_i^d \cap S_j^d \cap S_k^d \neq \emptyset$. Thus, to identify which n-tuplets of signatures intersect, we have to identify the number of K_n cliques in a graph. However, a K_n clique is only a necessary condition. It is not a sufficient condition to have a n-tuplets. We still have to verify that the signatures in a clique actually intersect (i.e., provide a none-empty intersection).

Consequently, if $|K_i|$ cliques are found, we test each of them for intersections, then we repeat, increasing the value of i until we find a value of i such that there is no clique or that none of the signatures in the cliques intersects. Thus, the optimal log limit is the clique of the largest possible size with signatures that actually intersect in the graph.

To summarize, since this solution is computationally intensive (indentifying cliques in a graph runs in exponential time), given the very large number of signatures in Snort, we left the problem of identifying those intersections to future work.

The case study section therefore reports on results where we systematically study signature inclusions and equalities.

6.5.2 Limitations of the Implementation

Besides the fact that our framework currently focuses on Snort and on the signatures equality and inclusion problems, it has two other limitations that slightly reduce the number of signatures we can currently use (Section 6.6). First, we focused on the most popular plug-ins in Snort signature and those that could be easily integrated in our approach. For instance, `byte_jump`, `flowbit`, `asn1`, `rpc`, `sameip`, `ftpbounce` and `threshold` are currently not supported. Consequently, the signatures using

these plug-ins cannot be converted into sets of packets by our supporting tool. Second, we observed in our case study that the `dk.brics.automaton` library fails to create automaton for some signatures (it returned an error message), and therefore we cannot use these signatures in our analysis. Despite these two limitations, the majority of Snort signatures were used in our analysis and we were able to obtain interesting results (see Section 6.6).

6.6 Case Study

In this section, we describe the current results obtained using our IDS-SSA tool on 12 Snort signature databases for the signature equality and inclusion analysis. We used IDS-SSA on nearly all Snort signature databases from Snort 1.8.6 to Snort 2.4.5. We observed similar results across Snort versions (i.e., 1.8.7, 1.9.0, 1.9.1, 2.0.0, 2.1.0, 2.2.0, 2.3.0, 2.3.1, 2.3.2 and 2.3.3) and therefore only report on versions 1.8.6 and 2.4.5. We selected Snort 1.8.6 because its log limit is one and cannot be changed. In particular, Snort 1.8.6 only provides an IDS event for the first signature that is triggered by a packet. Thus, this could cause several potential evasion attacks. We know that Snort developers were aware that a log limit of one could prevent logging of important IDS events. This is why newer Snort engines such as Snort 2.4.5 can log multiple IDS events on a single packet. However, we use Snort 1.8.6 because even if the problem of the log limit of one is known, we are not aware of any tool or study that was able to do a qualitative and quantitive analysis of the signature overlapping problem of this signature database such as the one presented in this chapter.

More recent versions of the engine and signatures are available (e.g., the 2.8.5 engine was released in 2010) but their signatures, though currently very similar to the ones of the above-mentioned versions, cannot be shown to any unregistered user, which would prevent discussing results in detail in this chapter. Thus, we selected versions 1.8.6 and 2.4.5 because they are representative of versions for which we can discuss signatures in details.

6.6.1 Design of the Experiment

The design of the experiment for each signature database version was the same. To describe this design, we follow the process of IDS-SSA. During our analysis, we used the default Snort configuration and the default signature database of Snort with one minor exception (see below).

The Snort signature database for version 2.4.5 (resp. 1.8.6) has a total of 3576 (resp. 1266) signatures. Some of these signatures are grouped into signature description files, according to their purpose, to facilitate their use (e.g., the signatures that monitor inappropriate content to web sites are grouped in one description file). One can then decide to not use one file or another thereby discarding a set of signatures at once (instead of selecting signatures individually). These signature files are not all used when Snort is executed with its default configuration. However, we made sure that all available signatures (files) were included in our analysis.

In these files, signatures can be tagged as *commented out* (e.g., they are deemed to not function properly), or *deleted* (e.g., a signature that will disappear from the file in a future release). We did not consider the 385 (resp. 25) commented and deleted signatures for Snort 2.4.5 (resp. Snort 1.8.6).

The tool limitations discussed in Section 6.5.2 further reduce the number of signatures we could consider: 930 (resp. 10) additional signatures are not considered for Snort 2.4.5 (resp. Snort 1.8.6). In the end, we used 2261 signatures for Snort 2.4.5 and 1231 signatures for Snort 1.8.6 (i.e., 63% and 97% of their untagged signatures, respectively).

6.6.2 Results

Table 6.3 summarizes the current results and presents the number of pairs of equal signatures (i.e., $S_i^d = S_j^d$), the number of inclusion chains of length two (i.e., $S_i^d \subset S_j^d$), the number of inclusion chains of length three (i.e., $S_i^d \subset S_j^d \subset S_k^d$) and the number of chains length of four (i.e., $S_i^d \subset S_j^d \subset S_k^d \subset S_l^d$). We also looked for inclusion chains of length five (i.e., $S_i^d \subset S_j^d \subset S_k^d \subset S_l^d \subset S_m^d$).

For this case study, we did not find any inclusion chain strictly longer than four. Note that when counting ICs, sub-chains made of signatures involved in an IC of length n count for ICs of length smaller than n. For instance, the table shows there exists an IC of length 4, suggesting there exist four signatures S_1^d, S_2^d, S_3^d, S_4^d such that $S_1^d \subset S_2^d \subset S_3^d \subset S_4^d$; these signatures (e.g., S_1^d, S_2^d, S_3^d with $S_1^d \subset S_2^d \subset S_3^d$) count when identifying ICs of length two and three.

Snort	Equal	Incl. Ch. of Two	Incl. Ch. of Three	Incl. Ch. of Four
1.8.6	4	306	24	0
2.4.5	4	264	3	1

Table 6.3: Signature Overlap Analysis Results: Signature Inclusions and Equalities

These results suggest redundancies between signatures (equal signatures). The inclusions also suggest numerous situations (in combination with intersections, see Section 6.6.2.2) that can be exploited by an attacker (e.g., IDS evasion). To the best of our knowledge, this is the first time such results, obtained from a systematic study, are reported.

In the case of Snort 1.8.6, the log limit is one and cannot be changed. Thus, every inclusion chain of two or more (and potentially the pairs of equal signatures) could potentially be subject to an evasion attack.

One could claim that Snort developers have such a tool to analyze the newer signature database such as Snort 2.4.5, since there is only one inclusion chain bigger than the default log limit of three. We could question this assumption based on the fact that their tool (if it exists) was unable to identify the pairs of equal signatures (one can not assume they left these signatures on purpose in the signature database) since there are four pairs of equal signatures in the signature database for Snort 2.4.5. In this case, identifying pairs of equal signatures is the easiest solution to implement for the three sub-problems (i.e., equality, inclusion and intersection) of the signature overlapping problem. Moreover, we looked for how long these pairs of equality have been unnoticed (stayed in the Snort signature database). Some stayed in the signature database in several consecutive versions. For example, $S_{272} = S_{273}$ and this pair was in Snort 2.3.1 to 2.4.5 (March 2005 to at least July 2005), $S_{893} = S_{1722}$ and this pair was in Snort 1.8.7 to 2.4.5 (July 2002 to at least July 2005); and

$S_{841} = S_{1656}$ and this pair was in Snort 2.3.0 to 2.4.5 (January 2005 to at least July 2005). However the fourth pair, $S_{2196} = S_{2195}$, was introduced in the Snort 2.1.0 signature database (released in December 2003) and is still included in the latest Snort database we have access to (released on the 17^{th} of February, 2010). Here is a more detailed analysis of these results.

6.6.2.1 Equality

In the case of Snort 1.8.6 and 2.4.5, the equalities between signatures are due to the default configuration of Snort, to duplications, and to human errors such as cut and paste errors (i.e., copying a signature similar to the one you want to specify, pasting it and forgetting to modify parts of its specification). All of the equality groups identified contain two signatures. Thus, this situation could be only problematic for Snort 1.8.6 (i.e., log limit of one), but, fortunately, in the case of Snort 1.8.6 each pair of equal signatures contain non-orthogonal signatures (i.e., they refer to the same attack/vulnerability).

An example of typical duplication is S_{475} and S_{455} in Snort 1.8.6. These signatures have the exact same specification with simply a different signature number and revision number. The other three instances of equal signatures for Snort 1.8.6 are due to the fact that we used the default configuration of Snort, specifically, to the fact that by default, variables `$EXTERNAL_NET` and `$HOME_NET` are both equal to **any** (i.e., any possible IP address). Then, the two signatures that only differ on the direction of the monitored communication (source and target machines specified with those variables) are considered equal.

In the case of Snort 2.4.5, one of the equal groups is likely due to human error. S_{272} and S_{273} are part of the Snort signature database since at least Snort 1.8.6. The difference between these signatures is one predicate in the specification of the content of the packet payload. However, this predicate has been removed in Snort 2.3.1, making them equal in Snort 2.4.5. A third case of equality is due, we believe, to a cut and paste error. Figure 6.4 presents these two signatures (Snort 2.4.5 signature database).

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (
msg:"WEB-CGI alert.cgi access"; flow:to_server,established;
uricontent:"/alert.cgi"; nocase;
reference:bugtraq,4211; reference:bugtraq,4579; reference:cve,2002-0346;
reference:nessus,11748; classtype:web-application-activity; sid:2195; rev:6;)

alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (
msg:"WEB-CGI catgy.cgi access"; flow:to_server,established;
uricontent:"/alert.cgi"; nocase;
reference:bugtraq,3714; reference:bugtraq,4579; reference:cve,2001-1212;
reference:nessus,11748; classtype:web-application-activity; sid:2196; rev:6;)
```

Figure 6.4: Signature 2195 and 2196 from Snort 2.4.5

As we observe, only the message has been changed in signature 2196 and we believe the person who specified this forgot to change the specification. We believe this is a human error because both signatures refer to different vulnerability identifiers from SecurityFocus (bugtraq) and CVE that are coherent with the message describing the attack. We identified that this error was introduced in the Snort 2.1.0 signature database and is still included in the latest Snort database we have access to (released on the 17^{th} of February, 2010). The other two cases of equality in Snort 2.4.5 are duplications.

6.6.2.2 Inclusion

In the case of Snort 1.8.6, we know that Snort developers were aware that a log limit of one could prevent logging IDS events. However, with the results provided by our tool, we are able to visualize (e.g., Figure 6.5) the extent of the signature inclusion problem in Snort 1.8.6 (and in any other Snort versions).

Figure 6.5 (derived from our results) partially shows the extent of the signature inclusion problem for a group of signatures in Snort 1.8.6: e.g., S_{1400} includes S_{1073} and is included in S_{1287}. Note that S_{999} is included in four other signatures! Since Snort 1.8.6 has a log limit of one, these inclusions lead to the risks of missing attacks, i.e., Snort generating FN_D. For example, a signature (e.g., the including signature) can prevent another signature (i.e., the included signature) from being

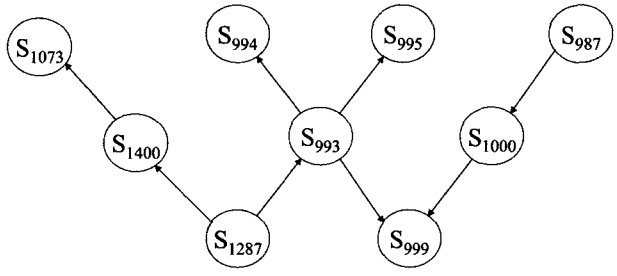

Figure 6.5: Inclusion Chain in Snort 1.8.6

triggered when the including signature is checked first by Snort. Earlier experiments we conducted in Chapter 5 indicate this is often the case.

Snort 2.4.5 (and Snort 2.8.5, the last version currently available, released in 2010), has a *log limit* of three. Three is the default configuration value, but this log limit can be changed (to a smaller or bigger value) in the configuration. In Snort 2.4.5, there is one inclusion chain of 3 and one inclusion chain of 4 (i.e., bigger than the default log limit). Fortunately, in both cases, the signatures in the inclusion chain (of 3 and 4) are non-orthogonal signatures.

Although these inclusion chains contain non-orthogonal signatures, it does not mean that it is not possible to use these inclusion chains in combination with another signature that is orthogonal (i.e., that refers to a different attack/vulnerability) to them to cause a FN_D (evasion attack). In fact, this specifies that we only need to identify one signature that intersects with the first signature of the inclusion chain (i.e., S_i^d in $S_i^d \subset S_j^d \subset S_k^d$) and then generate the proper sequence of packets within this intersection to show that an attacker can exploit this situation to make an evasion attack. Consequently, we can show that signature overlapping problems exist even in newer Snort signature databases such as Snort 2.4.5.

To mimic what an attacker, aware of the overlapping problem, would do, we first selected the following intersection chain identified in Snort 2.4.5, where: $S_{1537} \subset S_{1455} \subset S_{882}$. Figure 6.6 presents these three signatures (Snort 2.4.5 signature database).

Second, we identified a signature in the Snort 2.4.5 signature database that intersects with S_{1537} (thus, intersects with S_{1455} and S_{882}). If we look at S_{1002} (Figure 6.7), also from Snort

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS
(msg:"WEB-CGI calendar_admin.pl access";
flow:to_server,established; uricontent:"/calendar_admin.pl";
reference:cve,2000-0432; classtype:web-application-attack; sid:1537; rev:6;)

alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS
(msg:"WEB-CGI calendar.pl access";
flow:to_server,established; uricontent:"calendar"; nocase;
pcre:"/calendar(|[-_]admin)\.pl/Ui"; reference:bugtraq,1215;
reference:cve,2000-0432; classtype:web-application-attack; sid:1455; rev:7;)

alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS
(msg:"WEB-CGI calendar access";
flow:to_server,established; uricontent:"/calendar"; nocase;
classtype:attempted-recon; sid:882; rev:5;)
```

Figure 6.6: Snort signature 1537, 1455 and 882 from Snort 2.4.5

2.4.5, we can see that it intersects with S_{1537} since the only requirement for them to be able to intersect (all other plug-ins are the same) is that the attack packet payload should contain the string `calendar_admin.pl` and the string `cmd.exe`.

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS
(msg:"WEB-IIS cmd.exe access";
flow:to_server,established; uricontent:"cmd.exe"; nocase;
classtype:web-application-attack; sid:1002; rev:7;)
```

Figure 6.7: Snort signature 1002 from Snort 2.4.5

If we provide Snort 2.4.5 (using its default configuration) with a traffic trace that contains the proper TCP handshake on port 80 followed by a packet (also in the same TCP session) that contains `/scripts/..%c0%af../winnt/system32/cmd.exe?/c+dir+c:\+c:\calendar_admin.pl\`, it only provides IDS events for S_{1537}, S_{1455} and S_{882}, and no IDS event for S_{1002} (log limit of three). In this particular case, Snort 2.4.5 does not log specifically the IDS event for S_{1002} because, as described in Section 6.3, in the default Snort configuration, the IDS events are ordered in the event queue using the length of the string in their `content` plug-ins (i.e., the IDS event corresponding to a signature with a longer string in their `content` plug-ins is ordered first). Thus, since S_{1002} has the shortest string in its `content` plug-in, its corresponding IDS event is not logged.

One question remains: Does this constitute a real attack or is it a synthetic attack? In this case, it is a real attack. We carefully selected the string `/scripts/..%c0%af../winnt/system32/cmd.exe?/c+dir+c:\+c:\calendar_admin.pl\` since it can be used to exploit vulnerability BID 1806 against Microsoft IIS Server version 4.0 and 5.0 on Windows 2000.

This vulnerability allows an attacker to execute commands on the target system. Here the string `/scripts/..%c0%af../winnt/system32/cmd.exe?/c+dir+c:\+c:\calendar_admin.pl\` allows the attacker to use the command `cmd` to execute the command `dir` on the folder `c:\` and on the folder `c:\calendar_admin.pl\`, which is used as a decoy to hide the `cmd` command.

We actually tested this attack against a vulnerable Windows 2000: it works; and Snort did not log the IDS event for S_{1002}, which is the intent of the attack. We acknowledge that this vulnerability is old and that it is probably not existent today. However, this example is easy to understand and clearly shows how someone can take advantage of overlapping signatures in an IDS signature database, even using newer Snort version such as Snort 2.4.5, to hide its real intention (i.e., evasion attack).

We argue that we could easy identify other signatures that intersect with S_{1537} that will prevent the network administrator from getting the IDS events reflecting the real intention of the attacker.

6.6.3 Discussion

The equality between signatures identified in the result section has limited impacts on the accuracy of the IDS. Such equalities can lead to a misunderstanding of the situation (i.e., direction of the attack) by the network administrator or could mislead the network administrator (e.g., which IDS event really corresponds to the attack?) when the signatures are orthogonal, which is the case for Snort 2.4.5 (Figure 6.4).

Signature inclusions, such as those identified in the result section and the one presented in Section 6.2, have a more important impact when the inclusion chain contains orthogonal signatures.

In any case, these equalities and inclusions make it easier for an attacker to exploit the log limit.

In the case of Snort 2.4.5, the attacker only has to identify one orthogonal signature with the first signature of an inclusion chain of three to be able to prevent the network administrator from getting the IDS events targeted by the attack. Furthermore, we demonstrate that we were able to use an inclusion chain of three to create a real and successful attack against a real target system.

As a result, we argue that the signature intersection analysis, in combination with the results presented in this chapter, could easily reveal exploitable situations in the newer versions of Snort. Furthermore the intersection analysis would help us to answer this question: Which value of the (log limit) configuration parameter should be used? In the case of Snort, given its current design, with one configuration value for all the signatures, the only safe solution would be to select the maximum (to be indentified) number of intersecting signatures for the database it is using. However, as we demonstrated in Section 6.3, increasing the log limit only leads to more effective squealing attacks and performance issues.

Consequently, we believe that more work needs to be done in signature overlap analysis to quantify and address this problem, which will be part of our future work. An alternative could be to revisit the signature database that shows inclusions, equalities and intersections to limit their number.

6.7 Conclusion

In this chapter, we presented an approach to systematically verify the signature overlapping problem of an IDS to quantify the signature overlapping problem, an area in which very little systematic theoretical and empirical work exists. We defined the signature overlapping problem as the equality, inclusion or intersection of the sets of packets that trigger different signatures in an IDS signature database. We strongly believe that defining a systematic approach like ours, to complement the other IDS verification and validation approaches, is a necessary first step to improve IDS signature specifications.

Our approach consists in transforming IDS signatures into sets and finite state automata. This allows us to use set theory and automaton theory operators to systematically identify signature overlaps. We built a tool infrastructure to support our approach and, as a first step, analyzed equalities and inclusions of signatures to systematically check whether several IDS signature databases contain such overlaps.

We showed that our approach and tool would likely apply, with minor modifications, to a large family of signature-based network IDSs. In our case study, we restricted our analysis to one widely-used IDS, specifically Snort (and evaluated different versions of Snort).

We were able to automatically analyze several Snort signature database versions between versions 1.8.6 and 2.4.5 (though we only report on these two versions since they are representative of all our observations). We demonstrated that high log limit could lead to more effective squealing attacks and performance issues. We showed that these inclusion chains could prevent Snort 1.8.6 from detecting attacks. We also showed that in Snort 2.4.5, these inclusion chains can be used with an intersecting orthogonal signature to derive a real successful attack that is not detected by the IDS. To the best of our knowledge, this is the first time such an attack has been documented, and also thanks to our systematic approach, this is the first time such quantitative analysis is reported in the literature. Moreover, using our systematic approach, we were able to perform both quantitative and qualitative analysis of this problem in Snort, thus providing evidence of the effectiveness of our automaton and set theory approach. We showed that our qualitative analysis could lead to improvements of the Snort signature database.

Future work will focus on the signature intersection analysis, which we showed to be very computationally intensive. Moreover, we will explore the idea of using the IDS-SSA to compare the space covered by signatures from different IDS vendors that identify the same attacks. For instance, we propose to explore the possibility of comparing signatures from Snort and Bro.

Our proposed testing models presented in Chapters 4 and 5, and this verification method allowed us to identify different key accuracy problems related to the IDS engine and also to the IDS signature

database. This provides us with a great body of knowledge to tackle the IDS specification and IDS implementation part of our research methodology (Figure 1.1). We decided to focus our work on the verification problems related to the IDS signatures. Consequently, in Chapter 7 we propose an approach to automatically generate IDS signatures that address the verification problems identified in Chapter 4, related to missing and incorrect verification signatures.

Chapter 7

Automatic Generation of IDS Verification Rules

In Chapter 4, we identified that the verification problem was insufficiently addressed by the two IDSs we tested. (1) We showed that there were signatures missing in the signature database of the tested IDSs to verify the success or failure of an attack. In this case, the tested IDSs provided the same IDS events whether the attack was successful or not. As a result, it is difficult for network administrators to distinguish between IDS events related to a compromised target system and other IDS events. (2) We also showed that some incorrect verification signatures led to FN_V. We propose in this chapter an automated approach to generate the IDS specification for the verification problem that addresses (1) and (2). Before describing this approach, we present an overview of the different techniques used to verify attacks.

Five different complementary approaches are now recognized to address the verification problem. The first approach is to use the network configuration in context with the attack. This approach is based on the hypothesis that an attack can only be successful if it is able to reach the target system. For example, the time to live (TTL) of an IP packet can be used to verify attacks [19]: When the

TTL of an attack packet is too low for the packet to reach the target system, the attack is impossible, and the attack can be ignored or an IDS event can be raised by the IDS with the proper context (i.e., the attack cannot reach its destination). The second approach uses the protocol specification to identify whether the attack can be successful or not. This approach is based on the hypothesis that protocol implementations always behave according to their specification and that an attack is only possible when it is sent at the proper state of the communication protocol. For example, for a TCP (attack) packet to be accepted by a target system, it has to be part of an open TCP session. The stateful improvement of IDSs (e.g., [18, 19]) is a good example of a protocol specification approach that seeks to address the verification problem. The third approach is to rely on the knowledge of the target system products (e.g., name and version) and vulnerability databases (e.g., SecurityFocus) to determine whether the target system is indeed vulnerable or not to an attack (i.e., whether the attack can be successful or not) [60]. The fourth approach is to rely on a vulnerability assessment of the target system using a dedicated tool (e.g., Nessus) to verify attacks more accurately than by using the target system products [8]. The fifth approach relies on the target system reaction to an attack and is used in well-known IDSs (e.g., Snort [18], Bro [19] and Snort UC Davis [10]). This approach consists in looking at messages (e.g., an error message) or behaviours (e.g., the target system does not respond) before and after the attack to decide whether or not the attack has been successful. However, this approach, as implemented in those IDSs, has been shown to only partially address the verification problem (Chapter 4).

In this chapter, we use the target reaction approach to generate verification signatures and we propose to automate their generation to address the missing and incorrect verification signature problems identified in Chapter 4. We choose the target reaction approach because it is the one used in the IDSs we tested in Chapter 4 (i.e., Snort and Bro).

In this chapter, we use the term rule (i.e., IDS rule) instead of signature. This emphasizes the fact that the IDS signatures used in our study are logical expressions and this also facilitates the explanations related to the automatic generation of IDS signatures.

To do this, we employ a data mining technique and real attack scenarios that allow us to automatically generate IDS verification rules that predict the success or failure of attacks. We use a classification algorithm to generate these rules, since based on our related work (Section 3.2), classification algorithms seem to be a more appropriate type of data mining algorithms to use to generate verification rules. Then, these rules are used with a modified version of Snort and the results are compared with Bro, Snort and Snort UC Davis. The results indicate that our approach (1) can be used to automatically learn IDS verification rules, (2) can speed up the process of generating these rules, and (3) reduces human errors that could result in incorrect rules (e.g., as identified for Bro in Chapter 4). The contribution of this chapter is four-fold:

- We propose an approach that uses a standard data mining algorithm to automatically generate IDS verification rules, thus reducing the effort and errors related to the manual generation of such rules.
- We illustrate how this approach can be applied with a widely-used IDS.
- We present a tool that implements our approach.
- We report on a case study that shows that our verification rules improve the accuracy of Snort and Bro.

The rest of this chapter is structured as follows. Section 7.1 provides some background information on data mining technologies. Section 7.2 presents our approach. Section 7.3 describes the IDS we used to test and evaluate the accuracy of our IDS verification rules. Section 7.4 describes a case study. Conclusions are drawn in Section 7.5.

7.1 Background on Data Mining Technology

This section introduces data mining terminology to facilitate the understanding of our approach. For example (from [28]), consider the (over) simplified problem illustrated in Figure 7.1, Figure 7.2 and

Figure 7.3 where one wants to identify under which conditions we can play outside given the values of three characteristics, namely Outlook, Temperature and Windy. The input to the data mining algorithm is a table (Figure 7.1), called the training data set, where rows and columns are referred to as instances and attributes, respectively. Each row is an instance of values of the attributes. Rows can be used by a data mining algorithm to predict the values of an attribute (e.g., Play) using the values of the other attributes. In the data mining literature, this is called a *classification problem* [28]. The input of Figure 7.1 contains 14 instances, characterized by four attributes (i.e., Play, Outlook, Temperature and Windy).

Play	**Outlook**	**Temperature**	**Windy**
Yes	Sunny	Hot	No
Yes	Sunny	Hot	Yes
Yes	Overcast	Hot	No
No	Rainy	Mild	No
No	Rainy	Cold	No
No	Rainy	Hot	Yes
Yes	Overcast	Hot	Yes
Yes	Sunny	Mild	Yes
Yes	Sunny	Cold	Yes
No	Rainy	Mild	Yes
Yes	Sunny	Mild	No
No	Overcast	Mild	Yes
Yes	Overcast	Hot	Yes
No	Rainy	Cold	Yes

Figure 7.1: Training Data Set

```
if Outlook = Sunny then
    Yes
else if Temperature = Hot then
    if Outlook = Overcast then
        Yes
    else
        No
else
        No
```

Figure 7.2: Rule

	Play	Not Play
Play	8	0
Not Play	0	6

Figure 7.3: Confusion Matrix

The output of a data mining algorithm is called a *prediction model* and takes the form of a

rule (for a classification algorithm) involving attribute values that predict one attribute's values. Figure 7.2 shows one such prediction model for the training data set of Figure 7.1 when we use the C4.5 [61] data mining algorithm (this algorithm is further discussed in Section 7.2.3.1). The rule has three predicates: `Outlook = Sunny`, `Temperature = Hot` and `Outlook = Overcast`. This rule is used to specify whether we can go play outside. This rule specifies that we can play outside when it is sunny or when it is overcast and hot, otherwise we cannot go play outside. Note that in general, the set of predicates that appear in a rule is a subset of the set of attributes or attribute values since not all attribute (values) in the input table are necessarily used to predict the predicted attribute values.

To evaluate the accuracy of the generated rules, it is common practice to use a confusion matrix [28], such as the one shown in Figure 7.3: a confusion matrix is in general provided along with the rule by the data mining algorithm. A confusion matrix shows the number of instances for which the actual values of the predicted attribute are the rows (i.e., the value from the training data set) and the predicted values of that attribute are the columns (i.e., the values when using the rule). Figure 7.3 shows that all the instances are correctly classified, i.e., the predicted value equals the value in the training data set.

When the data set is large enough, or two different sets are available, it is common to build a prediction model using one part of the data and to verify its accuracy on the other part of the data: thus the wording of training data set and testing data set. When the data set is not large, a standard technique called n-folds cross-validation [28] is used to randomly create training and testing data sets from the data set to assess and evaluate the prediction models. It is common practice to use this technique with n=10 [28], which is what we do in this chapter.

7.2 A Data Mining Approach for the Automatic Generation of IDS Verification Rules

Our IDS Verification Rule Generator (IDS-VRG) tool was developed with two requirements in mind: (req. 1) using the target reaction approach to automatically improve the accuracy of current IDS verification rules and (req. 2) to identify new IDS verification rules. Thus, in this chapter, the prediction models are also referred to as IDS verification rules to simplify the description. Our approach is iterative, as illustrated in Figure 7.4. First, the Capture System (step 1) captures and/or generates documented traffic traces using various sources (e.g., malware execution, vulnerability exploitation program execution, corporate network traffic) and stores them in a *documented data set*. Labels indicate whether traffic traces contain successful attacks, failed attacks, or are normal traffic traces. Note that normal traffic traces are not needed when actually verifying whether an attack has succeeded. However, as discussed in Section 7.2.3.2, this is required when generating rules to be used for verification. The *IDS Verification Rule Generator* (step 2) then executes the data mining algorithm to generate rules, reports on the accuracy of the generated rules, and requests the user to check (step 3) whether the generated rules are semantically sound and syntactically complete (i.e., the verification rules make sense for use in an IDS to verify the success of a specific attack) (Section 7.2.4). An IDS can then be used to compare existing rules (referred to as "Current IDS Rules" in Figure 7.4) with the generated rules by using both sets of rules on attack traffic (i.e., the documented data set). If the generated rules are deemed better than existing rules, they can be used instead. The procedure is iterative since, when new attack traces are available, new rules can be generated by the IDS Verification Rule Generator and compared to existing ones. This iterative and automatic approach of the IDS-VRG addresses the dynamic nature of the verification problem. Note that the IDS-VRG is the continuation of the VEP approach presented in Figure 4.1 where the step 1 to 3 of the VEP approach are used to provide network traffic (step 1) to the IDS-VRG. This is in line with our research methodology presented in Figure 1.1 of Section 1.2 where IDS testing

and evaluation results (i.e., Chapter 4) are used as requirements to improve IDS signatures (i.e., this chapter).

The selection of the IDS that we used is discussed in Section 7.2.1. The initial selection of network traffic (step 1) is discussed in Section 7.2.2. The tasks performed by the IDS Verification Rule Generator (step 2) are discussed in Section 7.2.3. The criteria used to evaluate the IDS verification rules (step 3) are discussed in Section 7.2.4.

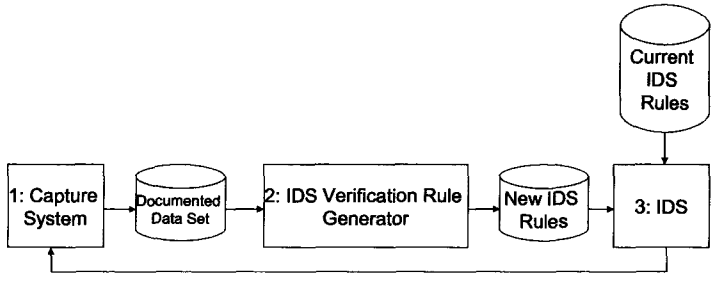

Figure 7.4: Automatic Generation of IDS Verification Rules

7.2.1 Selection of an IDS

Our approach requires an IDS rule database (i.e., the Current IDS Rules in the first iteration of Figure 7.4) and an IDS to be tested (step 3 in Figure 7.4). We selected the Snort rules since Snort is one of the most widely used and maintained open source database. We also used it for two other reasons. First, the work presented in Chapter 4 has shown that work still needs to be done to improve the Snort verification accuracy. Second, another widely used and maintained IDS, namely Bro, and a proposed solution to the verification problem based on Snort, namely Snort UC Davis [10], generate their verification rules using the Snort rules. It is important to mention that these approaches do not use data mining techniques to generate their verification rules. Instead, verification aspects are manually added to the detection rules. Thus, for comparison purposes, the Snort rules are good starting points to assess our approach, and to identify whether it is feasible to automatically generate equivalent or better verification rules than the ones contained in the Bro

and Snort UC Davis rule databases.

We selected the Snort 2.3.0 rule database (January 2005) as the initial set of IDS rules because it is the one used by Snort UC Davis. Bro is independent of the Snort rule database version it uses since any Snort rule database can be automatically converted into Bro rules using *s2b* (Section 2.4).

7.2.2 Selection of a Training Data Set

In order to obtain meaningful and accurate verification rules using a data mining technique, we cannot simply use the traffic traces as inputs. Our experience with data mining techniques is that using raw data typically leads to meaningless and inaccurate rules because the data mining algorithm cannot learn what input or output properties are potentially of interest, but only which ones matter once they are defined. In other words, without some additional guidance, the data mining algorithm is unlikely to find the precise conditions under which an attack succeeds. This guidance, in our context, comes in the form of attributes that characterize traffic traces. We first specify those attributes (Section 7.2.2.1), and then discuss how we generated the training data set using those attributes (Section 7.2.2.2).

7.2.2.1 Attributes

The input to the data mining algorithms we used is a table where the rows represent *instances*, i.e., traffic traces, and the columns represent *attributes* of those instances, i.e., characteristics of the traces. We used three groups of attributes: the attack result, the rule(s) triggered by the attack traces, and the target reactions identified in the traffic traces. We discuss these attributes below.

Result is the predicted value of the verification rules. The possible values are `success`, `failure`, and `normal`.

The *rule attributes* are the Snort rules (*Snort(i):m* where i is the rule number and m the message associated with this rule) that are triggered on each traffic trace. The possible values are `true` and `false`.

The *target reaction attributes* are indicators of failure or success of attacks. We build upon the approaches of Bro [9] and Snort UC Davis [10], where standard protocol messages (e.g., HTTP 200 OK) are monitored after an attack is detected to infer the success or failure of the attack. We also added special target reaction indicators to model when no reaction is observed. We used an inactivity timeout of 30 seconds, which is the default value for Snort and Bro, to identify the absence of a reaction. We also added our own reaction *indicators*. We identified that TCP/UDP port status and the host status of the target system can also be indicators of success or failure of an attack. These target reaction *indicator* attributes are referred to as:

- $StandardMessage(protocol, i)$: tells whether or not a standard message i from a *protocol* (e.g., HTTP, FTP, SMTP, IMAP or POP) is in the traffic traces. The values are `true` and `false`.
- $HostPortState(host, port, i, j)$: tells whether or not a *port* of a *host* (i.e., target, attacker) has changed from state i (i.e., closed, open) to state j (i.e., closed, open). The values are `true` and `false`
- $HostState(host, i)$: tells whether or not a *host* (i.e., target, attacker) is in a particular state i (i.e., up, down). The values are `true` and `false`

7.2.2.2 Training Data Set Generation

We selected the documented VEP traffic traces presented in Chapter 4 as our training data set. An instance is generated for each traffic trace from the VEP data set. The *Result* attribute (either `success` or `failure`) is provided by the `Success` field of the documented data set (Section 4.2.3.4). The *Snort(i):m* attributes are generated for each traffic trace using Snort 2.3.0 on the VEP data set with the IDSEF (Section 4.2.4). For the target reaction indicators, we created another Snort rule database since the default Snort rule database does not contains rules related to all the target reaction indicators. We used Snort 2.3.0 with this new rule database on the VEP data set with the IDSEF to generate the target reaction indicator attributes.

We are aware that recent techniques can be used to circumvent the target reaction approach (e.g., [62]), whereby the attacker ensures that when an attack is successful, the target reacts as if the attack had failed. The VEP data set we used does not contain traffic traces using these techniques.

In the context of the verification problem, normal traffic is not required to assess IDSs (Section 1.1). However, to generate accurate verification rules we need normal traffic (Section 7.2.3.2). We added simulated instances to the training data sets for normal traffic instead of using real normal traffic because of the simplicity of the simulation solution: we do not need packet traces and execute them, we can simply add instances (rows) to the training data (table) and simulate the values of the attributes (columns). For this approach to be useful, we have to consider two issues: how to simulate normal traffic instances and how many simulated instances are needed in the training data sets. To simulate normal traffic instances, we first create one instance by looking, one by one, at every attribute contained in the training data sets: if the attribute can characterize normal traffic, we assigned it the `true` value and otherwise the `false` value. In doing so, we assumed that no attack detection rule from Snort is triggered on normal traffic. Thus, every *Snort(i):m* attribute that corresponds to a rule that detects an attack has a value of `false` for the simulated instance. We acknowledge this is a strong assumption. However, we are able to generate accurate verification rules using this hypothesis as long as the detection rule is accurate (Section 7.4.2).

For instance, suppose a training data set with the three attributes: *Results*, *Snort(971):WEB-IIS ISAPI .printer access* (i.e., an attack attempt) and *StandardMessage(HTTP,200)* (i.e., a target reaction indicator). The corresponding simulated instance is `normal` (value for attribute *Result*), `false` (no attack attempt in normal traffic), `true` (standard message) since the .printer attack attempt cannot characterize normal traffic (based on our hypothesis) and it is possible to observe an HTTP 200 OK message in normal traffic.

With regard to the number of simulated instances used in the training data set, our heuristics was motivated by the necessity to ensure that the data mining algorithm still provides a verification rule (i.e., a rule that distinguishes between successful and failed attack), which forces us to be greedy on

the number of simulated instances we add. As a heuristic, to balance the `normal` instances with the `success` and `failure` instances, we added $n/2 + 1$ (rounded) copies of the simulated normal traffic instances described earlier in the data set, where n is the smallest number of instances between those that have the `success` and `failure` values for their result attribute. For instance, if we have 91 instances with the `failure` value and 9 with the `success` value then 5 simulated normal traffic instances are added.

Note that an attack only attempts to exploit one specific vulnerability and that only a specific group of rules are used in IDSs to identify an attack. Thus, a verification rule has to be generated for each vulnerability to reflect the reality of the relation between IDS detection rules, attacks and vulnerabilities. Consequently, we grouped the traffic traces (i.e., instances) into multiple training data sets: one per vulnerability. We only used the vulnerabilities for which Snort has corresponding rules and for which the VEP data set has successful and failed attack attempts. Indeed, the data mining algorithm needs a diverse set of instances where all the possible attribute values (in particular values for the attribute being predicated) are used to produce accurate verification rules. In this case, 16 training data sets were created to generate verification rules using 16 vulnerabilities of the VEP data set (see Section 7.4.2 for the list of the 16 vulnerabilities we used).

7.2.3 IDS Verification Rule Generator

In this section, we discuss our choice of data mining algorithm(s) (Section 7.2.3.1) and the need for normal traffic traces (Section 7.2.3.2). We then discuss evaluation criteria for generated IDS verification rules (Section 7.2.4).

7.2.3.1 Generation of IDS Verification Rules

Once the input data we have just described is available, classification algorithms or decision tree algorithms can be considered to solve our classification problem [28].

To solve a classification problem, some algorithms, such as C4.5 [61], partition the training data

set in a stepwise manner using complex algorithms and heuristics to avoid over-fitting the data with the goal of generating models that are as simple as possible. Others, like Ripper [44], are so-called covering algorithms that generate rules in a stepwise manner, removing observations that are *covered* by the rule at each step so that the next step works on a reduced set of observations. With covering algorithms, predicates in a rule are interdependent in the sense that they form a *decision list* where predicates are supposed to be applicable in the order in which they are generated. One important issue is that the order in which predicates have to be evaluated when evaluating the rule may not be the order in which the information used to evaluate the predicates is available, i.e., the order in which packets are observed. We therefore decided to use C4.5 as our classification algorithm.

However, the usage of classification and decision tree algorithms such as C4.5 prevents IDS-VRG from automatically generating *IDS-ready* verification rules. For example, these algorithms do not generate rules with temporal constraints between the predicates of the verification rules. In the case of IDS verification rules, predicates represent information arriving at different periods of time: for instance, information about an attack and information about the target response. The order in which the predicates should be monitored by an IDS is not specified in the verification rules generated by classification or decision tree algorithms such as C4.5. While the temporal constraints are not required by the IDS to verify attacks, without them, the verification rules would not be as accurate as they could be (i.e., would raise FP_V). Therefore, the output of those algorithms has to be manually post-processed before including them in an IDS rule database. A network administrator would have enough knowledge, in terms of network administration, communication protocols and vulnerabilities, to find the order in which the predicates composing a rule have to be evaluated. In our experiments, every verification rule that was generated using the classification and decision tree algorithms was manually modified to reflect the temporal constraints intended among the predicates contained in the verification rules. We know that this mitigates the automation process of generating verification rules, but we will show in Section 7.4 that our data mining approach, although not completely automated, could prevent human errors. Moreover, we plan to address the missing

temporal constraints in our future work to automate further our approach.

7.2.3.2 Normal Traffic

Normal traffic is not required by definition for assessing the verification problem since the verification problem is to distinguish successful attacks from failed attacks and not distinguishing attacks from normal traffic. However, to complete the specification (learning) of the verification rules with proper attack detection, the IDS behaviour on normal is also required since to verify an attack an IDS first needs to detect it.

Consider for instance the training data set in Figure 7.5. Figure 7.6(a) and Figure 7.6(b) show verification rules generated with the C4.5 data mining algorithm [61]. For Figure 7.6(a), normal traffic is not used (i.e., the last row of the training data set is not used), whereas for Figure 7.6(b) normal traffic is used. The verification rule in Figure 7.6(a) has one predicate: `HTTP 200 OK = true`, and specifies that when the IDS sees a HTTP 200 OK message, this indicates a successful attack. Recall from Section 7.1 that a prediction model does not necessarily involve all the attributes, when some attributes do not help classify instances. This verification rule is incorrect because the `HTTP 200 OK = true` can also hold for normal traffic. The verification rule in Figure 7.6(b) is what a network security expert would expect as a verification rule where the two predicates `HTTP Attack = true` and `HTTP 200 OK = true` are used together to respectively detect the attack and distinguish between a successful and failed attack attempt.

Result	HTTP Attack	HTTP 200 OK
success	true	true
failure	true	false
normal	false	true

Figure 7.5: Training Data Set

7.2.4 Evaluation of IDS Verification Rules

We evaluate the IDS verification rules generated by C4.5 using two criteria: accuracy and usability. Accuracy relates to the TP_V and TN_V rates (Section 2.5): to evaluate this we used a 10-fold cross-

```
if HTTP 200 OK = true then
    success
else
    failure
```

(a) Rule without Normal Instances

```
if HTTP Attack = true then
    if HTTP 200 OK = true then
        success
    else
        failure
else
    normal
```

(b) Rule with Normal Instances

Figure 7.6: Rule Examples

validation approach. Usability relates to whether rules are semantically sound and syntactically complete (i.e., the verification rules make sense for a network security expert to use them in an IDS to verify a specific attack), as discussed below.

We say that a verification rule is *usable* if it meets the two following criteria: (criterion 1) it has to be syntactically complete, i.e., it has to include all three possible values of the *Result* attribute (i.e., `success`, `failure` and `normal`); and (criterion 2) it has to be semantically sound, i.e., it must contain at least one predicate referring to the vulnerability used to create the training data set, i.e., one Snort rule number. These criteria are systematic, can be used by a system to automatically verify the usability of a verification rule, and do not depend on the user of IDS-VRG.

To illustrate this, Figure 7.7 presents an example of an unusable verification rule and Figure 7.8 presents a usable IDS verification rule generated using the training data set for the vulnerability corresponding to Bugtraq ID (BID) 2674 (which is detected by Snort rule number 971). The verification rule presented in Figure 7.7 specifies that if Snort raises an IDS event for rule number 1292, then the attack is successful, otherwise the attack fails. This verification rule is syntactically incomplete because it does not provide a conclusion for all predicted values: `normal` is missing (criterion 1). Moreover, this verification rule is not semantically sound because it does not contain a Snort rule to detect the exploitation of BID 2674 (criterion 2): Snort rule number 1292 specifies a target reaction (i.e., `directory listing`) that could be related to an attack. Thus, this verification rule is not *usable* because it does not specify whether identifying a directory listing is part of normal traffic (criterion 1). If a directory listing is actually part of normal traffic, this verification rule is unable to distinguish normal traffic from successful attacks; and if it is not part of normal traffic this verifica-

tion rule is unable to distinguish failed attacks from normal traffic. Moreover, this verification rule is not *usable* because a directory listing is not an attack against vulnerability BID 2674 (criterion 2).

The IDS verification rule presented in Figure 7.8, on the other hand, is usable. It provides all possible conclusions (criterion 1) and a predicate describing the attack attempt that refers to the exploitation of BID 2674, specifically Snort rule 971 (criterion 2).

For the semantically sound criterion (criterion 2), it is also crucial to verify whether there are attributes that are equivalent. We say that two attributes a_1 and a_2 are equivalent (with respect to a training data set *tds*) if there is a bijection f between the sets of values of a_1 and a_2 such that for all instances x in *tds*, we have $f(a_1(x)) = a_2(x)$. In other words, a_2 can be seen as a renaming of a_1 for *tds*. For example, suppose that for all instances in the training data set for vulnerability BID 2674 (detected by Snort rule 971), the values of the Snort rule 1292 attribute are equal to the values of the Snort rule 971 attribute. Since the C4.5 algorithm only provides one verification rule as an output, and because it only uses one attribute for classification purposes when there are several equivalent attributes in the training data set, the generated rule could either contain the `Snort(1292)` or `Snort(971)` attributes. Therefore, if the verification rule only contains the predicate `Snort(1292) = true` the verification rule is not *usable* (criterion 2): this is the wrong rule for BID 2674. To ensure a semantically sound verification rule, we use an algorithm that identifies attribute equivalences. To do so, one possible approach could be to identify and remove equivalent attributes in the training data (i.e., before generating the verification rule) and only keep the attributes that satisfy the criterion 2. One issue though (recall the definition of equivalence) is that two attributes can be equivalent for a training data set, i.e., for a vulnerability, but may not be equivalent for another training data set (i.e., another vulnerability), which would prevent any a priori identification and removal of equivalent attributes. In IDS-VRG, we therefore manually removed equivalent attributes after the generation of the verification rules.

```
if Snort(1292):dir listing = true then
    success
else
    Failure
```

Figure 7.7: Unusable Verification Rule Example for BID 2674

```
if Snort(971):WEB-IIS .printer access = true then
    if StandardMessage(HTTP,response) = true then
        success
    else
        failure
else
    normal
```

Figure 7.8: Usable Verification Rule Example for BID 2674

7.3 Multi-Session IDS

IDS verification rules we generated using the $HostPortState(host, port, i, j)$ or $HostState(host, i)$ target reaction indicators (Section 7.2.2.1) require a multi-session IDS engine (i.e., multiple transport layer sessions have to be analyzed to trigger such verification rules). Verification rules only using the $StandardMessage(protocol, i)$ indicators can be implemented in Snort because they can be monitored using one session. Snort relies on a multi-packet IDS engine that is only able to monitor packets within one transport layer session to identify attacks. However, some of the IDS verification rules generated with IDS-VRG require the IDS engine to monitor packets in multiple sessions to distinguish between successful and failed attacks. Moreover, the absence of any reaction from the target after an attack, required by some verification rules (Section 7.4.2.1), also prevents these verification rules from being implemented using the Snort IDS engine. To overcome these problems, we developed a Multi-Session IDS (MS-IDS) as the IDS to test the generated verification rules.

In this section, we outline the architecture of MS-IDS. MS-IDS has a multi-session IDS engine that can monitor complex communication patterns involving multiple packets in multiple sessions. In particular, this allows MS-IDS to capture the reaction of the target during an attack. MS-IDS is composed of two IDS engines and three IDS rule databases (see Figure 7.9).

The Snort IDS engine identifies relevant packets by using two IDS rule databases: the default Snort rule database and the Indicators. Recall from Section 7.2.2.1 that we developed a set of Snort

rules to provide the target reaction indicators required by the IDS verification rules. All the events generated by the Snort IDS engine are sent to the MS-IDS engine. The MS-IDS engine then uses the database containing our verification rules to verify the attack attempts. The resulting events are displayed in a GUI and stored in a file.

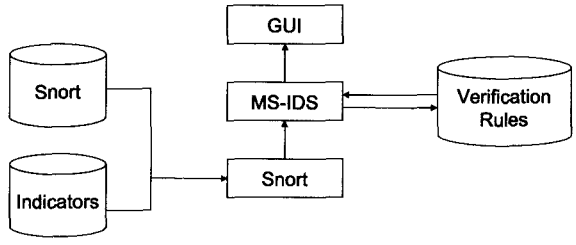

Figure 7.9: MS-IDS Overview

Based on results provided in [48], we decided to use interval logic (a form of temporal logic) as a paradigm for our multi-session MS-IDS engine. Since we did not know if we could have access to an interval logic IDS engine such as the one presented in [48], that the approach of [48] is very well documented and feasible to implement, we decided to develop one in Prolog. Prolog offers an ideal programming environment because it is a rule-based language that facilitates the implementation of the interval logic rules [48] that are the basis of the interval logic.

7.4 Case Study

In this section, we describe the IDS verification rules obtained using IDS-VRG. Section 7.4.1 presents the design of this case study. Section 7.4.2 presents the verification rules that we obtained using IDS-VRG and a comparative analysis against Bro and Snort UC Davis when our rules are used with MS-IDS. Section 7.4.3 discusses the limitations of IDS-VRG.

7.4.1 Experiment Design

We used IDS-VRG to generate verification rules for 16 vulnerabilities (Section 7.2.2.2) and manually added temporal constraints to make them IDS-ready (Section 7.2.3.1). To evaluate those rules we

compared them (i.e., we executed MS-IDS) to the verification rules included in Bro and Snort UC Davis on the attack scenarios of our VEP data set. We selected Bro and Snort UC Davis because they contain verification rules (manually) derived from the default Snort rule database. To compare these three IDSs, we used the verification metrics presented in Section 2.5 to assess their ability to distinguish between successful and failed attack attempts. We used the same test oracle to assess the verification accuracy of Bro, Snort UC Davis and MS-IDS as the one presented in Section 4.3.1.3. To analyze the results presented in Section 7.4.2, we used the verification accuracy A_V, the TP_V and the TN_V rates described in Section 2.5.

Remember that the A_V is defined as the ratio of properly predicted values (i.e., TP_V and TN_V) over the total number of test cases. The TP_V rate is defined as the ratio of properly predicted values for the successful test cases (i.e., TP_V) over the total number of successful test cases. The TN_V rate is defined as the ratio of properly predicted values for the failed test cases (i.e., TN_V) over the total number of failed test cases. These measures have to be used together to compare the accuracy of different verification rules. A_V is not sufficient since it hides the fact that FP_V and FN_V do not have the same importance. Even if it is desirable to minimize FP_V, this should not be done at the cost of increasing FN_V. An accurate verification rule is therefore one that has no FN_V (i.e., TP_V rate $= 1$) and that minimizes the number of FP_V (i.e., TN_V rate ~ 1). Thus, it is a verification rule that only has TP_V and maximizes the TN_V.

7.4.2 Results

Table 7.1 presents the verification accuracy (A_V), the TN_V rate and the TP_V rate obtained when using Bro, Snort UC Davis, and MS-IDS on the attack scenarios of the 16 training data sets (Section 7.2.2.2). The BID column identifies the training data set (i.e., test case group) using the BugtraqID (BID) (i.e., vulnerability identifier) related to the vulnerability exploited in the training data sets.

7.4.2.1 Accuracy

Table 7.1 shows that MS-IDS is either more accurate than or equivalent to Bro and Snort UC Davis for all the 16 vulnerabilities in the training data sets, achieving perfect (100%) TP_V and TN_V rates. We conducted a manual semantic analysis of our verification rules and compared them to Bro and Snort UC Davis to better understand the root causes of the differences between MS-IDS, Bro and Snort UC Davis. We drew a number of conclusions from this analysis.

		MS-IDS			Bro			Snort UC Davis		
BID	**Type**	$\mathbf{A_V}$	$\mathbf{TN_V}$	$\mathbf{TP_V}$	$\mathbf{A_V}$	$\mathbf{TN_V}$	$\mathbf{TP_V}$	$\mathbf{A_V}$	$\mathbf{TN_V}$	$\mathbf{TP_V}$
2708	WA	100%	100%	100%	100%	100%	100%	100%	100%	100%
1806	WA	100%	100%	100%	100%	100%	100%	100%	100%	100%
8035	WA	100%	100%	100%	96%	96%	50%	72%	71%	100%
3335	WA	100%	100%	100%	71%	96%	0%	71%	96%	0%
2674	WA	100%	100%	100%	92%	100%	0%	92%	100%	0%
4482	DOS	100%	100%	100%	5%	0%	100%	95%	100%	0%
514	DOS	100%	100%	100%	96%	100%	0%	96%	0%	100%
1163	DOS	100%	100%	100%	37%	0%	100%	37%	0%	100%
5556	DOS	100%	100%	100%	45%	0%	100%	45%	0%	100%
10115	DOS	100%	100%	100%	53%	0%	100%	53%	0%	100%
7106	AA	100%	100%	100%	1%	0%	100%	1%	0%	100%
7294	AA	100%	100%	100%	6%	0%	100%	6%	0%	100%
10108	AA	100%	100%	100%	18%	0%	100%	18%	0%	100%
9633	AA	100%	100%	100%	22%	0%	100%	22%	0%	100%
10116	AA	100%	100%	100%	28%	0%	100%	28%	0%	100%
8205	AA	100%	100%	100%	19%	0%	100%	19%	0%	100%

WA = Web Attack
DOS = Denial of Service
AA = Admin Attempt

Table 7.1: IDS Verification Rules compared to Bro and Snort UC Davis

First, our verification rules are semantically equivalent to the correct verification rules of Bro and Snort UC Davis, i.e., the rules for BID 2708 and 1806: the three IDSs achieve 100% A_V, 100% TN_V and 100% TP_V.

Second, our verification rules address the verification problems identified in Chapter 4 related to the missing verification rule by improving the verification accuracy over Bro and Snort UC Davis in a number of cases. For example, consider our rule for BID 8035. We obtained a more accurate verification rule than Bro and Snort UC Davis because our verification rule used the port state indicators instead of the standard message of the protocol to verify the attacks.

Third, for BID 2674, 3335 and 4482 both Bro and Snort UC Davis have a 0% TP_V rate, i.e., they did not detect successful attacks. In the case of Bro, note that the IDS rule for BID 4482 is not instrumented. Thus, this rule is not a verification rule and Bro is unable to verify this attack. Our IDS-VRG also generated verification rules that could replace the incorrect Bro verification rules for BID 2674 and 3335 identified in Chapter 4. For BID 3335, Bro is waiting for the wrong reaction from the target system (Section 5.2.2). For BID 2674, Bro waits to evaluate whether or not the target system is a Microsoft IIS Server. Bro uses the HTTP reply message (e.g., HTTP 200 OK) from the server to gather this information. However, the server does not provide an HTTP reply message when the attack against BID 2674 is successful. Thus, Bro is unable to detect a successful attack against this vulnerability. This explains the TP_V rate of 0% for these two BIDs when using Bro.

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS
(msg:"WEB-IIS ISAPI .printer access attempt";
flow:to_server,established; flowbits:set,flag; flowbits:noalert;
uricontent:".printer"; nocase;
sid:971; rev:9;)

alert tcp $HTTP_SERVERS $HTTP_PORTS -> $EXTERNAL_NET any
(msg:"WEB-IIS ISAPI .printer access success";
flow:to_client,established; flowbits:isset,flag; flowbits:unset,flag;
content:!"HTTP";
sid:10971; rev:1;)

alert tcp $HTTP_SERVERS $HTTP_PORTS -> $EXTERNAL_NET any
(msg:"WEB-IIS ISAPI .printer access failure";
flow:to_client,established; flowbits:isset,flag; flowbits:unset,flag;
content:"HTTP";
sid:20971; rev:1;)
```

Figure 7.10: Snort Verification Rule for BID 2674

In the case of Snort UC Davis, for BID 2674, 3335 and 4482, we also have a 0% TP_V rate for other reasons than Bro. Note that the IDS rule for BID 3335 is not instrumented in Snort UC Davis. Thus, this rule is not a verification rule and Snort UC Davis is unable to verify this attack. In the case of BID 2674 and 4482, the verification rules that detect attacks related to these vulnerabilities

are correct from a logical point of view, but they require a *not* operator to verify successful attacks, which the Snort engine interprets differently than what was meant by the authors of the Snort UC Davis verification rules. For example, consider our usable verification rule for BID 2674 in Figure 7.8. It specifies that the IDS should first identify the attack specified by Snort rule 971 and, if the attack is not followed by any HTTP response message from the target system, then the attack is successful, otherwise, the attack has failed. Figure 7.10 shows how Snort detects and verifies attacks for BID 2674 using the `flowbits` mechanism. Figure 7.10 is a simplified version of the one found in Snort UC Davis for the purpose of illustrating the problem. Note that the third rule is not in Snort UC Davis, but we included it in this example, so the IDS could also distinguish between normal traffic and failed attack attempts. Figure 7.10 and Figure 7.8 are very similar: Snort UC Davis rule 971 (i.e., a modified version of Snort rule 971 with the `flowbits` plug-in) is first used and then an HTTP answer from the target is either detected (third rule in Figure 7.10) or not (second rule in Figure 7.10): the last two rules in Figure 7.10 correspond to the two alternatives of the rule in Figure 7.7 and Figure 7.8. It is the reaction of the target system that allows verification, and the meaning of *no HTTP message response should be sent by the target system* is therefore critical. To trigger rule 10971 (i.e., verify that the attack is successful), a packet has to at least match the rule header (i.e., `tcp $HTTP_SERVERS $HTTP_PORTS -> $EXTERNAL_NET any`). Thus, at least a packet has to be sent from the target system after the attack to trigger this rule. In the case of this attack, no reply (i.e., packet) is expected (Figure 7.7 and Figure 7.8). Thus, this Snort specification of the verification rule for BID 2674 is incorrect. The Snort IDS engine interprets Snort rule 10971 as *the next packet should not be an HTTP packet*, whereas the intent of the designer of the rule was *there is no HTTP packet*. This explains the TP_V rate of 0% for these two BIDs when using Snort UC Davis. The problem is that it is not possible to specify such a verification, whereby the absence of reaction needs to be checked, using the current Snort signature language. Any rule in Snort UC Davis that requires the detection of no reaction from the target system, which is the case for a buffer overflow attack (e.g., BID 2674) and a DOS attack (e.g., BID 4482), should have the same problem. Since

our MS-IDS engine does not only rely on Snort to evaluate rules, we do not have this problem and we have accurate verification rules.

Fourth, Table 7.1 shows that a large number of rules in Bro and Snort UC Davis have a detection part but do not have a verification part (TP_V rate of 100% and TN_V rate of 0%), as opposed to our rules. Their corresponding BIDs are highlighted in Table 7.1. In other words, IDS-VRG generated verification rules not found in Bro or in Snort UC Davis. There are exceptions though. Snort UC Davis contains verification rules, inherited from Snort 2.3.0, for BID 8205 and 7294. However, Bro has a detection part for BID 514 but it is unable to detect the attack because it is an IGMP attack and Bro is unable to decode the IGMP protocol (Section 4.3.2.1). In this case, the TP_V rate is 0% and the TN_V rate is 100% (silence is interpreted as a verified failed attack, see Section 4.3.1.3) because Bro remains silent for all attack attempts.

7.4.2.2 Usability

We analyzed the usability of the verification rules generated by IDS-VRG. We present the verification rules based on the attack classes (i.e., Web Attack, Admin Attempt and DOS) because we observed that verification rules from the same attack class contain similar predicates. We present the C4.5 verification rules and not the IDS verification rules with temporal constraints to show that the temporal constraints can easily be inferred by a network security expert from these verification rules.

For all training data sets in the Admin Attempt attack class (i.e., for BID 7106, 7294, 10108, 9633, 10116 and 8205), the C4.5 algorithm provides usable verification rules (according to criteria 1 and 2, Section 7.2.4). All the verification rules generated in the Admin Attempt attack class are similar to the verification rule generated from the BID 7294 training data set: what differs is the Snort rule being used for attack detection. To verify the attack, all the verification rules in the Admin Attempt attack class monitor the effect of the attack on the state of ports of the target system. Thus, to simplify the current discussion, we only present in Figure 7.11 the verification

rules for BID 7294 as generated by C4.5.

```
if Snort(2103):NETBIOS SMB trans2open buffer overflow attempt = true then
    if HostPortState(target,port,close,open) = true then
        success
    else
    failure
else
    normal
```

Figure 7.11: IDS Verification Rule for BID 7294

This verification rule specifies that an attack is successful for BID 7294 when the attack matches the Snort rule 2103 and when a port on the target system other than the attacked port is changing its state from closed to open. The attack fails when the attack matches the Snort rule 2103 and when all ports on the target system other than the attack port remain closed. There is no attack when the attack does not match the Snort rule 2103. In the temporal version of that C4.5 verification rule, we added the requirement that the attack attempt has to occur before the port state verification.

However, even if this verification rule is correct (i.e., it should not generate FP_V), it is not complete (i.e., it may generate FN_V) because the attack may have a different effect than changes to a port state: for instance it could result in a remote shell being opened on the target system. The current version of the VEP data set used to generate our training data sets only contains Admin Attempt that open a direct (i.e., the attacker connects to the target system) or reverse (i.e., the target system connects to the attacker) remote shell on the target system when it is successful and exploits the corresponding vulnerability. This illustrates one limitation of our results: a larger, more diverse set of attacks would lead to more complete verification rules. Recall from Table 7.1 and the discussion in the previous section that Bro and Snort UC Davis do not have verification rules for these attacks. One solution to address this limitation is to use our approach to generate verification rules that verify only the failed attacks (i.e., normal reaction when an attack failed) and a successful attack should be everything that does not match the failed attack reaction, i.e., a *negative* IDS verification rule. However, this is outside the scope of this thesis and we plan to address to explore this solution in our future work.

For all the DOS attack class training data sets (i.e., BID 4482, 514, 1163, 5556 and 10115), the C4.5 algorithm also provides usable verification rules. All the verification rules generated from the DOS attack class training data sets are similar to the verification rules generated for BID 514: what varies is the Snort rule used to detect the attack, and the target state verification (i.e., host down, port closed, application unavailable) after the DOS attack. Thus, we only present the verification rule for BID 514 to simplify this analysis. Figure 7.12 describes the verification rule generated using the C4.5 algorithm. This verification rule describes that an attack attempting to exploit vulnerability BID 514 is successful if the attack matches the Snort rule 273 and the target system is down. The attack fails if it matches the Snort rule 273 and the target system is up. When the traffic does not match the Snort rule 273, the traffic is normal. We specified that the attack attempt has to come before the target system state change in the IDS verification rule. This is what is expected when a DOS is attempted against a target system. Thus, this verification rule is correct and usable. Again, recall that Bro and Snort UC Davis do not have verification rules for these BIDs, with the exception of BID 4482 for Snort UC Davis. However, we demonstrated in the previous section that the Snort UC Davis verification rule for BID 4482 is incorrectly specified.

```
if Snort(273):DOS IGMP dos attack = true then
    if HostState(target,up) = true then
        failure
    else
        success
else
    normal
```

Figure 7.12: IDS Verification Rule for BID 514

For all the Web Attack class training data sets (i.e., for BID 2708, 1806, 3335, 2674 and 8035), all our verification rules are usable and equivalent, in terms of usability, to the correct verification rules of Bro and Snort UC Davis (BID 2708, 1806 and 8035). All the verification rules generated in the Web Attack class are similar to the verification rule of BID 1806: what varies is the Snort rule that detects the attacks and the standard HTTP response message used to verify the attack. Thus, we only present the verification rule for BID 1806 to simplify the analysis of the results. Figure 7.13

presents this verification rule for BID 1806, generated using the C4.5 algorithm.

```
if Snort(1002):WEB-IIS cmd.exe access = true then
    if StandardMessage(HTTP,200) = true then
        success
    else
        failure
else
    normal
```

Figure 7.13: IDS Verification Rule for BID 1806

The rule specifies that an attack succeeds when the attack matches the Snort rule 1002 and there is an HTTP message 200 OK from the target system. The attack fails when it matches the Snort rule 1002 but there is no HTTP message 200 OK from the target system, and there is no attack against the BID 1806 vulnerability when the traffic does not match Snort rule 1002.

Recall that our verification rules for BIDs 2674 and 3335 can replace the incorrect verification rules in Bro. Bro verification rules also instrument the Snort rule being used to detect these attacks, but these rules use the 200 OK HTTP message for verification purposes. However, for BID 3335, the Bro verification rule is not correct because it is the type of HTTP error message that decides the success or failure of the attack. In the case of BID 2674, as explained previously, it is the absence or the presence of an HTTP message after the attack that defines the success or failure of the attack. The Snort UC Davis does not have this problem. It has the correct verification rule for BID 2674, and the verification rule for BID 3335 was not instrumented in Snort UC Davis. However, as explained in Section 7.4.2.1 the Snort UC Davis verification rule for BID 2674 does not work because it is not specified correctly in Snort.

7.4.3 Limitations

The IDS verification rules obtained using our IDS-VRG are more accurate than or equivalent to the ones used by Bro and Snort UC Davis. However, as discussed in Section 7.4.2.2, some of our verification rules are too specific because of the lack of diversity of the attack scenarios contained in our VEP data set and of the attributes used to monitor the target reaction behaviour. However, this

is a limitation of the data set used as input, and not a limitation of the methodology we followed: a larger, more diverse data set would lead to more complete rules. We also plan in our future work to explore the possibility of *negative* IDS verification rule to address this issue.

Nevertheless, this means that we need to have VEPs that exploit the vulnerabilities to generate IDS verification rules for them, which is not always the case. In reality, when the documentation of a vulnerability is just released, there is usually no VEP available and what IDS vendors want is to develop (manually) the IDS detection and verification rules as soon as possible, so their IDS users would be able to detect and verify this attack. Thus, our approach is efficient when there is a backlog of (missing) IDS verification rules to generate (e.g., in the case of Bro and Snort) to catch up with what should have been done with previous vulnerabilities by the IDS rule developers. However, based on our results in Section 7.4.2, we still think that this approach has to be used to help tweaking a manually created IDS verification rules for two reasons. First, when VEPs become available, our approach could be used to confirm that a manually generated rule is similar to the verification rule provided by our approach, to increase the level of confidence into the manually created verification rule. Second, as being used in this chapter, our approach could generate a more accurate (correct) verification rule when the manually created one has accuracy problems, which was the case for some IDS verification rules in Bro and Snort UC Davis.

Another limitation, discussed in Section 7.2.3.1, is that the data mining algorithm we used does not generate verification rules with temporal constraints: the rule shows predicates but does not indicate in what order the predicates have to be evaluated. Nevertheless, we have illustrated that identifying the adequate order of evaluation of the predicates would be a straightforward task for a network administrator.

Dreger [63] identified that the state that needs to be kept by the IDS and the per packet analysis time are two major factors that slow down network intrusion detection systems. Thus, using verification rules could introduce processing costs for IDS. In this research, we did not address the processing costs related to these verification rules. However, we know that although some verification

rules can be very accurate, they could degrade the performance of an IDS. For instance, the target reaction approach increases the per packet analysis time because the success and error message of protocols that have to be monitored in addition to packets already monitored by the IDS. Moreover, these protocol messages are frequent on a network. Thus, the gathering of this information involves the frequent triggering of rules monitoring these messages and influences the per packet analysis time. However, in this case, the protocol messages only have to be observed after attacks, thus only a subset (i.e., the ones that follow attack attempts) of these messages is required.

Using verification rules also has an impact on the information the IDS has to keep in memory. For instance, our verification rules require that some information (e.g., attack attempt) has to be kept in memory by the IDS while waiting for the target reaction. Thus, verification rules would also increase the processing cost because the IDS has to keep the states of the verification rule in memory.

7.5 Conclusion

In this chapter, we proposed an approach that generates automatically verification rules for IDSs. The overall objectives were to provide techniques that automatically generate IDS verification rules to improve IDS accuracy by preventing human errors, to address the problems of missing and incorrect verification rules identified in Chapter 4 and to reduce the number of FP_V. This approach relies on documented traffic traces and a data mining algorithm to generate the verification rules. To our knowledge, this is the first time a data mining algorithm is used to generate verification rules and we believe that this approach has the potential to help improve the accuracy of IDS rules in the future. We also propose new target reaction approach indicators (i.e., host and port state) and show that these indicators improve the verification accuracy.

We built our IDS Verification Rule Generator using the C4.5 data mining algorithm and its implementation in the Weka framework to automatically generate verification rules. However, our

IDS Verification Rule Generator has some limitations as it cannot generate verification rules with temporal constraints (required to more accurately verify the attack). Thus, the temporal constraints were manually derived to create actual IDS verification rules. Verification rules without temporal constraints are less accurate, and thus generate more FP_V. However, we have shown that temporal constraints could be easily added to our verification rules by a network administrator.

We proposed a multi-session IDS, called MS-IDS, to test our verification rules because the current Snort rule specification language (Snort 2.8) does not support multi-session rules. We used our verification rules with MS-IDS and compared their accuracy against Bro and Snort UC Davis on an existing set of attack network traces.

Our analysis showed that our automatically generated verification rules are more accurate than or equivalent to the ones used by Bro and Snort UC Davis. We automatically generated rules that are semantically equivalent to some of the correct verification rules of Bro and Snort UC Davis and, furthermore, we were able to generate correct verification rules to replace some of the incorrect verification rules in Bro and Snort UC Davis. New verification rules were also generated that were not included in these IDSs.

Chapter 8

Conclusion

In this thesis, we proposed three software verification and validation techniques in Chapter 4, 5 and 6 to identify the root causes of the detection (i.e., evasion) and verification problems of IDSs, in the context of attack traffic, by proposing well-defined, systematic procedures to measure and assess these accuracy problems. Based on these results, we clearly derived research objectives to address the root causes of these accuracy problems in a systematic manner. From these research objectives, in Chapter 7, we also proposed new IDS signatures (specification). Consequently, in this thesis, contributions to address the detection and verification problems were made by improving and proposing software verification and validation techniques, and new IDS specifications.

More precisely, in Chapter 4 we presented the Vulnerability Exploitation Program (VEP) testing model. We proposed a VEP testing model to automatically and iteratively assess the detection and verification problems of IDSs. Our VEP testing model addresses at the same time two different, but complementary, problems which are the verification of the implementation (i.e., the IDS engine) and the validation of the engine specification (i.e., the IDS signatures). We also proposed a methodology to keep up with the dynamic nature of the accuracy problem (i.e., new attacks are identified daily). The resulting system, the Automatic Experimentation System / Virtual Laboratory (AES/VLab), was used to automatically generate test cases using a well-defined test criterion and using real attacks

generated by VEPs to assess the detection and verification problems of IDSs. We also fulfilled a need of the IDS research community by making publicly available on the Internet the documented test cases we generated with our VEP testing model. We presented an analysis of the limits of the VEP testing model. We also performed a case study to assess the detection and verification problems using our VEP testing model on two widely used IDSs (i.e., Snort and Bro).

In Chapter 5, we presented the IDS Engine Stimulator (IDSES) testing model. This testing model clearly maps the IDS engine testing problem to software verification testing principles, thus defining clearly the sub-problems of verifying IDS engines. The IDSES testing model is used to specifically assess the detection problem in the IDS engine (i.e., verification of the IDS engine). IDSES addresses both the identification of the root causes of problems in IDS engines and the VEP availability problems. Using our IDSES, we performed a case study to assess the detection problem using different test criteria on a widely used IDS (i.e., Snort). The IDSES testing model is fully automated to also address the dynamic nature of the accuracy problem. We concluded that our IDSES can automatically identify detection problems in IDS engines that prevent the IDS from detecting attacks.

In Chapter 6, we presented the IDS Signature Space Analysis (IDS-SSA) approach. This verification method complements the IDSES by verifying the IDS specification (signature database). The IDS-SSA is used to specifically assess redundancy (i.e., inclusion/intersection) between signature specifications that could cause detection problems in the IDS specification (i.e., verification of the IDS signature database). Using our IDS-SSA, we performed a case study to assess the detection problem in the IDS specification using different signature databases of Snort. The IDS-SSA testing model is fully automated to also address the dynamic nature of the accuracy problem. We concluded that our IDS-SSA can automatically identify detection problems in the IDS signature databases that prevent the IDS from detecting attacks.

In Chapter 7, we presented an approach to automatically improve the IDS specification and implementation that addresses some of the accuracy problems identified by the IDS verification and

validation. More precisely, we proposed an approach for the automatic generation of IDS signatures and a new IDS engine which is able to use those signatures to address the accuracy verification problems identified in Chapter 4. We developed a data mining approach to automatically generate verification rules (i.e., verification signatures) to address the verification problem. We developed a multi-session IDS engine called MS-IDS to use these verification rules. We also presented the results obtained when these new IDS engine and signatures were tested against other IDSs (i.e., Bro, Snort and Snort UC Davis). We concluded that our verification rules were generally more or as accurate as the ones found in Bro, Snort and Snort UC Davis.

We can conclude that the work presented in this thesis has improved the body of knowledge in IDS research for IDS verification and validation techniques and automatic IDS signature specifications. Moreover, our work has already impacted the IDS research community, through the distribution of our VEP data set to many researchers and organizations (i.e., more than 20 universities).

However, this research work has some limitations. Consequently, in the case of the VEP testing model, our future work will look into the VEP availability problem. Our future work will also look into using state machine testing criteria, to better assess the detection accuracy of IDS engines for the IDS Engine Stimulator testing model. In the case of the IDS Signature Space Analysis, our future work will focus on the signature intersection analysis, which we showed to be quite computationally intensive, to verify whether there are sets (and how many they are) of intersecting signatures of size greater than the signature limit and to calculate (if possible) or approximate the optimal signature limit.

Moreover, we will explore the idea of using the IDS-SSA to compare the space covered by signatures from different IDS vendors that identify the same attacks. For instance, we propose to explore the possibility of comparing signatures from Snort and Bro. Finally, in the case of automatic generation of IDS signatures, our future work will focus on addressing the missing temporal constraints to automate further our approach and look into negative IDS verification signatures to address the problem of missing successful attack reaction in the training data set.

Nevertheless, after completing this research work, we strongly believe that we proved that (1) it is important to understand the dynamic nature of the IDS accuracy problem, (2) any research work that tries to improve IDS accuracy must start with an assessment of the accuracy problem to provide a clear understanding of this problem, (3) this work must be iterative to ensure that the resulting work on IDS technology is still up-to-date, and (4) to account for the dynamic nature of the accuracy problem, one of the key solutions is automation.

Bibliography

[1] Treinen, J.J., Thurimella, R.: A Framework for the Application of Association Rule Mining in Large Intrusion Detection Infrastructures. In: Proceedings of the Recent Advances in Intrusion Detection (RAID). (2006) 1–18

[2] Manganaris, S., Christensen, M., Zerkle, D., Hermiz, K.: A Data Mining Analysis of RTID Alarms. In: Proceedings of the Recent Advances in Intrusion Detection (RAID). (1999)

[3] Julisch, K.: Mining Alarm Clusters to Improve Alarm Handling Efficiency. In: Proceedings of the Annual Computer Security Applications Conference (ACSAC). (2001) 12–21

[4] Julisch, K., Dacier, M.: Mining Intrusion Detection Alarms for Actionable Knowledge. In: Proceedings of the ACM Conference on Knowledge Discovery and Data Mining (SIGKDD). (2002) 366–375

[5] Julisch, K.: Clustering Intrusion Detection Alarms to Support Root Cause Analysis. ACM Transactions on Information and System Security **6** (2003) 443–471

[6] Vigna, G., Kemmerer, R.A.: NetSTAT: A Network-Based Intrusion Detection Approach. In: Proceedings of the Annual Computer Security Applications Conference (ACSAC). (1998) 25–34

[7] Patton, S., Yurcik, W., Doss, D.: An Achilles' Heel in Signature-Based IDS: Squealing False Positives in Snort. In: Proceedings of Recent Advances in Intrusion Detection (RAID). (2001)

[8] Krügel, C., Robertson, W.K.: Alert Verification Determining the Success of Intrusion Attempts. In: Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA). (2004) 25–38

[9] Sommer, R., Paxson, V.: Enhancing Byte-Level Network intrusion Detection Signatures with Context. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS). (2003) 262–271

[10] Zhou, J., Carlson, A.J., Bishop, M.: Verify Results of Network Intrusion Alerts Using Lightweight Protocol Analysis. In: Proceedings of the Annual Computer Security Applications Conference (ACSAC). (2005) 117–126

[11] Timm, K.: IDS Evasion Techniques and Tactics. `http://www.securityfocus.com/infocus/1577` (2002)

[12] Hacker, E.: IDS Evasion with Unicode. `http://www.securityfocus.com/infocus/1232` (2001)

[13] Marty, R.: THOR - A Tool to Test Intrusion Detection Systems by Variations of Attacks. Master's thesis, ETH Zurich (2002)

[14] Handley, M., Kreibich, C., Paxson, V.: Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics (HTML). In: Proceedings of USENIX Security Symposium. (2001)

[15] Roelker, D.: HTTP IDS Evasions Revisited. `http://docs.idsresearch.org/http_ids_evasions.pdf` (2006)

[16] Ptacek, T., Newsham, T.: Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. Technical report, Secure Networks (1998)

[17] Vigna, G., Robertson, W., Balzarotti, D.: Testing Network-Based Intrusion Detection Signatures using Mutant Exploits. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS). (2004) 21–30

[18] Beale, J., Foster, J.C.: Snort 2.0 Intrusion Detection. Syngress Publishing (2003)

[19] Paxson, V.: Bro: A System for Detecting Network Intruders in Real-Time. Computer Networks **31** (1999) 2435–2463

[20] Massicotte, F., Gagnon, F., Labiche, Y., Couture, M., Briand, L.: Automatic Evaluation of Intrusion Detection Systems. In: Proceedings of the Annual Computer Security Applications Conference (ACSAC). (2006) 361–370

[21] Gagnon, F., Massicotte, F., Esfandiari, B.: Using Contextual Information for IDS Alarm Classification. In: Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA). (2009)

[22] Leita, C., Dacier, M., Massicotte, F.: Automatic Handling of Protocol Dependencies and Reaction to 0-Day Attacks with ScriptGen Based Honeypots. In: Proceedings of the Recent Advances in Intrusion Detection (RAID). (2006) 185–205

[23] Massicotte, F., Briand, L.C., Labiche, Y.: Specification-based testing of intrusion detection systems. Carleton University Technical Report SCE_08_03, Carleton University (2008)

[24] Massicotte, F., Labiche, Y.: Specification-Based Testing of Intrusion Detection Engines using Logical Expression Testing Criteria. In: Proceedings of the International Conference on Quality Software (QSIC). (2010)

[25] Massicotte, F.: Packet Space Analysis of Intrusion Detection Signatures. In: Proceedings of the Recent Advances in Intrusion Detection Poster Session (RAID). (2009)

[26] Massicotte, F., Labiche, Y., Briand, L.: Toward Automatic Generation of Intrusion Detection System Verification Rules. In: Proceedings of the Annual Computer Security Applications Conference (ACSAC). (2008) 279–288

[27] Debar, H., Dacier, M., Wespi, A.: Towards a Taxonomy of Intrusion Detection Systems. Computer Networks **31** (1999) 805–822

[28] Witten, I.H., Frank, E.: Data Mining: Practical Machine Learning Tools and Techniques, Second Edition (Morgan Kaufmann Series in Data Management Systems). Morgan Kaufmann (2005)

[29] Lippmann, R., Fried, D., Graf, I., Haines, J., Kendall, K., McClung, D., Weber, D., Webster, S., Wyschogrod, D., Cunningham, R., Zissman, M.: Evaluating Intrusion Detection Systems: The 1998 DARPA Off-line Intrusion Detection Evaluation. In: Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX). (2000)

[30] Lippmann, R., Haines, J.W., Fried, D.J., Korba, J., Das, K.: The 1999 DARPA Off-line Intrusion Detection Evaluation. Computer Networks **34** (2000) 579–595

[31] Moore, D., Shannon, C., Brown, D.J., Voelker, G.M., Savage, S.: Inferring Internet Denial-of-Service Activity. ACM Transactions on Computer Systems **24** (2006) 115–139

[32] Aussibal, J., Borgnat, P., Labit, Y., Dewaele, G., Larrieu, N., Gallon, L., Owezarski, P., Abry, P., Boudaoud, K.: Base de traces d'anomalies légitimes et illégitimes. In: Proceedings of the Conference on Security in Network Architectures and Information Systems (SAR-SSI). (2007)

[33] Mutz, D., Vigna, G., Kemmerer, R.A.: An Experience Developing an IDS Stimulator for the Black-Box Testing of Network Intrusion Detection Systems. In: Proceedings of the Annual Computer Security Applications Conference (ACSAC). (2003) 374–383

[34] The NSS Group: Intrusion Detection Systems Group Test (Edition 2) (2001)

[35] Yocom, B., Brown, K.: Intrusion Battleground Evolves. Network World Fusion (2001) 53–62

[36] Mueller, P., Shipley, G.: Cover Story: Dragon Claws its Way to the Top. Network Computing **12** (2001) 45–67

[37] Rossey, L.M., Cunningham, R.K., Fried, D.J., Rabek, J.C., Lippmann, R.P., Haines, J.W., Zissman, M.A.: LARIAT: Lincoln Adaptable Real-time Information Assurance Testbed. IEEE Aerospace Conference Proceedings (2002)

[38] Debar, H., Morin, B.: Evaluation of the Diagnostic Capabilities of Commercial Intrusion Detection Systems. In: Proceedings of the Recent Advances in Intrusion Detection (RAID). (2002)

[39] McHugh, J.: Testing Intrusion Detection Systems: A Critique of the 1998 and 1999 DARPA Intrusion Detection System Evaluations as Performed by Lincoln Laboratory. ACM Transactions on Information and System Security **3** (2000)

[40] The NSS Group: Intrusion Detection Systems Group Test (Edition 4) (2003)

[41] Mell, P., Hu, V., Lipmann, R., Haines, J., Zissman, M.: An Overview of Issues in Testing Intrusion Detection Systems. Technical Report NIST IR 7007, National Institute of Standard and Technology (2003)

[42] Barbará, D., Couto, J., Jajodia, S., Wu, N.: ADAM: A Testbed for Exploring the Use of Data Mining in Intrusion Detection. SIGMOD Record **30** (2001) 15–24

[43] Lee, W., Stolfo, S.J., Mok, K.W.: A Data Mining Framework for Building Intrusion Detection Models. In: Proceedings of the IEEE Symposium on Security and Privacy. (1999) 120–132

[44] Cohen, W.W., Singer, Y.: A Simple, Fast, and Effective Rule Learner. In: Proceedings of the National Conference on Artificial Intelligence (AAAI) and the Innovative Applications of Artificial Intelligence Conference (IAAI). (1999) 335–342

[45] Roger, M., Goubault-Larrecq, J.: Log Auditing through Model Checking. In: Proceedings of the Computer Security Foundations Workshop (CSFW). (2001) 220236

[46] Kumar, S., Spafford, E.: A Software Architecture to Support Misuse Intrusion Detection. In: Proceedings of the National Information Security Conference (NISC). (1995) 194–204

[47] Eckmann, S., Vigna, G., Kemmerer, R.: STATL: An Attack Language for State-based Intrusion Detection. Computer Security **10** (2002) 71–104

[48] Morin, B., Debar, H.: Correlation of Intrusion Symptoms: an Application of Chronicles. In: Proceedings of the Recent Advances in Intrusion Detection (RAID). (2003)

[49] Sekar, R., Guang, Y., Verma, S., Shanbhag, T.: A High-Performance Network Intrusion Detection System. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS). (1999)

[50] Ammann, P., Offutt, J.: Introduction to Software Testing. 1 edn. Cambridge University Press (2008)

[51] DeMontigny-Leboeuf, A.: A Multi-Packet Signature Approach to Passive Operating System Detection. CRC/DRDC joint Technical Report CRC-TN-2005-001 / DRDC-Ottawa-TM-2005-018, Communications Research Center Canada (2004)

[52] Beizer, B.: Software Testing Techniques. The Coriolis Group, Inc. (1990)

[53] Ostrand, T.J., Balcer, M.J.: The Category-Partition Method for Specifying and Generating Functional Tests. Communications of the ACM **31** (1988) 676–686

[54] Offutt, A.J., Liu, S., Abdurazik, A., Ammann, P.: Generating Test Data from State-based Specifications. Software Testing, Verification and Reliability **13** (2003) 25–53

[55] Yilmaz, C., Cohen, M.B., Porter, A.A.: Covering Arrays for Efficient Fault Characterization in Complex Configuration Spaces. In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA). (2004) 45–54

[56] Chen, T.Y., Lau, M.F.: Test Case Selection Strategies Based on Boolean Specifications. Software Testing, Verification and Reliability **11** (2001) 165–180

[57] Binder, R.V.: Testing Object-Oriented Systems: Models, Patterns, and Tools (The Addison-Wesley Object Technology Series). Addison-Wesley Professional (1999)

[58] Beale, J., Baker, A., Esler, J.: Snort IDS and IPS toolkit. Syngress Publishing (2007)

[59] Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation (3rd Edition). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2006)

[60] Dayioglu, B., Ozgit, A.: Use of Passive Network Mapping To Enhance Signature Quality of Misuse Network Intrusion Detection Systems. In: Proceedings of the International Symposium on Computer and Information Sciences. (2001)

[61] Quinlan, J.R.: C4.5: Programs for Machine Learning. Morgan Kaufmann (1993)

[62] Todd, A.D., Raines, R.A., Baldwin, R.O., Mullins, B.E., Rogers, S.K.: Alert Verification Evasion Through Server Response Forging. In: Proceedings of the Recent Advances in Intrusion Detection (RAID). (2007) 256–275

[63] Dreger, H., Feldmann, A., Paxson, V., Sommer, R.: Operational Experiences with High-Volume Network Intrusion Detection. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS). (2004) 2–11