UNIVERSITÀ DI PISA

Università degli Studi di Pisa
Facoltà di Ingegneria

Corso di Laurea Specialistica in

Ingegneria Informatica

A formalization and analysis of high-speed stateful signature matching for intrusion detection

Tesi di Laurea di
Luca Foschini

Relatori:
Prof. Giovanni Vigna, University of California, Santa Barbara
Prof. Gianluca Dini, University of Pisa
Prof. Gigliola Vaglini, University of Pisa

To my brother, Guido,

and to those who will never write a thesis

nor ever understand what a dedication means.

Contents

List of Acronyms

ACK ..... *TCP* ACKnowledgement flag

ACL ..... Access Control List

API ..... Application Program Interface

ARP ..... Address Resolution Protocol

ARPANet ..... Advanced Research Projects Agency Network

ASIC ..... Application Specific Integrated Circuit

ATM ..... Asynchronous Transfer Mode

BCAM ..... binary content addressable memory

CERN ..... European Organization for Nuclear Research

CERT ..... Carnegie Mellon University's Computer Emergency Response Team

CGI ..... Common Gateway Interface

CPU ..... Central Processing Unit

CSI ..... Computer Security Institute

DARPA ..... Defense Advanced Research Project Agency

DFA ..... Determinisi Finite-state Automaton

DNS ..... Domain Name System

EIDE ..... Enhanced Integrated Drive Electronics

FBI ..... Federal Bureau of Investigation

FIFO ..... First In First Out

FIN ..... *TCP* (Transmission Control Protocol) FINished connection flag

FPGA ..... Field Programmable Gate Array

FPR ..... False Positive Rate

FTP ..... File Transfer Protocol

HIDS ..... Host-based Intrusion Detection System

HTTP ..... Hyper Text Transfer Protocol

ICMP ..... Internet Control Message Protocol

IDS ..... Intrusion Detection System

IEEE ..... Institute of Electrical and Electronics Engineers

IMP ..... Interface Message Processors

IP ..... Internet Protocol

IPS ..... Intrusion Prevention System

LDAP ..... Lightweight Directory Access Protocol

MAC ..... Media Access Control

MTU ..... Maximum Transmission Unit

NIC ..... Network Interface Controller

NIDS ..... Network-based Intrusion Detection Systems

NIS ..... Network Information Service

NIST ..... National Institute of Standards and Technology

NSF ..... National Science Foundation

NSFNET ..... National Science Foundation NETwork

NTP ..... Network Time Protocol

OC ..... Optical Carrier

OS ..... Operating System

PCI ..... Peripheral Component Interconnect

POP ..... Post Office Protocol

ROC ..... Receiver Operating Characteristic

RPC ..... Remote Procedure Call

RST ..... *TCP* ReSeT flag

SATA ..... Serial Advanced Technology Attachment

SMB ..... Server Message Block

SMTP ..... Simple Mail Transfer Protocol

SQL ..... Structured Query Language

SRAM ..... Static Random Access Memory

SRI ..... Stanford Research Institute

SSL ..... Secure Socket Layer

SUID ..... Set User ID

SYN ..... *TCP* SYNchronization flag

TCP ..... Transmission Control Protocol

TELNET ..... TELetype NETwork

TOCTTOU ... Time-Of-Check-To-Time-Of-Use

TPR ..... True Positive Rate

TTL ..... Time-To-Live

UCLA ..... University of California, Los Angeles

UCSB ..... University of California, Santa Barbara

UDP ..... User Datagram Protocol

UML ..... User Mode Linux

UNIX ..... UNIX Operating System

VDE ..... Virtual Distributed Ethernet

VPN ..... Virtual Private Network

WWW ..... World-Wide Web

XML ..... Extensible Markup Language

XPath ..... *XML* Path Language

Abstract

The increase in bandwidth over processing power has made stateful intrusion detection for high-speed networks more difficult, and, in certain cases, impossible. The problem of real-time stateful intrusion detection in high-speed networks cannot be solved by optimizing the packet matching algorithm used by a centralized process or by using custom-developed hardware. Instead, there is a need for a parallel approach that is able to decompose the problem into subproblems of manageable size.
In this thesis, we present a novel parallel matching algorithm for the signature-based detection of network attacks. The algorithm is able to perform stateful signature matching and has been implemented on off-the-shelf hardware. The initial experiments confirm that by parallelizing the rule matching process it is possible to achieve a scalable implementation of a stateful, network-based intrusion detection system.

Keywords: Intrusion Detection, Rule Matching, Stateful Analysis, High Speed Networks, Parallel Systems.

Chapter 1

Introduction

1.1 Computer Security

In recent years, networks have evolved from a mere means of communication to a ubiquitous computational infrastructure. Networks have become larger, faster, and highly dynamic. In particular, the Internet, the world-wide *TCP/IP* network, has become a mission-critical infrastructure for governments, companies, and financial institutions. As a consequence, network attacks have started to impact practical aspects of our lives.
Today's networks are very heterogeneous environments where highly-critical software applications (e.g., the control system for a dam) run side-by-side with other non-critical network-based applications (e.g., a mail server). As a consequence, network-based attacks against apparently "non-critical" services may produce unforeseen side effects of devastating proportions.

1.1.1 Growth in the Internet and Insecurity

The Internet is a network of networks composed of a set of autonomous subnetworks. The Internet has an open architecture and it is composed of different administrative domains with different (and possibly conflicting) goals. Governments, companies, universities and organizations rely on the Internet to perform mission-critical tasks.
The story of the Internet began back in the 60s on the world's first packet switching network at the National Physics Laboratory in the United Kingdom. A few years later, the *DARPA* (Defense Advanced Research Project Agency) developed the *ARPANet* (Advanced Research Projects Agency Network), and many others followed since then.
By the end of 1969, the *ARPANet* had 4 sites: *UCLA* (University of California, Los Angeles), *SRI* (Stanford Research Institute), (Menlo Park), *UCSB* (University of California, Santa Barbara) , and the University of Utah. The *IMP* (Interface Message Processors) were built around Honeywell 516 computers, and were linked using dedicated 55 Kbps phone lines.
The *ARPANet* moved to *TCP/IP* on January 1st 1983, and, after that, *DARPA* funded the development of Berkeley *UNIX*, which used a *TCP/IP* implementation, that introduced

the socket programming abstraction.
In more recent years, the *ARPANet* became a subset of the Internet and the *NSF* (National Science Foundation) deployed a supercomputer network, *NSFNET* (National Science Foundation NETwork) that created a backbone of high speed links (56Kbps links in 1986) During the 90s the Internet grew considerably in both size and traffic volume and starting from the mid '90s it became a full-fledged mass media, when Tim Berners-Lee (*CERN*) created the World-Wide Web.
In the past ten years the Internet has become the most important means of communication. Unfortunately, as the infrastructure and services have evolved so have the hype and sophistication of the attacks.
Table 1.1 reports a comparison between the growth of the Internet and the growth in insecurity. Discarding a conspicuous initial lag, the pace of security incidents and reported vulnerabilities has been catching up in the last years.
Today computer networks play a fundamental role in every kind of software-based attack. In addition, most of the attacks focus on the networks themselves, rather than simply using them as a communication channel.

1.1.2 The Plague of Software Bugs

There is an overall concept of system security in terms of:

- privacy/confidentiality
- integrity/consistency
- availability.

When a system lacks one or more of this requirements it is deemed insecure. This often happens because of some mistake done at some level in the development and deployment process.
The *IEEE* Standard Glossary of Software Engineering Terminology [otCSotI90] defines bug (or fault) as:

> an incorrect step, process, or data definition in a computer program when a fault gets triggered, it might generate a failure.

Bugs create unexpected behaviors that can possibly lead to a vulnerability [Wik07c]. A vulnerability is a weakness in a system allowing an attacker to violate (by means of an exploit) its confidentiality, integrity or availability.
Some applications might work as designed but they contain vulnerabilities. Broadening the concept of bug, security problems can arise when systems are utilized out of the scope for which they were designed or when configured in an unsafe manner (i.e., a strong authentication system with an easily-guessable default password)
There is a large number of generic security vulnerabilities produced by bad coding practices or design flaws. The Security Portal for Information System Security Professionals

Date	Hosts	Reported Incidents	Reported Vulnerabilities
Dec 1969	4	-	-
Sep 1971	18	-	-
Dec 1973	31	-	-
Oct 1974	49	-	-
Jan 1976	63	-	-
Mar 1977	111	-	-
Aug 1981	213	-	-
May 1982	235	-	-
Aug 1983	562	-	-
Oct 1984	1024	-	-
Oct 1985	1,961	-	-
Feb 1986	2308	-	-
Dec 1987	28,174	-	-
Oct 1988	56,000	6	-
Oct 1989	159,000	132	-
Oct 1990	313,000	252	-
Oct 1991	617,000	406	-
Oct 1992	1,136,000	773	-
Oct 1993	2,056,000	1,334	-
Oct 1994	3,864,000	2,340	-
Jul 1995	8,200,000	2,412	171
Jul 1996	16,729,000	2,573	345
Jul 1997	26,053,000	2,134	311
Jul 1998	36,739,000	3,734	262
Jul 1999	56,218,000	9,859	417
Jul 2000	93,047,785	21,756	1,090
Jul 2001	125,888,197	52,658	2,437
Jul 2002	162,128,493	82,094	4,129
Jan 2003	171,638,297	137,529	3,784

Table 1.1: Source: Internet Systems Consortium and *CERT*, http://www.kernelthread.com/publications/security/net.html. Security numbers are for the entire year, and only include those that were reported to *CERT*.

[Inf06] reports a wide -although not exhaustive- classification of common causes of security vulnerabilities: most of them are directly related to software, and are caused by bad coding or configuration practices such as:

buffer overflows	empty or common passwords
unexpected input	mis-configuration
unhandled input	trust relationships
race conditions,	protocol flaws
default configuration	

These software flaws can be found and exploited in server software, client applications, and the operating system itself. Sometimes protocols are inherently flawed and therefore any applications making use of that specific protocol will be vulnerable.
Attackers exploit vulnerabilities in order to gain access to restricted information and services or to simply widen their privileges. Attacks have been classified according to different characteristics, e.g., by the privilege level that the attacher can gain, by type of actions the attacker performs, or by the technique utilized. A more precise taxonomy of attack types can be found in the work of Arturo Servin [Ser05], and a more thorough analysis of attacks will be carried on in Chapter 2.
In the last years, software engineers have become aware of the risks and security issues associated with the design, development, and deployment of network-based software [VM01] because damages and losses deriving from attacks have been increasing significantly. However, we are still far from a comprehensive solution to the security problem.

1.1.3 Losses and Damages Caused by Security Problems

The number of reported vulnerabilities have been increasing constantly, as reported by the *CERT* [CER06a]
Every year, starting from 1996, the *CSI* (Computer Security Institute) with the participation of the San Francisco *FBI* Computer Intrusion Squad conducts a survey of security incidents and practices in industry, government, and academia.
The 2006 *CSI/FBI* (Federal Bureau of Investigation) Computer Crime and Security Survey [CSI06] reports that the estimated average loss per respondent is about $170,000, which is an impressing amount even though it shows a decreasing trend with respect to previous year (about $200,000)
The top four categories of losses as reported by the survey are:

1. viruses;
2. unauthorized access;
3. laptop or mobile hardware theft; and

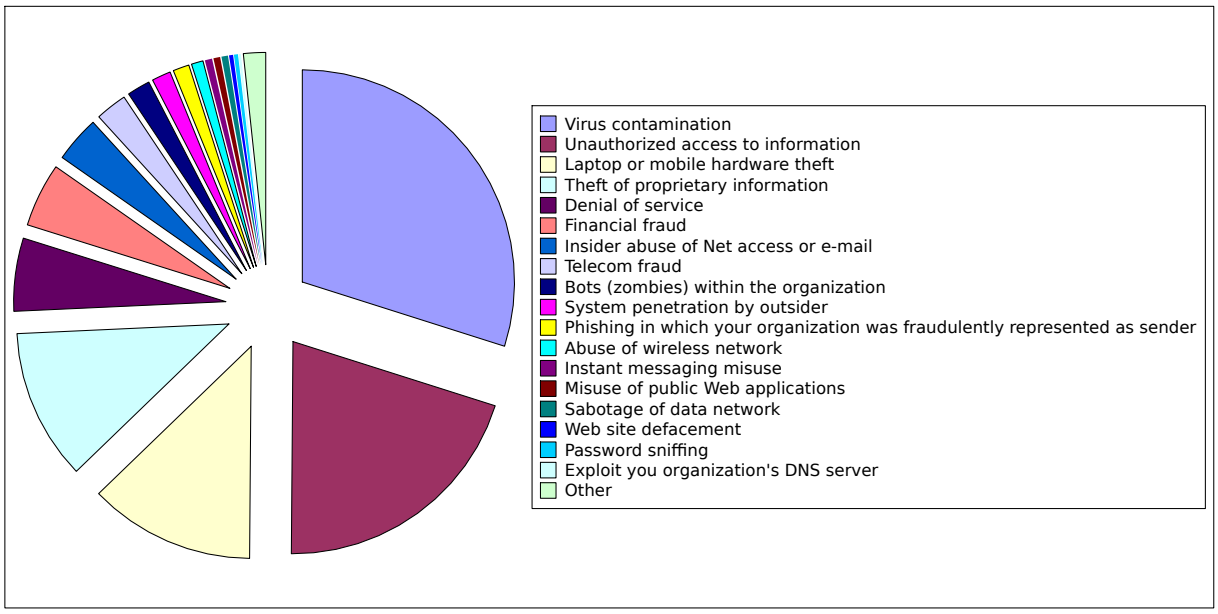


Figure 1.1: Dollar amount losses by type. (Source : the 2006 *CSI/FBI* Computer Crime and Security Survey [CSI06])

4. theft of proprietary information.

The four previous items accounted for nearly three-quarters (74.3 percent) of the total losses as can be seen from Figure 1.1
From those data emerges that security is becoming a concrete concern in today's organizations. Security costs a lot to companies,and vulnerabilities represent risks that can possibly become enormous losses. Therefore, they need to be mitigated in the most effective way.
An extensive documentation on facts and statistics about computer security, viruses, and loss of data due to computer complexity and/or viruses can be found here [Ans04]

1.1.4 Countermeasures to Attacks

In the previous sections we have seen the extent and impact of security vulnerabilities in networked systems. It could be argued that since the causes of most security problems are well known, the solution to those problems would be simply to avoid those causes to occur. This is somehow true, but infeasible. In a perfect world, vulnerabilities simply would not exist. In order to solve, once for all, all the security problems it would be sufficient to enforce a few steps, such as:

- strong authentication of both services and users;
- reliable authorization/access control;

- effective abuse control;
- secure design of protocols, operating systems, and applications;
- bug-free implementation of protocols, operating systems, and applications;
- perfect code policy;
- strong policy enforcement.

Recently the idea of reconsidering the software maintenance policies with security in mind has been proposed more than once.
For instance, the *CERT* Secure Coding Initiative [CER06b] works with software developers and software development organizations to reduce vulnerabilities resulting from coding errors before they are deployed by providing rules and recommendations for secure coding practices and libraries designed with security in mind.
Unfortunately, the feeling is that these remarkable initiatives will take some time to catch up with the real world, where effective security protections are almost always not deployed. The truth is that administrators do not always keep up with vendor updates/patches, sites do not monitor or restrict access to their internal hosts, organizations do not devote enough staff/resources to improve and maintain security (e.g., user education), sites do not implement policies (if they have one), and infrastructure service providers are driven by market/service, not security.
Users, on their side, insist on using flawed applications (e.g., mail readers that automatically execute attachments) and beta services with poor configuration management.
From the above, one can easily deduce that a simple prevention practice cannot be effective in this context.
Even though the consciousness about security problems seems to be increasing in the last years, as the *CSI/FBI* Computer Crime and Security Survey [CSI06] reports [1] a way to detect and nullify security risks is needed. Here is where intrusion detection comes into play.

1.1.5 Intrusion Detection

Intrusion detection is the research field that study how to detect and possibly nullify incorrect or anomalous activity that could jeopardize the integrity of a computer system. Intrusion detection research starts from understanding what are the most common attack techniques utilized to violate the security of computer systems [MSK01]. The second step is

[1] This year's questionnaire asked: which techniques does your organization use to assist in the evaluation of the effectiveness of its information security. 82 percent of respondents report that their organizations use security audits conducted by their internal staff and the use of the other techniques, such as penetration testing, automated tools, security audits by external organizations, e-mail monitoring software and Web activity monitoring software are also widely used for evaluation (58 to 66 percent of the respondents). From the *CSI/FBI* Computer Crime and Security Survey 2006 [CSI06]

No one will do that!	We know its the default, but the administration can turn it off
Why would anyone do that?	
We've never been attacked	If we don't run as administrator, stuff breaks
We're secure: we use cryptography	But well slip the schedule
We're secure: we use *ACL*s	It's not exploitable
We're secure: we use a firewall	But that's the way we've always done it
We've reviewed the code, and there are no security bugs	If we only had better tools...

Table 1.2: "Ridiculous Excuses We've Heard" from Writing Secure Code

to be informed about the current approaches to preventing such attacks [Bis02]. The third step is to understand and deploy technologies that support the detection of attacks [NN02].
Originally, system administrators performed intrusion detection by sitting in front of a console and monitoring user activities. They detected intrusions by noticing, for example, that a vacationing user is logged in locally or that a seldom-used printer is unusually active. Although effective enough at the time, this early form of intrusion detection was ad hoc and not scalable [Sto].
The next step in intrusion detection involved audit logs, which system administrators reviewed for evidence of unusual or malicious behavior. Administrators mainly used audit logs as a forensic tool to determine the cause of a particular security incident after the fact. There was little hope of catching an attack in progress.
As storage became cheaper, audit logs moved online and researchers developed programs to analyze the data. However, analysis was slow and often computationally intensive, and, therefore, intrusion detection programs were usually run at night when the systems user load was low. Therefore, most intrusions were still detected after they occurred.
In the early '80s, researchers developed real-time intrusion detection systems that reviewed audit data as it was produced. This enabled the detection of attacks and attempted attacks as they occurred, which, in turn, allowed for real-time response, and, in some cases,

attack preemption. More recent intrusion detection efforts have centered on developing products that users can effectively deploy in large networks. This is no easy task, given increasing security concerns, countless new attack techniques, and continuous changes in the surrounding computing environment.
In the future, the importance of intrusion detection will grow. Fast response to attacks is becoming more and more fundamental since today networks attacks can spread very fast and easily[SPW02].
For example, at the time of this writing, a zero-day vulnerability has been found in the handling of animated cursor in various versions of Microsoft Windows (Windows 2000, XP, 2003 and Vista according to CERT). The vulnerability is caused by the lack of a boundary check in a copy operation in the routines that handle animated cursors and can be exploited by tricking a user into visiting a malicious website using Microsoft Internet Explorer or opening a malicious e-mail message. Successful exploitation allows execution of arbitrary code. This means that virtually everyone who uses Internet Explorer on Windows, that is, the majority of Web surfers, is at risk.

1.2 Motivation

The present work contributes an analysis of the feasibility and a proof-of-concept implementation of a parallel, stateful intrusion detection system for high speed networks.

1.2.1 The Need for High-speed Processing

NIDS (Network-based Intrusion Detection Systems) perform security analysis on packets obtained by eavesdropping on a network link. The constant increase in network speed and throughput poses new challenges to these systems. Current *NIDS*s are barely capable of stateful real-time traffic analysis on saturated Fast Ethernet links (100 Mbps). As network technology presses forward, Gigabit Ethernet (1000 Mbps) has become the de-facto standard for large network installations. In order to protect such installations, a novel approach for network-based intrusion detection is necessary to manage the ever-increasing data volume.
Network speeds have increased faster than the speed of processors, and therefore centralized solutions have reached their limit. This is especially true if one considers in-depth, stateful intrusion detection analysis. In this case, the sensor has to maintain information about all attacks in progress (e.g., in the case of multi-step attacks) or they have to perform application-level analysis of the packet contents. These tasks are resource intensive and in a single-node setup may seriously interfere with the basic task of retrieving packets from the wire.

1.2.2 Parallelizing IDS and Multi-step Attacks

To be able to perform in-depth, stateful analysis it is necessary to divide the traffic volume into smaller portions that can be thoroughly analyzed by intrusion detection sensors. This approach has often been advocated by the high-performance research community as a way to distribute the service load across many nodes. In contrast to the case for standard load balancing, the division (or slicing) of the traffic for intrusion detection has to be performed in a way that guarantees the detection of all the threat scenarios considered.
A number of attacks are multi-step; for instance, to perform a *DNS* spoofing attack it is necessary for an attacker to intercept a *DNS* request sent by the victim and send a forged packet with a malicious name-*IP* association with the same identification number of the captured one. This packet has to reach the victim *before* the legitimate response.
In order to detect this attack, a *NIDS* has to detect:

1. the request,
2. the forged reply,
3. the legitimate reply,

in this order. Therefore, the *IDS* (Intrusion Detection System) needs to keep some representation of the attack state.
If a random division of traffic is used in order to distribute the load, sensors may not receive sufficient data to detect an intrusion, because different parts of the manifestation of an attack may have been assigned to different slices. Therefore, when an attack scenario consists of a number of steps, the load balancing mechanism must assure that all of the packets that could trigger those steps are sent to the sensor configured to detect that specific attack[2] [KVVK02].

1.3 Contribution of the Present Work

The present work is aimed at developing and analyzing a novel model for parallel stateful intrusion detection that is able to scale in order to keep up with the pace of high-speed network links.
More precisely, in this work we make the following contributions:

- We introduce a novel architecture for the parallel matching of stateful network-based signatures.
- We present a novel algorithm that allows for the detection of complex, stateful attacks in a parallel fashion.

[2]This is a simplification carried out for the sake of clarity, a more in-depth analysis will be carried out in Chapter 4

- We provide a precise characterization of the bottlenecks that are inherent to the parallel matching of stateful signatures in the most general case.
- We developed optimizations to reduce the impact of these bottlenecks and improve the performance of parallel detection.
- We describe a demonstrative, yet working, implementation of the system based on the `Snort` intrusion detection engine.
- We provide an evaluation of the implemented system on a real-world testbed.

Chapter 2

Intrusion Detection

Intrusion detection is the process of analyzing a stream of events in order to identify evidences of attack. Intrusion detection can be applied to different domains (e.g., the logs produced by web server or the network packets transferred on a network segment) and can be based on different techniques.
Before going through an in-depth analysis of intrusion detection and intrusion detection systems, we give some examples of intrusions and attacks in general.

2.1 Intrusions and Network Attacks

Intrusion is the attempt by an attacker to access or manipulate restricted information and services or, to prevent legitimate clients to do so (i.e., by compromising a system).
According to the work of Anderson [And80], continued by Axelsson in [Axe00, Axe99], attacks can be classified on the basis of two criteria, that is, whether or not the user is authorized to access the computer system and whether or not she can access a particular resource on the system. This categorization implies that we can divide intruders in three classes:

External Penetrators: those who are not authorized to access the system.

Internal Penetrators: those who are authorized to access the system but are not authorized to access the information they have accessed. This class can be further divided in:

Masqueraders: those who operate under another user's identity

Clandestine Users: whose who evade logs and audits (usually authenticating the system as an administrator)

Internal penetrators are the culprits of most of the security breaches [CSI06].

Misfeators : those who use their privileges maliciously.

From the perspective of the goal that the intruder attempts to accomplish, attacks can be roughly categorized in four categories:

- information gathering;
- unauthorized access;
- disclosure of information;
- denial of service.

From the point of view of the technique utilized in the attack, we can enumerate:

Network Attacks that focus on the networks themselves, rather than simply using them as a communication channel. The descriptions of some network attacks follows.

Sniffing is the technique at the basis of many attacks. The attacker sets her interface in promiscuous mode and can access all the traffic in a network segment.

Since many protocols (*TELNET*, *FTP*, *HTTP*, *POP*) transfer information in clear text, it is possible to collect sensible data by sniffing the network. There are plenty of tools, most of them in the public domain, to setup sniffing attacks.

Spoofing is an attack in which one person or program successfully masquerades as another by falsifying data and thereby gaining an illegitimate advantage. For instance in an *IP* spoofing attack, a host impersonates another host by sending a datagram with the address of some other host as the source address.

IP spoofing is used to impersonate sources of security-critical information (e.g., a DNS server or a NIS server) and to bypass address-based authentication.

Hijacking is an attack in which an intruder interposes herself in a legitimate conversation and steals the role of one of the actors. It can be done at different abstraction levels such as link (*ARP*), transport (*UDP*, *TCP*) and even at the application level (*HTTP* and *NIS*).

Man-in-the-middle is an attach that can be considered a two-sided hijacking, since the attacker interposes herself in between a conversation between two actors, leaving the actors unaware of the interception. The perpetrator of a man-in-the-middle attack usually can access and modify all the information exchanged by the communication partners.

Fragmentation is an attack whose goal is to cause troubles during the reassembling stages on the receiving host. Bugs (or features) in fragmentation handling can be exploited to cause *denials of service* (e.g., when the size of the reconstructed packed exceeds the memory available) or to evade firewalls and intrusion detection systems (if the packet reconstruction policy differs from the one used by the end-host, and, therefore malicious packets can be seen as legitimate ones by the *IDS* under certain conditions [PN98]).

Denial of service is a class of attacks in the attacker compromises or even tears down a service by means of exhausting the resources of the servers she is attacking. There are many known ways to do that: *SYN* flood, Ping of Death, *UDP* storm, etc.

Reconnaissance is an attack that tries to identify and fingerprint the machines that the attacker wants to compromise and the operating systems running on them. This is done by means of different types of *scan* such as *TCP* (connect, half-open, *FIN*, *ACK*, idle scan) and *UDP* portsweeps. A more precise analysis of reconnaissance techniques will be provided in Section 3.2.1

Local Attacks are possible when the attacker has already an account on the system she wants to compromise, i.e., the attacker is an *internal penetrator*. The vast majority of the local vulnerabilities in *UNIX* systems exploit *SUID* (Set User ID)-root programs to obtain root privileges. There are many attacks based on this idea: an indicative, yet non all-encompassing, list follows.

Environment attacks: in those attacks, the attacker gains restricted privileges maliciously modifying environment variables.

Input arguments attacks: these attacks exploit the lack of size-checking and sanitization in user-provided data to perform overflows, command injections, or directory traversal attacks.

Symlink attacks: these attacks leverage the fact that some applications that create temporary files for logging/locking do not test if the file already exists or whether is a symbolic link.

Race condition attacks: in this case the attacker races against the applications by exploiting the gap between testing and accessing a resource. The whole family of the *TOCTTOU* (Time-Of-Check-To-Time-Of-Use) attacks is based on this idea.

File descriptor attacks: these attacks exploit the fact that sometimes *SUID* applications open files to perform their tasks and fork external processes. If the *close-on-exec* flag is not set, the new process will inherit the open file descriptors of the original process. If the spawned program is interruptible, or worse, is interactive, a malicious user can use the file descriptors of the father process and write to them.

Format string attacks: this is a complicate attack based on the simple idea that the C **printf* functions are not type safe, and, if the arguments are interpreted as a format specifier, it is possible to write arbitrary values in the process memory by providing a carefully crafted format string.

Overflow attacks: these attacks encompass more a common usage pattern than a technique *per se*, but we treat them separately since they are one of the most popular type of attacks. The lack of boundary checking is one of the most common mistakes

in C/C++ applications. Buffer overflows can be exploited both locally and remotely and can modify both the data flow and the control flow of an application. Recent tools have made the process of exploiting buffer overflows easier if not completely automatic. Much research has been devoted to finding vulnerabilities, design prevention techniques, and developing detection mechanisms. The most common overflows attacks are:

stack-based overflows, such as return address overflow, jump into libc, off by one, longjump overflows, and array overflows.

heap based overflows that exploit in-band control structures for dynamic memory management and C++ vtables.

integer overflows caused by unexpected results when comparing, casting, and adding integers.

Web Attacks are attacks against web-based authentication, authorization, and *HTTP* protocols implementations. Beyond the classical techniques of eavesdropping and brute-force guessing, examples of web-specific attack are:

***SQL* injections,** where arbitrary *SQL* commands are injected into database servers by exploiting poor sanitization of client-provided data. Similar concepts are used to inject *LDAP* [Fau05] and *XPath* [Kle] based application as well.

session fixation, which forces the user's session ID to a known value. The ID can be a fixed value or can be obtained by the attacker through a previous interaction with the vulnerable system [Kol].

cross-site scripting, where the privacy of a client is violated by executing commands on the client impersonating critical servers.

web spoofing, also known as a man-in-the-middle attack, which exploits url rewriting techniques to hijack the user browsing session.

response splitting and request smuggling, which are attacks to *HTTP* protocol implementations.

The aforementioned categorizations are not in contrast with each other and are aimed at characterizing attacks from different points of view. It is remarkable to notice that many real-world attacks do not fit in any or the categories above but, instead, use a combination of those attacks.

2.2 A Typical Intrusion Scenario

Intruders usually carry on their attack in more than one step, starting from a process of information gathering on the target through the very core of the attack.
More precisely, as reported in [Inf06], a typical attack scenario might be made of the following steps:

Information gathering. The intruder starts seeking as much information as possible on her target, operating as stealthy as possible. Some of the most used information gathering techniques are *DNS* lookups or e-mail address harvesting. Social engineering is also widely-used a technique that make use of human weaknesses to gather information [MS03].

Further information gathering. In an attempt to gather more information an attacker will usually perform ping sweeps, port scanning, and check web servers for vulnerable *CGI* scripts. The intruder will try *OS* and application fingerprinting to discover which software versions of applications are running on the target host.

Attack. Having gathered a list of possible loopholes, the intruder will start trying out different attacks against the system. Many tools that try automatically well-known attacks can help in this step. More sophisticated tool such as *nessus* [Nes01] can even automatically gather information on the target, therefore avoiding previous steps, in order to automatically suggest to the intruder which attacks are likely to succeed.

Presence establishment. After a successful intrusion, an attacker will try to hide her tracks deleting log files and setting up a covert channel in order to maintain control of the attacked machine. An attacked machine is often used as a trampoline to attack another one. Many attacked and compromised machines can be turned into *zombies* and or *bots* and made part of large-scale bot-nets.

A growing trend among intruders is to randomly scan Internet addresses, looking for a specific hole or number of holes. While in the 80s most intruders were skillful experts, with individually-developed methods for breaking into system, today anyone can scan Internet sites using readily-available tools and exploit scripts that capitalize on widely-known vulnerabilities.
Figure 2.1 illustrates the relationship between sophistication of attacks and intruder knowledge from 80s to present.

2.2.1 Countermeasures

Preventing attacks from being successful is the main focus of computer security. As reported by Mamata D. Desai in [Des02] intrusion detection can be deployed at different levels, with different goals:

Prevention blocks attacks from their very initial steps. Prevention techniques include installing firewalls, keeping software up-to-date and using *ACL*s.

Preemption operates against likely threats prior to an intrusion attempt.

Deterrence discourages an intrusion by making her way into the system harder or decreasing her perceived gain of a positive attack outcome.

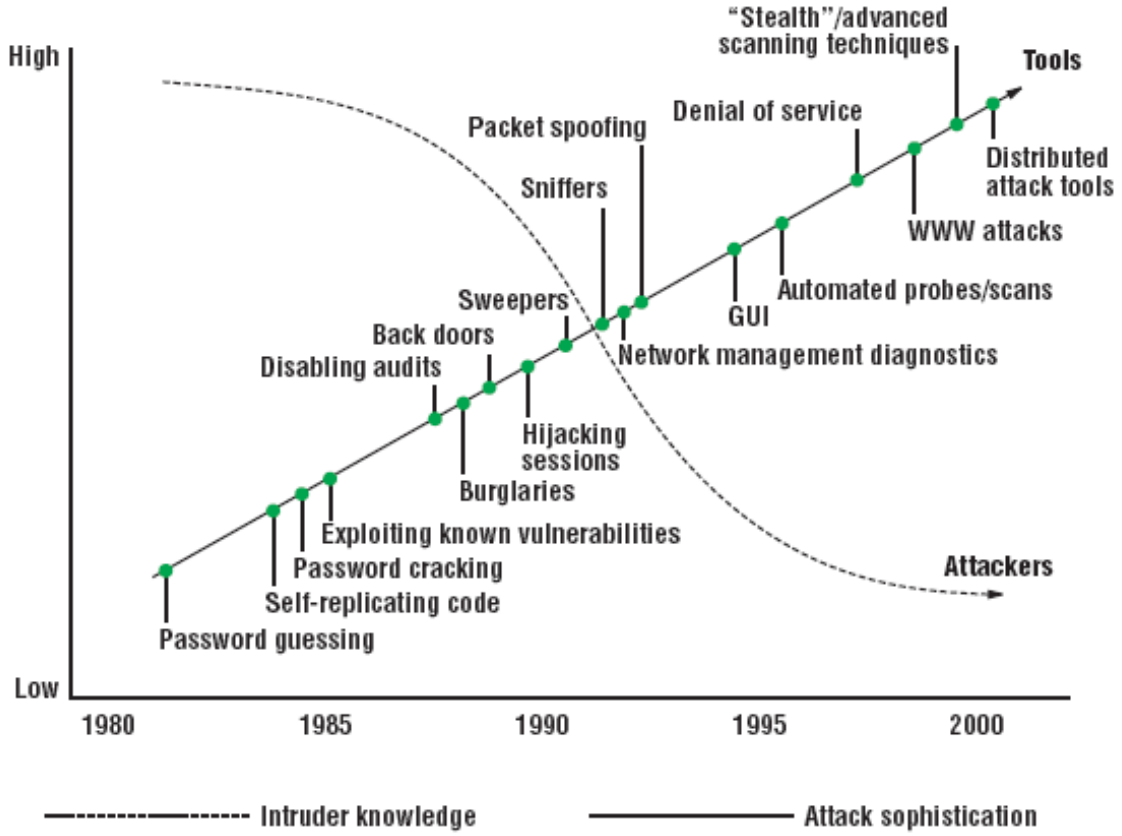


Figure 2.1: Attack sophistication versus intruder technical knowledge. Source: *CERT*, see: [JMA00]

Deflection lures the intruder into believing that her attack was successful in order to study her moves (*fishbowling*) or to deviate her goals.

Active response counters an intrusion in real time while the attack is being attempted. Recently systems [Roe] have been proposed that are able to catch attacks in their early steps and setup rules to neutralize them[1]. This kind of system are called *IPS* (Intrusion Prevention System), because they prevent the attack from being successful even though they do not prevent the whole attack from start to end.

Detection it the most widely used technique together with prevention. Detection systems

[1]This is made possible by the fact that packets are diverted from the normal flow through the kernel stack and processed by the intrusion detection/prevention system before being handed out to applications. In this fashion, malicious packets can be filtered out and never reach applications.

monitor activities and raise an alert if some action is categorized as anomalous or malicious. This kind of systems do not counter directly the attack but rely on somebody who takes care of the alert and operates accordingly.

2.3 Intrusion Detection Systems

IDS (Intrusion Detection System)s are an important instrument for the protection of computer systems and networks. They support mechanisms to perform automatic threat recognition and they can also provide possible defenses.

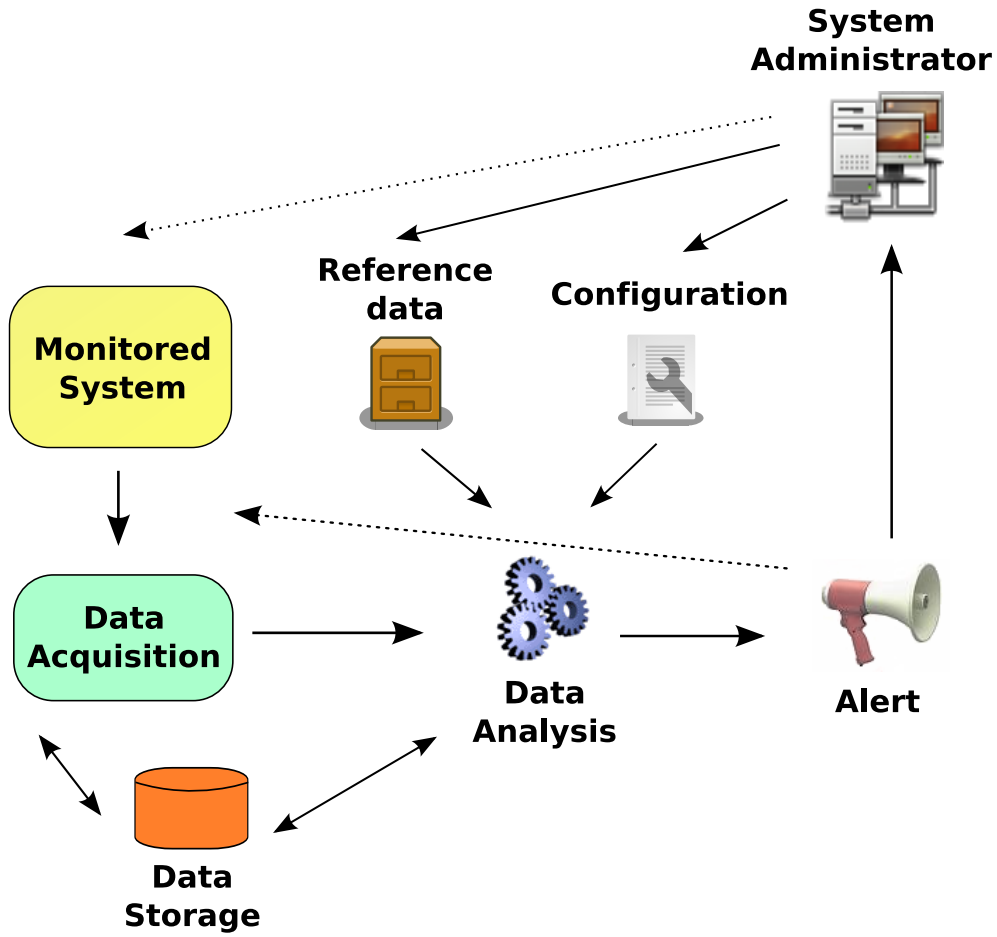


Figure 2.2: General intrusion detection system structure. Modified from original version available here: [ST]

The general architecture of an *IDS* is outlined in Figure 2.2. Data are collected from the monitored system and analyzed by the main detection engine. Both the detection engine and the data collector can use databases and filesystems to store information temporary

or permanently.
The detection engine performs the data analysis and its behavior is defined by configuration rules and possibly also by reference data. Once an alert is raised, depending on the system setup, different actions can take place and can affect virtually all the components of the system.
Some of the interactions in the figure are optional (dashed lines), such as the possibility for both the system itself and the network administrator to modify the monitored system in response to an alert. If the *IDS* reacts automatically to an alert by modifying the monitored system (i.e., trying to prevent the attack to be successful) then the system is known as an *IPS*. In usual deployments, the *IDS* simply alerts the system administrator who will perform the appropriate defensive actions.

2.3.1 Anomaly detection and Misuse Detection

A common way to characterize *IDS*s is to examine the models that they use as the basis for detection. If the models are used to specify what the "normal behavior" of an application is, then the system is usually called *anomaly-based*, because an attack will be detected as an event (or a series of events) that does not fit the system's idea of what is "normal." If the models are used to specify what is the manifestation of an attack, then the system is called *misuse-based*, and the models are often referred to as *signatures*.

Anomaly Detection Systems

Anomaly-based intrusion detection techniques depict an intrusion as a deviation from "normal behavior". The so called "normal behavior" needs to be defined, or learned. The first step in deploying anomaly-based detectors is therefore building up a knowledge base about the monitored domain (e.g, a flow of packets in a network link, a sequence of system calls, a sequence of *HTTP* request, etc.) usually derived from long-term system observations and define thresholds beyond which the system activity is considered abnormal. These systems can be learning-based or specification-based, depending on the way in which they determine what is the normal behavior of a system.

Misuse Detection Systems

Misuse detection systems define what an intrusion is in the observed system and uses a knowledge-base of system vulnerabilities and patterns of known security violations - so-called signatures - as a model of the intrusive process. Misuse detection systems identify evidence of attacks by searching for patterns of known attacks in the data collected from the monitored system. If an action does not precisely match a signature it is deemed not malicious and considered acceptable.

Comparison Between Misuse-based and Anomaly-based Intrusion Detection

Both anomaly-based and misuse-based systems have pros and cons.

Anomaly-based systems are able to detect previously unknown attacks, new exploits and unforeseen vulnerabilities, but they traditionally produce more false positives, which are erroneous detections. This is because anomaly detection systems discriminate between intrusions and legitimate activities based on thresholds (set or learned) that characterize the normal behavior, which need system-specific, hard-to-provide, tuning.
Misuse-based systems, on the other hand, are usually faster and have a low false positive rate, but cannot detect attack for which they do not have a description (such as 0-day exploits[2]), and, therefore they depend on continuous updating of the signature database. The advantage of this approach, in addition to the fact that the potential for false alarms is very low, is the fact that the detailed contextual analysis makes it easy for the security officer to take preventive or corrective action, while, instead, is not easy to trace the identified anomaly back to the causing attack. This makes it difficult to automatically initiate countermeasures.
Because of their low false positive rate and their higher performance, and given that they are simpler to implement and to configure, misuse-based detection approaches are the basis for the majority of the existing network-based intrusion detection systems. Unfortunately, as the the speed of network links increases, keeping up with the pace of events becomes a real challenge.
In order to overcome the shortcomings of both approaches, as pointed out by Desai in [Des02], a new breed of systems is being developed that combines the low false-positive rate of signature-based intrusion detection systems and the ability of anomaly detection systems to detect novel unknown attacks. These detectors leverage their knowledge of both intrusive and normal behaviors and use that knowledge to classify events. The results reported by Hwang et al. in [KHQ] using such kind of systems are very encouraging.

Rule Specification

Signature-based intrusion detection systems need some way to describe their models of attack and which action the system has to accomplish when a signature is matched.
Usually [EVK00, sno04] a rule is described as composed of a predicate, a state, and an action. The predicate describes the constraints on the values of the event and state attributes. If the predicate evaluates to true for an event then the action is triggered. The action consists in modifying the rule state or raising an alert (that is, generating a new event for further processing). The rule maintains state in order to keep track of the relevant parts of the state of the system. For example, a counter may be used to count the probes sent that are part of a port scan.
The above considerations give us a general framework in which we can define attack languages even though they do not outline any model for the semantics of attack signatures. A number of languages are proposed in the literature, which differ in the characteristics of the signatures that can be described.

[2] 0-day exploits are released before, or on the same day of the release of a vulnerability. The term derives from the number of days between the public advisory and the release of the exploit. Since 0-day attacks are generally unknown to the public, it is often difficult to defend against them. See: [Wik07d]

An important step towards defining a model for semantics of attacks signatures has been done by Meier in [Mei04] inspired to the one proposed by Zimmer et al. in [ZU99] for event triggering in active databases. As the author says, "the presented model represents a kind of a checklist for the development of a signature specification language or for the comparison of existing signature specification languages."
The model we present in Chapter 4 loosely defines the semantics of the signatures that our system will be able to handle. An effort towards a more formal definition of the semantics of attack signatures of the parallel model that we present will be taken into consideration as future work in Chapter 7.

2.3.2 Host Based and Network Based IDS

IDS can be further classified from the point of view of the domain of operation as *HIDS* (Host-based Intrusion Detection System) and *NIDS* (Network-based Intrusion Detection Systems)

Host-based Intrusion Detection Systems

Since they are simpler to implement, *HIDS*s have been the first type of intrusion detection systems developed and deployed in a large scale when networks were not as prevalent as now. *HIDS*s analyzed audit logs looking for suspicious activities and are mostly used in the aftermath of attacks to understand weaknesses of the system and to prevent future attacks.
Today, *HIDS*s are more responsive to attacks even though they are still based on the same concepts as before. Typically, *HIDS*s monitor system, events and security logs on *Windows* and *UNIX* environments. When a log file changes the *IDS* compares the new log entry with known signatures, and, if an attack is detected, the system raises and alarm or puts in action defensive responses.
*HIDS*s include other mechanisms to monitor the system status such as checking key files and executable via checksums at regular interval or listening to known port activities.
According to Desai, Host-Based *IDS*s are characterized by the following advantages:

- verify success or failure of an attack;
- monitor specific system activities;
- detect alerts that network-based systems miss;
- are well-suited for encrypted and switched environments;
- perform near-real-time detection and response;
- require no additional hardware;
- have a lower cost per entry.

Host based systems also have disadvantages; for example, they affect the performance of the host being monitored.

Network-based Intrusion Detection Systems

*NIDS*s monitor traffic on the network. They capture all the packets crossing the networks by placing they interface in promiscuous mode then pass those packet to a detector that performs signature matching on them. Signatures usually describe combinations of condition on packet headers and payload such as the presence of the word "cgi-bin" in the payload or the evidence that packets are directed to port 80.
The strengths of *NIDS*s are:

- lowers cost of ownership;
- detection of attacks that host-based systems miss;
- are more resistant to tampering with evidence.
- perform real-time detection and response;
- detect of unsuccessful attacks and malicious intent;
- do not impact the monitored hosts.

However, network-based *IDS*, are vulnerable to evasion attacks [PN98], where an attacker desynchronizes the views of the *IDS* and the monitored system.

Comparison Between Host-based and Network-based Intrusion Detection

Both *HIDS*s an *NIDS*s have advantages and disadvantages. Host-based audit records are qualitatively better suited for intrusion recognition. On the other hand, *NIDS*s can be less costly if installed at a central place of the protected net without influencing the efficiency of the end systems.
More precisely, host-based solutions require the configuration of each individual host in the protected environment and they can be used on a single operating system at one time. On the other hand, being based on the analysis of standard network protocols common to all operating systems, *NIDS*s can monitor a heterogeneous set of hosts in the same time.
Another problem with *HIDS*s is that the efficiency of the hosts is impaired by the event logging and its analysis and event logging itself can be delayed by the *OS*, slowing down the responsiveness of the *IDS*. *NIDS*s are not concerned with this problems, since they monitor passively broadcast networks, they do not impair performance of hosts, and can respond to events almost instantaneously.
A major problem from the security point of view of *HIDS*s is that since the *IDS* lives on the host, if the host is compromised the audit trials can be compromised by an attacker that breaks into it. Theoretically speaking when an attack is successful the *IDS* has failed its purpose and becomes useless, but, practically, having the possibility to check for evidence

of attacks in a *post-mortem* setup is still useful. Since *NIDS*s are invisible to the intruder, they cannot be turned off, and the data they collect cannot be compromised.
As a final remark, many of the more seriously documented cases of computer intrusions have utilized a network at some point and the network itself, having become a critical point of the environment, can be the target of attacks that *NIDS*s can monitor while *HIDS*s cannot.
To sum it up, due to the comparatively good recognition accuracy, *NIDS*s are the most popular and commonly used approach at present because of all the aforementioned reasons and moreover because they are easy to install and configure.
Both *HIDS*s an *NIDS*s are facing the increasing performance of both networks and end systems. As pointed out by Meier et al. [MMK05], this leads to an increasing audit data volume that has to be analyzed. In addition, new attacks and vulnerabilities imply that the number of signatures to be analyzed increases as well, and, therefore, the efficiency of *IDS*s becomes critical. Today, systems under heavy load discard some of the log data that should be processed. This leads to inaccuracies in detection and makes it impossible to perform effective counter measures.

2.3.3 IDS Evaluation

The principal criteria utilized to evaluate the performances of an *IDS* are:

responsiveness, which discriminates between real-time and offline systems. Real-time systems can respond to the evidence of attacks as soon as they acquire and analyze the information from the monitored system. Therefore they can react and have the possibility to foil still ongoing attacks. Offline systems, instead, analyze batches of audit logs and can discover attacks only already occurred.

response time, which describes ho an *IDS* reacts to an event. *IDS*s can behave actively or passively with respect to attack evidence. Active responses encompass the possibility for the *IDS* to operate on the monitored system in order to neutralize the root cause of the detected attacks, while a passive *IDS* will simply report to the system administrator that an anomalous event has occurred.

interoperability, which describes the ability of an *IDS* to interact with other *IDS* or network devices in order to exchange data (for instance, an *IDS* could be able to talk to the firewall of the network and delete routes to prevent ongoing intrusions).

coverage, which characterizes the attacks that an *IDS* can detect under ideal conditions. For instance for a signature-based *IDS* the coverage is simply the number of signatures.

accuracy, which is one of the most important feature of an *IDS*. Accuracy measures the ability of the system to successfully detect attacks, that is the *TPR* versus the rate of false alarm, or *FPR*. An ideal *IDS* should detect all the attacks (100% *TPR* (True Positive Rate)) and never raise a false alarm (0% *FPR* (False Positive Rate)).

This concept can be clarified and visualized using a *ROC* (Receiver Operating Characteristic) curve[3]. A *ROC* curve can be used to compare performance of different *IDS* as their operating point (i.e. how their rules are tuned) is varied. An ideal *IDS* would have a *ROC* curve wrapping around the upper left corner of the space, while the baseline to compare with for *ROC* analysis is a random classifier (i.e. the poorest possible accuracy), that is represented by the bisector of the *ROC* space.

A more in-depth overview on intrusion detection evaluation can be found in the *NIST* Interagency Report n. 7007 [oSTIPM03].

2.4 Parallel Intrusion detection

As discussed in Chapter 1 as networks become faster and the number of signatures[4] increases, the processing time associated with intrusion detection become significant.
Therefore there is an emerging need for security analysis techniques that can keep up with nowadays' analysis demand. Many attempts to ameliorate the problem, under different perspectives, have been done and the most promising solution from the efficiency and scalability points of view has been found in parallelization [WF07].
Parallelization can occur at different levels of the intrusion detection system. In Chapter 3 we will take into consideration the current work on this particularly new research field. We will take into consideration present solutions to augment intrusion detection with parallelization.
We make a clear distinction between parallel *IDS*s and distributed *IDS*s. A distributed *IDS* utilizes distributed monitoring at various locations in a network and its goal is to improve the quality of detection by gathering more data. Our work will not focus on distributed systems but will try to propose a model and a parallel algorithm to improve the efficiency of intrusion detection on high-speed networks.

[3] A ROC curve [Wik07b], is a graphical plot of the *TPR* versus the *FPR* for a binary classifier system as its discrimination threshold is varied.

[4] To give an example, in 2003 the `Snort` default ruleset was composed of 1500 signatures while as today (April 2007) there are about 8500 rules in the default ruleset.

Chapter 3

Present Work

3.1 Improving IDS Performance

The problem of keeping up with the fast pace of events in high speed networks has been addressed from different perspectives by current research in intrusion detection.
The commercial world attempted to respond to this need and a number of vendors now claim to have sensors that can operate on high-speed *ATM* or Gigabit Ethernet links. For example, ISS [ISS01] offers NetICE Gigabit Sentry, a system that is designed to monitor traffic on high-speed links. The company advertises the system as being capable of performing protocol reassembly and analysis for several application-level protocols (e.g., *HTTP*, *SMTP*, *POP*) to identify malicious activities.
Other IDS vendors (like Cisco [CIS01]) offer comparable products with similar features. TopLayer Networks [Top01] presents a switch that keeps track of application-level sessions. The network traffic is split with regard to these sessions and forwarded to several intrusion detection sensors. Packets that belong to the same session are sent through the same link. This allows sensors to detect multiple steps of an attack within a single session. Unfortunately, the correlation of information between different sessions is not supported. This could result in missed attacks when attacks are performed against multiple hosts (e.g., ping sweeps), or across multiple sessions (e.g. *FTP* session uses two distinct connections). Very few research papers have been published that deal with the problem of stateful intrusion detection on high-speed links.
We can group the research focuses on the topic in two distinct approaches:

- enhance the throughput by ameliorating the bottlenecks of current systems, or
- deploy new systems that exploit parallelization to improve speedup.

In the following sections, we will take a look at both the approaches.

Optimizations on the Single Sensor Model

The tentative of improving current solutions has been proposed by many and leverages both hardware-based and algorithmic-based solutions. Hardware-based are most common

solutions used to achieve high performance.
*NIDS*s are currently directed towards custom network solutions involving *FPGA*s, *ASIC*s and the so called *Network processors*[1].
As pointed out by Colajanni et al. in their work [CM06], custom hardware cannot be considered a long term solution because only pushes further the maximum manageable throughput without giving a scalable solution. Moreover, custom hardware solutions are not flexible and cannot be easily extended by the end user.
In particular, many *ad hoc* hardware systems operate on single packets mostly performing pattern matching and do not provide support for complex *stateful* signatures, where the evidence necessary to detect an attack is spread across multiple packets at different points in time.
A great deal of research has been devoted to trying to increase the efficiency of pattern matching on packets. In this realm we can find the work of Lu et al. [LZL+06] in which a system to processes multiple characters per clock cycle using multiple parallel deterministic finite automata is proposed.
Their system is implemented in hardware by means of a *BCAM* (binary content addressable memory) and an embedded *SRAM*. The authors start from the evidence, confirmed by other [FV01, WF07], that the content matching phase dominates the overall runtime in today's *NIDS*s (e.g., this phase accounts up to the 75% in `Snort` 2.6.0). By employing an efficient representation of the transition rules in each *DFA*, their system significantly reduces the complexity using dedicated hardware and can handle the throughput of an *OC*-768 link (40Gb/s).
Although this approach seems very promising and the authors foresee that it could keep up with 100Gb/s data throughput, its major downside is that the matching is performed on a per-packet basis and it is not possible to extend the system to make it capable of dealing with complex stateful attacks (e.g., co-ordinated port scans, see Section 3.2.1).
Other solutions to improve the pattern matching capabilities of *IDS*s are software-based.
Sekar et al. [SGVS99] describe an approach to perform high-performance analysis of network data. They propose a domain-specific language for capturing patterns of normal and or abnormal packet sequences (therefore, their system encompasses both misuse and anomaly detection). Moreover, they developed an efficient implementation of data aggregation and pattern matching operations. A key feature of their implementation is that the pattern-matching time is insensitive to the number of rules, thus making the approach scalable to large number of rules.
The shortcomings in the work of Sekar et al. is that the processing of the packets is done on a central sensor that therefore needs to be able to keep up with the fast pace of packets on high-speed networks. An estimate of the maximum throughput that today's computers are able to handle is given by Luca Deri in [Der05]. He shows that by means of a memory

[1] Network processors are integrated circuits specifically designed for the networking application domain, i.e., to elaborate packet in packet-switched networks. Network processors usually have optimized functions such as pattern matching, key lookup (used for routing and switching lookup address), data bitfield manipulation, queue management and control processing [Wik07a].

efficiency/low latency technique[2] it is possible to achieve a Gigabit/s Ethernet speed (1.48 Mpps) in packet capturing. Techniques like the one described by Deri can be used in intrusion detection sensors to enhance the performance of a single matcher, but for *OC*-48 through *OC*-768 (40Gbps) links a single matcher is not enough.
Building up from the foundations laid by Sekar, but using a more general and implementation-independent -yet profitable, in real world experiments- approach, Meier et al. [MMK05] propose a method for system optimization in order to detect complex attacks.
They propose a method to reduce the analysis run time. This method focuses on complex, stateful, signatures. Therefore, it is a step forward with respect to the pattern matching optimization proposed by [LZL+06]. Starting from Petri-net-based modeling of attack signatures, they observed several structural properties of signatures that can be exploited to speed-up the detection process. This includes the avoidance of redundant evaluations of conditions by identifying common expressions as well as the separation between intra- and inter-event conditions. Furthermore, they claim that matching signature instances can be efficiently based on comparison operations of inter-event conditions by indexing existing signature instances using the values of their features and that the analysis efficiency can be improved by controlling the evaluation order of conditions. Moreover they developed and tested a prototype implementation of their system on a handcrafted worst-case scenario of trials data and discovered that it performs very well, compared to other system, such as STAT [EVK00] in terms both of runtime and memory usage.

The Parallel Approach

We now review some approaches that aim at parallelizing the existing architectures with particular emphasis on stateful processing.
Sommers et al. [SP05] propose a way to exchange state among sensors by serializing it into a file and exchanging the serialized version. Their approach is focused on providing a framework to transfer fine-grained state among different sensors in order to enhance the *IDS* with many features. As the authors state "these include coordinating distributed monitoring; increasing *NIDS* performance by splitting the analysis load across multiple *CPU*s in a variety of ways; selectively preserving key state across restarts and crashes; dynamically reconfiguring the operation of the *NIDS* on-the-fly; tracking the usage over time of the elements of a *NIDS* scripts to support high-level policy maintenance; and enabling detailed profiling and debugging."
Although the authors provide a way to exchange fine-grained state among sensors, no emphasis on high performance emerges from the paper. The method is not immediately responsive to attacks and cannot keep on with fast pacing traffic.
A number of approaches have been proposed to address the problem of matching stateful signatures on high-speed networks in a parallel fashion. In [KVVK02] a distributed ar-

[2]The system he proposes, called *ncap* is based on a commercial network adapter and dispatches packets directly from the *NIC* driver to userland bypassing kernel. The system can achieve wire-speed captures throughput leveraging a 64-bit *PCI*-X bus with memory-based interrupt, interrupt mitigation and prioritization, and enhanced performance and bandwidth.

chitecture was proposed to partition the traffic into slices of manageable size, which were then examined by a dedicated intrusion detection sensor. This slicing technique took into account the signatures used by the intrusion detection sensors, so that all the evidences needed to detect an attack were guaranteed to appear in the slice associated with the sensor responsible for the detection of that particular attack. Therefore, no communication among the intrusion detection sensors was necessary.
More precisely the overall architecture of the proposed system is sketched in Figure 3.1

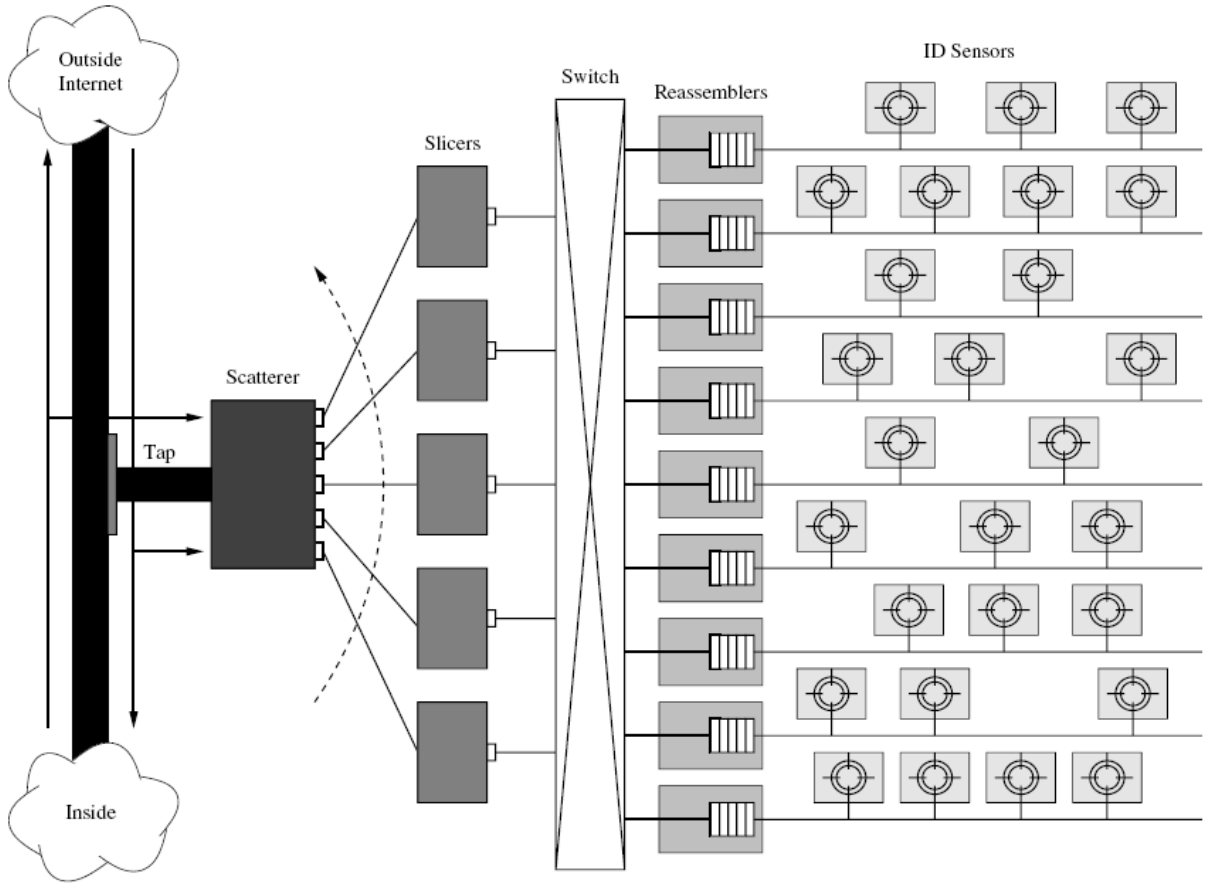
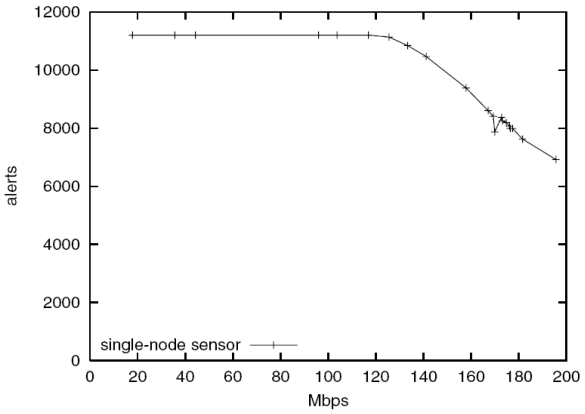


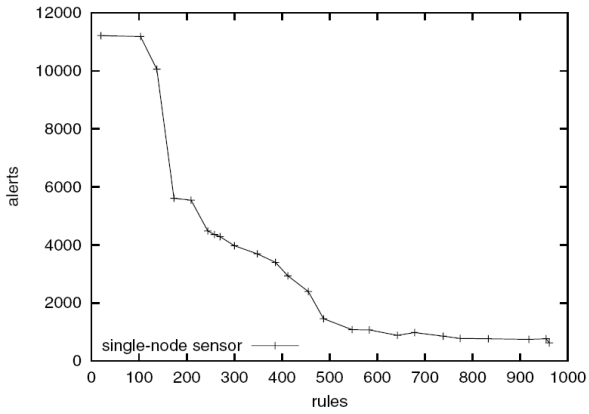
Figure 3.1: High-level architecture of the high-speed intrusion detection system. Source: [KVVK02]

The system consists of a network tap, a traffic scatterer, a set of m traffic slicer, a set of n traffic reassemblers, and a set of p intrusion detection sensors. The network tap monitors the traffic stream on a high-speed link and extracts sequences of link frames that are passed to the scatterer which, in turn, partitions the traffic into m subsequences in a round-robin fashion. Each slice is then processed by a slicer. The task of the slicers is to route the frames they receive to the sensors that may need them to detect an attack. This task is not performed directly by the scatterer because the frame routing may be complex and time

consuming, and, therefore, the scatterer could be overwhelmed by the high throughput. The traffic slicers are connected to a switch component, which allows a slicer to send a frame to *one or more* outgoing channels. All the sensors associated with a channel are able to access all the packets sent to that channel. The traffic reassemblers' task is to make sure that the packets appear on the channel on the same order they appeared on the high-speed link, since packets could have arrived to the channel taking different paths, over different slicers. Each sensor is associated with different attack scenarios (i.e., tries to match different attack signatures), which, in turn, are associated to different event spaces. The slicers have to ensure that all the packets relative to the event space matched by a sensor are forwarded to it.



(a) Parallel detection rate for increasing traffic levels



(b) Parallel detection rate for increasing number of signatures

The experiments proposed by Kruegel et al. shows how their parallel matching system outperforms a single *IDS* instance that easily gets overrun by the high throughput (Figure 3.1)

Unfortunately, even though the system was able to improve the overall performance of the detection process, it was not able to effectively partition the traffic in the case of complex stateful signatures. This is because, in general, the correct partitioning of the traffic is only known at runtime and therefore any approach that uses a static partitioning algorithm needs to over-approximate the event space associated with a signature.

In this work, we propose a novel technique to address this issue. Our technique is similar to the one proposed in [KVVK02] in that it uses a partitioning technique to reduce the traffic on a high-speed link to slices of manageable size, and, in addition, it uses a number of parallel sensors. However, in our approach the sensors are able to communicate through a high-speed, low latency dedicated control plane. By doing this, they can provide feedback to each other and synchronize their detection processes. As a result, it is possible to match complex, stateful signatures without having to over-approximate the event space.

Colajanni et al. in their work [CM06] enhance the previous architecture by providing dynamic load balancing mechanism for the traffic-distribution algorithm. A monitoring

system is added to the previously described architecture in order to trigger load balancing actions when some of the sensors risk to lose packets because overwhelmed by the traffic. In this situation the monitoring system triggers the migration of state associated with tracked connections from a sensor to another one, and, at the same time, instructs the traffic slicer to divert the traffic relative to those connections from the overloaded sensor to the target of the migration. In some sense this architecture mix the technique described by Sommers et al. in [SP05] with the architecture proposed by Kruegel et al. in [KVVK02].
Later, Wheeler et al. [WF07] proposed a taxonomy of *IDS*s that encompasses the previous techniques. More precisely the previous techniques are classified as "node level parallelization" since the sensor nodes are used as atomic entities. The architecture proposed by Kruegel et al. [KVVK02] is an example of data and function parallelization since different sensors have different rules loaded (i.e., perform different functions) and are fed with different traffic slices (different data). Wheeler et al. take into consideration other kinds of parallelization that they call "component" and "sub-component" level parallelization. In component parallelism, individual components of the *IDS* are isolated and given their own processing element. For instance, pre-processing and content matching can be put in separate processes (paying attention in ensuring data consistency in different steps). Sub-component parallelism refers to the parallelization of individual components of the *IDS*; for instance, the parallelization of the content matching proposed by Lu et al. [LZL+06] is an example of sub-component parallelization.
To complete the overview of present work on high-speed intrusion detection, we review some optimizations that improve the parallel *IDS*.
Xinidis et al. have proposed *ad hoc* techniques to improve the packet splitter that distribute the incoming traffic to sensors [XCA+06]. They argue that the splitter should be actively involved in analyzing the traffic and propose some enhancements. The authors propose to enhance the splitter with packet header inspection (since it is has a small *CPU*/memory footprint) in order to filter out early from the detection chain packets that surely cannot match any rule, such as, packets with no payload and with a header that does not match any header-based rule. Remaining packets are sent to sensors in order to be checked against the full set of rules (both header- and payload-based). Another technique employed by the authors is to smartly queue packets on the splitter in order to serve them to sensor in bursts of homogeneous sets. Sensors will gain from the similarity of packet features (i.e., header fields) because this will improve the locality of their memory accesses and therefore will reduce cache misses. Test showed that early filtering improves slightly the performance by lowering the load on the sensors (32% less packets to analyze, 8% increase in performance) and the locality buffer improved by 10% the overall performance (in particular, the throughput on the slowest link was improved by 14%).

3.2 Applications

We have taken into consideration a great deal of architectures and approaches to improve the efficiency of misuse detection system on high-speed networks but we still have given

little or no examples of what we want this system to be able to perform.
We can think the problem of stateful intrusion detection at different levels. A very simple form of state to be maintained for an *IDS* is the state associated with *TCP/IP* flows. As seen in the previous section, many systems are now "stateful" flow-wise, that is, the are able to detect attacks based on a particular sequence of packets of the same *TCP* connection. Thanks to the fact that is easy to partition a stream of packets in different flows (computing a hash on some header fields), a centralized system can be transformed into a parallel one made of sensors that process manageable slices of traffic with little or no effort.
Such systems work well in practice when they have only signatures whose scope is limited to single flows. But there are attacks that can be carried on exploiting multiple connections (we have seen the case of attacks against *FTP*, which employs two connections), but the event space of a signature can be approximated easily, making possible, again, parallelization by traffic slicing. Parallelization by traffic slicing works every time that an instance of every attack that needs to be detected has an event space that cannot span more than one slice. There are cases when the event space of a signature can be as big as the whole event space processed, or, in other words, one cannot easily avoid that different parts of the manifestation of an attack may be assigned to different slices. An example of such problematic attacks is the co-ordinated port scan.

3.2.1 Port Scan

As seen in Chapter 1, before an adversary tries to mount an attack against a target, she will usually try to perform some kind of information gathering in order to discover potential vulnerabilities [LFG+00]. As stated by the HoneyNet Project [Pro02],"Most attacks involve some type of information gathering before the attack is launched."
One of the most used techniques for information gathering is the port scan. A port scan is a technique employed to determine if a particular service is available on a predetermined host by observing the responses to carefully-crafted packets.
Panjwani et al. [pan05] have found that approximately 50% of attacks are preceded by some form of scanning activity. Catching a reconnaissance activity in its early steps therefore increases the possibility of preventing an attack from being accomplished.

Types of Port Scan

Port scans can be classified on the basis of the technique used by the attacker. We briefly review the various techniques drawing on the scanning capabilities of nmap [Fyo07], the state-of-the-art tool for network reconnaissance.

connect() scan is the most basic form of scan. The attacker initiated a full *TCP* connection (accomplishing the three-way handshake) to see if a particular port is accepting *TCP* connections.

***SYN* scan,** also known as "half open" scanning, is performed by sending a *TCP* packet with *SYN* flag set to test if a port is open. A received *SYN/ACK* packet informs the

attacker that the scanned port is open, while a *RST* packet received means that the port is not listening.

***ACK*, *FIN*, Xmas, Null scans** use a packet crafted with a combination of set flags to probe a port. Depending on the response of the target host the attacker will be informed if the scanned ports are closed or filtered. Most firewalls do not log this kind of attacks.

***UDP* scan** uses *UDP* packet that are sent to the target host. A received *ICMP* "port unreachable" informs that the port is closed. *UDP*-based scanning is slow and easily recognized and logged by *IDS*s.

Idle scan [Ant98] where an attacker scans the target harnessing an idle zombie host. This is the ultimate stealth scanning technique since the target host will not receive any packet from the attacker. Instead, the attack is performed on behalf of the attacker by the zombie host.

Decoy scan is a technique of camouflage. Every kind of the aforementioned scan techniques can be used mixing real scan packets with packets with spoofed address in order to confuse *IDS*s.

In addition to this type of port scans, `nmap` also offers a number of advanced features such as remote OS detection via *TCP/IP* fingerprinting, dynamic delay and retransmission calculations, parallel scanning, detection of down hosts via parallel pings, port filtering detection, direct (non-portmapper) *RPC* scanning, fragmentation scanning, and flexible target and port specification.
The aforementioned techniques can be used in different attacks setups. More precisely, from the point of view of number of attackers and targets scans can be divided in:

One-to-one scan where a single source performs a scan to a single target machine.

One-to-many scan where a single source scans more than one target machine. If the attacker performs scanning activity to a single port on multiple hosts it is commonly referred as "portsweep".

Many-to-one scan where more than one source machine scan a single target machine.

Many-to-many scan where a group of source machines (more than one) perform scans on a group of target machines (more than one).

Detection Techniques

When an attacker scans a single host, a host-based *IDS* can easily detect the scan. Moreover, if the scan is performed over a short period of time detection is even easier.
Traditionally, port scan detection is performed using a simple algorithm that checks whether a given *IP* source address touched more than a predetermined number of different ports

on the same host or different *IP* addresses within a given time window. This kind of approaches, as reported by Jung et al. in [JPBB04] have difficulties catching all but high-rate scanners and often suffer from significant levels of false positives.
Jung et al. [JPBB04] presented an algorithm based on sequential hypothesis testing called Threshold Random Walk that can perform a prompt detection of scans even when they occur at a very slow rate. Authors claim that Threshold Random Walk can generally detect a scanner after 4 or 5 connection attempts, with high accuracy.
Problems arise when scans are performed scanning few ports on different machines and possibly from different sources.
In this case, the adversary will divide the target space amongst multiple source *IP* addresses, so that each source scans a portion of the target. The result is that an intrusion detection system might not detect any of the sources. If the sources are detected, the system will not recognize that the various sources are collaborating.
This kind of co-ordinated reconnaissance is primarily of interest to a particular niche of defenders, such as those at the nation-state level.
In order to detect co-ordinated port scans some techniques rely on correlating a huge amount of information (usually a list of single source port scan). For instance, Gates [Gat06] proposes a method to find co-ordinated port scan evidence by employing an algorithm that is inspired by heuristics for the set covering problem.
Sridharan et al. propose [SYB06] a method to detect (single sourced) port scan on backbone links. They stress the fact that there are several important differences between detection port scans in a transit network and in an enterprise-level stub network. One of the most important is of course that link speeds at the core of a transit network are much higher (e.g., 10 Gbps) than at the edge, where it connects to a stub network.
Therefore, the link throughput and the memory needed to store the partial state information makes centralized solutions unfeasible. Moreover, on a backbone link the space of source and target address is virtually infinite, and thus it is impossible to partition the traffic on a per-connection basis, which excludes all the parallelized solutions seen so far.
In conclusion, present solutions to enhance *IDS* performance cannot deal with all the kinds of attacks. Co-ordinated port scans and all the attacks whose signatures spread all the event space cannot be parallelized by data parallelization, which is the only way to relief the sensor from livelock, and continue performing stateful intrusion. In the following chapters, we will a more general method for data parallelization in *IDS*s that can help solve these problems.

Chapter 4

Model Description

4.1 Model

Following the considerations drawn in Chapter 3, we briefly specify our intrusion detection model, which will be then used as the basis to describe different aspects of the intrusion detection process.
In general, as seen in Chapter 2, a misuse detection system scans a stream of events and detects attacks using attack signatures, which are models of the manifestation of an attack. The signatures, or rules, are matched against the stream of events by a rule matching system.
In this case, we define an *event* e as a set of attribute-value pairs (a_i, v_i).
The input to the rule matching system consists of a *stream (ordered set) of events*, which we represent as $\mathcal{E}$.
In the case of a network-based *IDS*, an event is an abstraction of what happens of relevant on the network with respect to our detection purpose. From this perspective, every packet coming from the tap is to be considered an event.
In general, every stream of information can be considered as a stream of events. Even though in this work we focus on packet analysis, most of the concepts discussed here can be reused in different contexts, such as system call analysis [MVKV06]. Even the alerts generated by the intrusion detection system itself can be seen as an event stream that can be re-processed in a hierarchical fashion [SHM02].
Event streams are checked against signatures, which represents a subsequence of the stream of particular interests. In order to detect an attack signature, rules are applied to the stream of events.
As seen in Section 2.3.1, a rule r is described as composed of a predicate P, a state S, and an action A [EVK00, sno04]. The predicate P describes the constraints on the values of the event and state attributes. If the predicate evaluates to true for an event then the action is triggered. The action A consists in modifying the rule state (in this case the rule is deemed "stateful") or raising an alert (that is, generating a new event for further processing). The rule maintains state S in order to keep track of the relevant part of the state of the system.

For example, a counter may be used to detect a port scan.
A number of architectures that implement a rule matching system as described above can be found in both the literature and the real world [Roe99, Pax98, EVK00]. These architectures are centralized, which means that a single processing node deals with all the events. The efficient design of a parallel system that implements the rule matching process is challenging.
Without loss of generality, herein, after we focus on network traffic analysis. The first thing to focus on is to understand what we want to gain from a parallel architecture that analyzes network traffic versus its non-parallel version.
The *NIDS* pattern matching activities carried on the packets require both memory (space) and processing power. Nevertheless, the first bottleneck encountered in present implementations is of course the processing power, since, as we have seen in Chapter 3, both network drivers and kernels easily fall under the weight of a fast paced packet stream. The memory usage should not be neglected, thought. In some cases, attacks could span a long period of time and therefore need a great amount of memory to store fine-grained information to perform intrusion detection. [1]
A parallel architecture should thus help lower the processing power demand. The only way to achieve parallelization in pattern matching activities on packets is to distribute the network traffic among different agents. In order to do that, we refine the standard rule/events model proposed above with some assumptions:

1. Each sensor tries to match one or more rules against the traffic it is fed with. Rule scanning is performed such that multiple matches are all considered. This means that if a packet triggers a rule, *and even if a terminal one, resulting in an alert,* the sensor is not dispensed from matching all the remaining rules.

2. The system does not rely on any predefined mapping between packets and sensors. There is not any locality concept in signatures, every packet could be sent to any of the sensors. This is a more general approach than the one described in [KVVK02] where the traffic slicer implemented and enforced a mapping between packet features and sensors based on the analysis of signatures' event spaces. Here, we take into consideration the most general case, that is, for any pair of packets, we do not have any *a priori* knowledge that they both belong to the same event space. Stricter assumptions in order to optimize the proposed model will be analyzed later.

From the assumptions above immediately follows that *every packet coming from the net has to be scanned against all the rules at least once.* This goal can be accomplished in different ways:

[1] As seen in Chapter 3, for instance in order to detect distributed (co-ordinated) port scans some technique [Gat06, SHM02] rely on correlating a huge amount of information (usually a list of single source port scan). If the traffic to be checked against a port scan was a backbone link, where the space of sensors and scanned address is virtually infinite, the memory needed to store the partial state information would probably overwhelm every other resource in a real world sensor implementation.

1. The sensors have different state (different rules or different state in the same rule), and every packet must meet with each of them; or

2. The sensors have the same state (i.e., they evolve in a synchronized fashion and all the sensor state is replicated) and a packet could hit just one, or

3. A mix of the above (sensors have overlapping states and packets have to hit some of them in order to cover all the rules states)

These solutions cover the spectrum of all the possible ones, ranging from the heavy duplication in the sensors state (memory) and no duplication in traffic (packets) of solution 1 to the heavy duplication in traffic and no duplication in memory of solution 2. Since we are seeking to reduce computation time in scanning packets because this is recognized to be a real bottleneck for a single machine, we want to move towards the solution that guarantees the minimum amount of traffic to be scanned, that is, solution 2.
In this setup the rule state is maintained in the *same way* on all the sensors, which means that:

Fact 1 *Each packet encounters the same working state no matter which sensor is going to take charge of it*

In particular, this means that there need to be some kind of (hopefully simple) action that **has to be performed in the same manner** on all the packets, that is, the **evaluation of the predicate for each rule**. In other words, this means that t

Fact 2 *he scanning part, up to and included the predicate evaluations mechanism, has to be present on every sensors in the same form.*

This is no a problem itself, since predicates can be preloaded in the sensors and will thus behave separately leading to a total parallel sensor speedup.
Unfortunately, as we have seen in Chapter 3, problems arise when the predicate evaluation is a function of the rule state itself and possibly modifies the state (this happens with rules that match signatures of stateful attacks, that is, attacks caused by an ordered sequence of events), more precisely, a matching at time t_1 determines the behavior of subsequent scanning at time $t > t_1$, more precisely, the predicate of the rule is in the form:

Definition 1

$$S_{R_i,t+1} = F(I_t, S_{R_{j,\forall j \in R},t})$$

where I_t is the input at time t, R_i is a generic rule, $S_{R_i,t}$ is the state of the rule i at time t and R is the general rule set loaded into a sensor (For the sake of simplicity we can simply restrict ourselves to the case in which the state of a rule i can be modified only by the rule itself so Definition 1 above would become $S_{R_i,t+1} = F(I_t, S_{R_i,t})$.
If one sees the rule matching process as a state machine, F is no more than its *transition function.*

The computationally expensive parts of the whole process are hidden in the formalism above and are most notably the evaluation of the I feature, performed in the predicate evaluations, which will eventually trigger a transition and the corresponding state update. Evaluating the predicate is part of the scanning process and it is perfectly parallelizable, since each sensor evaluates the same predicate on different packets and we cannot exempt from doing that for each packet if we do not have any *a priori* knowledge on the mapping between packets and sensors. However, the state update is instead really what we do not want in a parallel system, because it is an operation that needs to be repeated for all the sensors in the same manner since all the packets have to encounter the same state regardless the sensor by which they are being processed.
These considerations are not very encouraging when building a parallel system as described before. Amdahl's law [Amd67] here fits perfectly: the time needed by the state update has to be spent by all the sensors for each packet that modifies the state even though this happens on only one sensor, and, therefore, cannot be parallelized (see Section 4.4). During the state update, all the sensors do the same operation behaving as they where a single machine and without exploiting any parallelization.
Note that the above considerations hold for the scanned packets that match and thus update the rule state. Not all the packets will match (hopefully very few will) so we can still gain a lot by parallelizing the packet scanning process.
We will formalize the above considerations in Section 4.4, but we already understand that the bottleneck of the architecture deemed to address a general parallel stateful signature matching is updating the state of the rule when a match occurs.
Intuitively, we need to force the state update operation to be as fast as possible because it does not scale [2]. In order to do that, we need to rethink what the state of a rule is. Not all the state of a rule is needed for the *predicate evaluation*, and, therefore, we can redefine the concept of rule, trying to separate the state really needed for the scanning part from the one which is needed for other subsequent operations.
More precisely, we split the rule state into two parts: a *scanning state*, or predicate state, and a *working state*. The *working state* is needed to raise the alert but it does not modify the scanning state and thus the matching capability. Only the predicate state needed for the evaluation has to be updated in the aforementioned non-scalable way, while the actual working state can live on a separate centralized machine, that we call "control node" and be updated asynchronously. Later we will focus on the architecture of the system).
More formally, we can functionally decompose a rule as below:

Definition 2

$$\begin{aligned} S_{P_i,t+1} &= F_{SP}(I_t, S_{P_i,t}) \\ O_{P_i,t+1} &= F_{OP}(I_t, S_{P_i,t}) \end{aligned}$$

[2] Later on, we will be show other pitfalls in the scaling analysis.

Definition 3

$$\begin{aligned} S_{W_i,t+1} &= F_{SW}(O_{P_i,t}, S_{W_i,t}) \\ O_{W_i,t+1} &= F_{OW}(S_{W_i,t}) \end{aligned}$$

(S_{P_i}) is the *scanning (or predicate) state* and (S_{W_i}) is the *working state*.
Definition 2 describes the behavior of the generic sensor i in term of a *Mealy Machine* while Definition 3 describes the control action in terms of a *Moore Machine*. The fundamental point is that the control node can process the output from the sensors in a completely decoupled environment from where the sensors live.
The definitions above postulate that there is no feedback from the working state to the sensor state. This means that the two automata are working in "cascade". So far, this is not a limitation since we still haven't defined in *which* way the rule state management has to be split. This is a very important part of the model, and it is up to who writes the rules. Intuitively, we want to split the state management in a way that minimizes the state update process on the sensors and that does not require any feedback from the control node to the sensors.
For instance, a rule that can recognize a port scan attack can be implemented as a *TCP* stream-based predicate on the sensors that take care of filtering out packets that are part of a flow and a detection part on the control node that takes care of holding all the data structures needed for detection such as list of targets and scanners, counters and timers, or a whole graph implicit representation as described in [Gat06].

4.2 Architecture

Following the considerations drawn in Section 4.1, we now try to design an architecture that implements our model.
In what follows, we describe the architecture in the context of network misuse detection, but the architecture is general and it could be easily adapted to other domains. Also, without any loss of generality, we describe the architecture and algorithms assuming that a single signature is being matched at a given node. Multiple signatures matching can be trivially added but it complicates our discussion unnecessarily[3].
The proposed system scales the rule matching throughput of a sensor (i.e., a rule matching machine) by using a parallel architecture as shown in Figure 4.1.
In our setup, packets are obtained from a tap (T) into the high-speed network link(s) that copies the packets and sends them to a packet numbering unit (PNU), which, in turns, tags the packets with a logical timestamp (e.g., generated using a counter starting from zero)[4] and forwards them to the splitter. The splitter, or scatterer, sends the traffic to N

[3] In a real-world setting performances could be enhanced by signatures' similarities as described in [MMK05].

[4] Packet identifiers can either be written into unused or unneeded fields such as the source Ethernet address, or can be prepended/appended to the original packet and encapsulated into a new packet being aware of possible pitfalls originated with packets with size close to the link *MTU*.

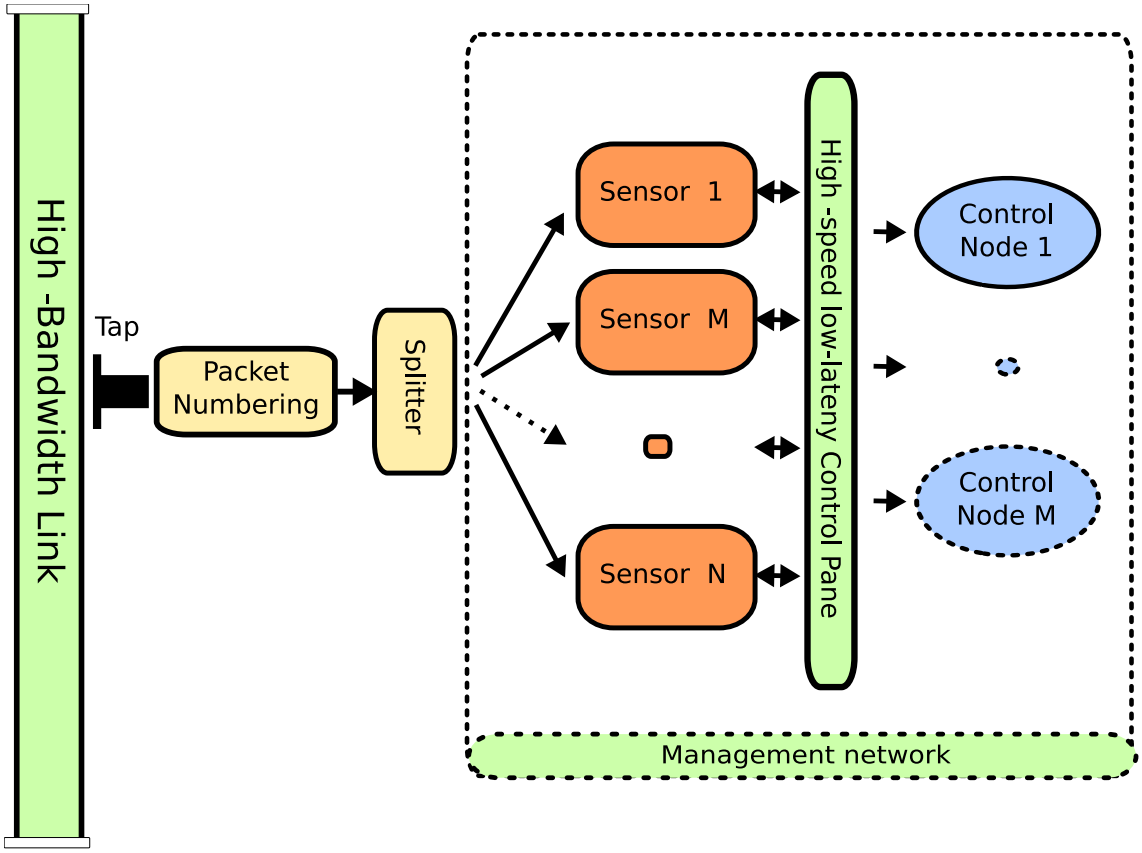


Figure 4.1: The architecture of the parallel rule matcher.

sensor nodes in charge of performing the detection activity in a round-robin fashion.
The sensor nodes are controlled by a control node that maintains the working state associated with the attack instances. The sensor nodes forward any packet [5] that matched a rule to the control node. The control node updates its state (the working state) according to the messages sent by the sensors. In addition, the sensors are able to talk to each other through a broadcast channel so when one of them matches a rule, the others can update their predicate state (once again, the same for all) consistently.
Non-matching packets can be processed in any order by the parallel rule matcher while matching packets should not. If a packet matches a rule, it could update the predicate state of the sensors; therefore, we need some mechanism to ensure that packets encounter the "correct"[6] state at least once. A way to ensure this would be to place an explicit synchronization (barrier) on the sensors in order to make the parallel matcher scan the packets in the right order, one by one. Unfortunately this condition, even though simply

[5] More precisely, only the relevant attributes of the packet such as, for example, (such as, source or destination *IP*) are forwarded to other sensors and control node.

[6] In this context "correct" means "as it would be in a single, centralized, matcher".

to enforce, would transform our parallel system back into a serial one.
The solution to this problem is that each sensor stores packets it has already scanned for rule matches in a buffer called inter-node match buffer (INMB). This buffer is required in case an earlier packet was matched by another node which triggered a sensor state change. The rule with the new state loaded must be matched against packets that came after the packet that triggered the state change (because this would happen in a serial rule matcher). By carefully designing a protocol for removing unneeded packets from the INMBs, we ensure that the set of nodes implement a parallel rule matching machine, i.e., the system functions as if it were a single high-throughput rule matching machine.
By having a single control node, we avoid synchronization issues in the working state, but we do not scale the state of the matching machine. We only scale its ability to filter out the traffic that does not match the signature. For any well-written practical signature, the amount of traffic that matches any rule is a very small fraction of the whole traffic, so even this parallelization should increase the matching throughput by orders of magnitude. Further parallelization may be obtained by having multiple control nodes (say M).
When the traffic is split in round-robin fashion, each node gets $1/Nth$ of the total throughput and of course this can be scaled by increasing the number of nodes N.
Clearly, this architecture for a parallel rule matcher can be enhanced with other general methods of parallelization, such as function parallelization (using separate systems to match different signatures), and component/subcomponent parallelization (for instance reassembling *TCP* streams in a separate process) as described by Wheeler et al. in [WF07] and discussed in Chapter 3.
Note that this model could sound similar to the well-known sensor model paradigm of distributed intrusion detection [SBD+91]. The sensor model is a concept widely used in *distributed* intrusion detection in order to monitor various locations in a network and it aims at improving the accuracy of the system, not its speed. The sensors in a distributed *IDS* work with independent states and do not need to synchronize with each other. These sensors only *preprocess* events that will be later mined by a central component and do not need any mutual feedback. Our model is different since the sensors have to work with a (hopefully little) portion of the state synchronized, and, therefore, they need to communicate with each other *in real-time*.

4.3 Algorithms, Protocols, and Analysis

The architecture in the previous section requires an appropriate algorithm and protocol to ensure correct matching. In this section, we present our proposed algorithm and protocol and analyze its scaling properties.

4.3.1 Algorithm and Protocol

We present an algorithm, called "sloppy rule matching" that works for signatures that give correct results even if packets being fed to a rule matching machine are reordered.

In practice, we expect the sloppy rule matching to work very well also for other rules, in particular we argue that with a careful design of the rules in term of state splitting (see Section 4.1) it is possible to move all the order-invariant parts on the sensors and have the ordered processing on the control node.
Naturally, sloppy rule matching is more efficient and it is easier to analyze its scaling properties under some reasonable assumptions. However, an exact rule matching would be significantly more complex and the scaling properties harder to evaluate.
It has to be noticed that the sloppy rule matching algorithm still needs that all the sensors have the same predicate state. In order to enforce this we will provide a set of communication primitives that the sensors will call upon every match to synchronize with each other. The sloppy rule matching simply eases the analysis of the system speedup but still needs all the components described so far to be fully functional.

4.3.2 Sloppy Rule Matching

In this section, we discuss a parallel algorithm for rule matching that is correct except for the reordering of some packets. More precisely, the parallel rule matching machine is equivalent to a single rule matching machine with a packet reordering operation applied to the input stream. The packet reordering will happen only over a small, localized part of the stream, depending on the difference in the matching throughput between the slowest and the fastest sensors. As such, in practical applications we do not expect this to cause significant problems. This rule matching algorithm correctly matches signatures that are invariant with respect to the order of events, as well as signatures, such as port scan, that are order invariant in practice. In fact, this sort of rule matching protocol is one of the most efficient ways to detect attacks such as port scan.
We now look at the parallel algorithm that implements sloppy rule matching. We assume that all messages between a pair of sensors are transported using a *FIFO* messaging system. Each sensor, say i, maintains the following variables:

- $lastInspected_i$: the packet id (recall that packets are tagged with increasing numbers before getting to sensors) of the last packet inspected.
- $pendingMatches_i$: This is a priority queue arranged in the ascending order of the value of packet id k_i containing the matches that are pending, i.e., not acknowledged by the other sensors yet. The head of the queue contains the packet id of the earliest packet that needs to be retained at sensor i.
- $earliestNeeded_j$: The PacketId of the earliest needed packet at sensor j.
- $ruleSet_i$: The rules loaded at sensor i.
- B_i: the buffer (B_i) in which packets are stored, ordered by packet id. Packets stored in this buffer include those yet to be scanned, as well as those already scanned but not discarded yet.

Note that when we refer to packets as being "earlier" or "later", we use packet id as the ordering parameter; with smaller corresponding to earlier and greater corresponding to later. In addition, when we say "packet x" we mean "packet with packet id x".
Now we are ready to specify the algorithm, and we do so for one sensor i:

New packet available from network: insert it at the end of buffer B_i.

Scan: scan the next packet, say P_k, which is the packet immediately after the last inspected packet ($lastInspected_i$) in B_i. Set $lastInspected_i$ to k. If a match occurred, call BroadcastMatch(), otherwise

BroadcastMatch($triggerMesg$): if the packet matches a rule (say) r_l then broadcast $< match(P_k, r_l, i) >$ to all nodes including self. Add $< k, triggerMesg, timestamp >$ to the $pendingMatches_i$ priority queue. The parameter $triggerMesg$ may be $null$ if no other match triggered this match. P_k is the packet that matched. Of course, in a real-world deployment only the relevant information needed by the other sensors will be broadcast. As a last operation, the matching packet P_k is tagged with the rule id that matched and is stored into the buffer.

ProcessMatch: a rule match $< match(P_k, r_l, j) >$ is received. Update rule set R_i according to the action of the rule. For every new rule added by the action, scan the packets up to and including $lastInspected_i$ in buffer B_i that are after packet k for a match against the newly added rules. For each match that occurs at packet h call BroadcastMatch($< match(P_h, r_m, i) >$).

ProcessMatchAck($< MatchAck(P_k, r_l, j) >$) : Mark in $pendingMatches_i$ that sensor j has acknowledged $< k, triggerMesg =< match(P_h, r_m) >, timestamp >$. If all the nodes have acknowledged this match, remove $< k, triggerMesg =< match(P_h, r_m, j) >, timestamp >$ from the priority queue $pendingMatches_i$ and send a match acknowledgment $< MatchAck(P_h, r_m, i) >$ to node j.

Calculate $earliestNeeded_i$: if $pendingMatches_i$ is empty, then set $earliestNeeded_i = lastInspected_i$, otherwise set it to the packet id at the head of the $pendingMatches$ queue.

Discard packets: let $discardPtr$ be the minimum of $earliestNeeded_j$ over all the nodes j. Discard all packets in B_i that are earlier than $discardPtr$. The sensors communicate to each other $earliestNeeded_i$ using a non reliable version of the bully algorithm [GM82] well-known distributed algorithm for leader election.

Send message to control node: all the packets earlier than $discardPtr$ and tagged with all the $RuleId$ of the rules that have matched and are sent to the control node. The control node has a reassembler that makes sure that the packets are processed in the same order in which they appeared on the high-speed link.

The bully algorithm, utilized to compute the *discardPtr*, simply states that not everyone needs to broadcast its current *earliestNeeded*, but only who thinks to have the lowest one. In the beginning, all the sensors think to have the lowest *earliestNeeded*, therefore everyone broadcast it. After broadcasting, each sensor waits a time T [7] during which it listens for answers by other sensors. Then it broadcast its own $earliestNeeded_i$ only if this is lower than all the ones received, otherwise it stops broadcasting.
During the time interval T, whichever sensor had the lowest *earliestNeeded* would have broadcasted it as soon as received some greater *earliestNeeded* from other sensors. Therefore, all the sensors know what the minimum *earliestNeeded* is.
The sensor that realizes to have won the election continues broadcasting its own *earliestNeeded*, which is going to change during the processing because its *pendingMatches* queue can change. By doing this, it allows other sensors to re-match the contention if their *earliestNeeded* becomes lower.
All the sensors can discard the packets from their INMBs periodically (an appropriate timer can be utilized to trigger discards). More precisely, the sensors can discard packets associated with the last *discardPtr* seen that has been stable for a given suitable time. This deferred deletion ensures that the transitory associated with the contention when the *discardPtr* changes has passed, so that there is not any packet needed earlier than *discardPtr*[8].
The canonical bully algorithm prescribes an acknowledgment for each communication, but since this would complicate unnecessarily the protocol, we omit it and we call this version of the protocol "non reliable bully algorithm" .

Correctness

We show correctness of the sloppy matching algorithm in two steps. First, we show that all packets are inspected by all the rules in the rule set, i.e., no packets are discarded without being scanned. Then, we show that the matching is equivalent to a packet-reordered matching.
To show that no packets are discarded without being scanned we note the key fact that if x matches at sensor n, then $earliestNeeded_n$ will not move beyond x (because the sensor is waiting for acknowledgments) until all other nodes have examined their buffers for matches triggered by this match, and matches triggered by matches triggered by this match, and so on.
Showing that the matching done by this algorithm is equivalent to a packet reordered matching is intuitively clear: a sketch of the proof follows.
If the two following conditions hold:

1. the states of the sensors are synchronized, that is, all the sensors have the same state

[7] If the event density over time in the stream read by a sensor is sufficiently high, then T could be an event count. In this way, the heartbeat of the system becomes the average inter-arrival time between events.

[8] This does not mean that the *discardPtr* is the highest possible, other *legitimate discardPtr*s could have been arrived during the back-off period. This makes the INMB purging operation conservative.

loaded; and

2. there are no pending matches,

then is trivial to see that the first packet will be scanned against all the rules in the same state since all the sensors have the same predicate state loaded.
If the packet does not match, then the conditions above remain unmodified and every new non matching packet will go through all the rules.
When a packet matches according to the sloppy algorithm, a match message is sent to the other sensors in order to make them update their scanning state.
If the matched packet was not the *earliestNeeded* then there will be at least a sensor that will update its state and scan the new rule state through its INMB.
In this case, it can happen then that two different packets y and w later than x match causing a state update that is broadcast again.
Now, any other packet z later than x, y and w and present in the INMB of a sensor i will be scanned against a state PS_i, which is function of the relative ordering in which i will receive match notification about x, y, and w. This relative order is determined by contingent conditions and relative instantaneous speeds of the sensors in scanning packets.
In a more formal way, we define the final state reached by i as the result of the iterative application of the transition function F to a starting state S_0 present on the sensor before the matching of x occurred. The final state S_k can thus be defined as:

$$S_k = F(I_{k-1}, F(I_{k-2}, F(\dots S_0))))). \tag{4.1}$$

For the sake of clarity we write Equation 4.1 as:

$$S_k = F^k(I_{k-1}, I_{k-2} \cdots S_0). \tag{4.2}$$

where $I_{k-1}, I_{k-2} \cdots$ is the *ordered* tuple (sequence) of input encountered by the sensor.
Now, if for each permutation π applied to the input sequence $F^k(I_{k-1}, I_{k-2} \cdots S_0) = F^k(\pi(I_{k-1}, I_{k-2} \cdots), S_0)$, then the transition function is said to be order-invariant[9].
Loosely speaking, order-invariant rules have actions that modify the state in a "non destructive way" (for instance, a counter that is incremented, or an element added to an unordered list). For example, suppose that a match occurs at a packet x. If there are two or more nodes with $lastInspected > x$ then we need the order-invariance property, because the relative speed at which the nodes scan their buffers could cause different state updates.
For instance, consider the sensors in the configuration below, where each sensor is paired with the ordered list of packets it is going to scan:

```
sensor1 4
sensor2 8 7 5 6
sensor3 12 11 10 9
sensor4 15 14 13
```

[9] We can also say that the transition function, and therefore the signature associated with it, is c-order invariant if the relations holds for each c-bounded permutation $|\pi_c(i) - 1| \leq c$.

The rules loaded at time 0 are:

```
rule 1)

i=0
if id==4 then:
  load
     if id == 7 then i++ else if id== 11 then i--

rule 2)

if i==1 and id==15 then do_something()
```

When sensor1 matches the first rule the match is broadcast to sensors 1,2,3,4 and all the sensors load the new rule state and start looking for packet 7 and 11 in their INMB.

1. If sensor2 is faster than sensor3, it will match 7, increment i, and broadcast its match (modifying the value of i in the other sensors too);
2. otherwise, sensor3 will match 11 decrement i and broadcast its match (modifying the value of i in the other sensors too, since i is part of the sensor scanning state and it is synchronized).

Depending on what happens first between 1 and 2, sensor4, which is scanning 15, will execute `do_something()` or not (having seen in consecutive times $i = 0, -1, 0$ or $i = 0, 1, 0$ depending on who won the race). It has to be noticed that with the proposed paradigm, which splits scanning states from working state, the rule before would have been implemented more wisely in 2 parts:

1. a stateless rule on the sensors in the form:

   ```
   if id in [4,7,11,15] then forward, and
   ```

2. a stateful rule on the control node like:

   ```
   case packet_id:
   4: i=0
   7: i++
   11: i--
   15: if i==1: do_something()
   ```

In this way, the execution order is well-defined, does not suffer from race conditions, and is equivalent to the sequential stream.
Note that thanks to the reassembler on the control node, all the packets "tagged to be dangerous" get to the sensor in the right order. Given that, it is really unnecessary to implement complicate rules potentially order-dependent. It is better to keep rule predicates order-invariant and move all the order-sensitive parts in the control node, instead. This is a rule of thumb, not a proof, though.

4.3.3 Exact Rule Matching

We now sketch how the sloppy algorithm should be modified to handle and resolve possible conflicts raised by order-sensitive signatures. The basic idea is that the packet discarding algorithm works as before, but, in this case, when a packet at a sensor matches a rule, the sensor registers and broadcasts a tentative match. Upon registering/receiving a tentative match, all the reversible rule actions are carried out by all the sensors. A tentative match is confirmed and irreversible actions are performed once the matched packet is ready to be discarded according to the packet discard algorithm. If the node receives a message from another node about a conflicting match to an earlier packet, a tentative match is discarded and its actions rolled back.
The concept of a conflict is simple, but its specification is a bit more complicated: a conflict occurs if a rule matched an earlier packet and the corresponding rule action invalidates the tentative match directly (e.g., it removes the rule or modifies its state.). Clearly, only rules that belong to the same signature may conflict. Thus, we only have to check any new tentative match against all same-signature rule matches on later packets, in order to find all conflicts.
A complete specification of the exact rule matching would require the definition of the methods in charge to perform the conflict checking and rollback operations described above. Since such a complicate protocol is not required for most of the real-life signatures, we provide an analysis just for the sloppy algorithm and leave the formalization of the exact rule matcher for future work.

4.4 Analysis

In this section, we analyze the scaling properties of the architecture. In particular, we show that a system based on parallel sensors has a higher rule matching throughput than a single-node systems when the number of sensors in the system is larger than or equal to a threshold N_T. More precisely, we show that the throughput of the system increases with the number of sensors N.
Note that we will only analyze the sloppy matching case. The exact matching behavior depends on the number of conflicts and rollbacks, and hence it is harder to analyze. We will analyze the cost (in terms of time) for matches as well as for non-matches separately, and then we will combine the costs to get an idea of how well the system performs. The

cost $C_n(i)$ of handling non-matching packets for the i-th sensor in a system with N sensors is:

$$C_n(i) = C_S + C_{sU} + C_{rU} \cdot N. \tag{4.3}$$

where C_S is the cost of scanning a packet against the rule set, and C_{sU} is the cost of sending (using network broadcast) periodic updates of *discardPtr* to the other $N-1$ sensors while C_{rU} is the cost for each sensor to receive the update.
We suppose here that the bully algorithm is working in stationary conditions, and that the leader with lowest *earliestNeeded* has been elected and keeps on sending updates to other sensors every T packets. Since each update has to be received by all the N nodes the C_{rU} is multiplied by N.
The aggregate throughput for a system with N sensors therefore becomes:

$$T_n = \frac{1}{\frac{C_S + C_{sU}}{N} + C_{rU}}. \tag{4.4}$$

For moderate N, the cost of receiving *discardPtr* updates can be made small enough, so that we can ignore the second term in the denominator. In this case, considering that the throughput for a single node matcher is $1/C_S$, we see that the throughput is approximately N times that of a single node rule matching machine (with low C_{sU} and C_{rU} compared to N). Also, consider the case of very large N, then the throughput becomes $1/(C_{rU})$.
Therefore, our parallel rule matching machine's throughput saturates at a value which depends on the frequency of updates of *discardPtr*.
It has to be noticed that, in the worst case, the bully algorithm could behave like a dummy all-to-all broadcast algorithm. If the leader (i.e., the sensor with the minimum *earliestNeeded*) changes frequently, every T new contention for the leader election is started. In this case, the cost for sending and receiving *discardPtr* has to be multiplied by N (all the sensors send the update every T events and all the sensors receive the update sent by all the others).
Following the above considerations, the equation 4.4 becomes:

$$T_n = \frac{1}{\frac{C_S}{N} + C_{sU} + C_{rU} \cdot N}. \tag{4.5}$$

That goes to zero for a high number of sensors.
The formulae above hides the fact that the cost associated with *discardPtr* update C_{sU} and C_{rU} can be chosen by the system designer and can be arbitrary low. Loosely speaking, if we had infinite memory we would not worry about discarding unneeded packets and then we will avoid the buffer synchronization mechanism and its pitfalls. Following this consideration, increasing the maximum INMB size of the sensors and, therefore, increasing the period of *earliestNeeded* broadcasts, would shrink the costs C_{sU} and C_{rU} arbitrary. In this case, even if the asymptotic analysis above is still valid, the system will not suffer from the cost associated with the updates for values of N that can be encountered in the real world. Given this consideration, for the analysis that follows we will suppose that the

bully algorithm works in stationary conditions as reported in Equation 4.4.
Now we take into consideration the analysis in the case of matching sensors and we suppose that every packet causes a match. The analysis in the case of matching sensors is more complicated. The worst case cost is one broadcast/acknowledgment per match, and sensors have to scan their entire INMB for each match. Let us consider the analysis separately for the sensors and for the control node.
For the i-th sensor of an N-sensor system we have:

$$C_m(i) = C_S + C_sB + S_B \cdot N + C_{PS} \cdot N + C_{sU} + C_{rU} \cdot N, \tag{4.6}$$

where C_sB is the cost of broadcasting (send) the match message to the other nodes (using network broadcast it is just paid once for each match) and S_B is the size of the INMB. C_{PS} is the cost, for every node, of receiving the broadcast message and updating its predicate state (thus, the factor is multiplied by N)
The product $S_B \cdot N$ deserves further investigation. As stated in Section 4.3.1, when a new matching for the packet x modifies the sensor state, then all the packets later than x that are in the INMB are scanned against the new rule state. The cost of this additional scanning is difficult to analyze, but we attempt to estimate it using the differential state concept.
We define *differential state* the difference between two consecutive states visited by a sensor. The term "difference" is used improperly, since it is not clear what we are differencing and this is not an algebraic or a set-wise difference. Intuitively, this difference denotes the new information added to the state with respect to the predicate to which the state is applied. For instance, if the state of a sensor contains a list of destination *IP* address then the differential state can be a new *IP* added to the list, if the state we are taking into consideration is a counter, then the difference is simply the algebraic difference between the values before and after the update. In the most favorable scenario, each packet in the INMB later than x have to be checked against only the differential state $dS_{x,x+1} = (S_{x+1} - S_x)$, and if the cost of this checking is proportional to the dimension of the differential state, that is $C_{S_{x+1}} = C_{dS_{x,x+1}} + C_{Sx}$, this would be equivalent to what happens in a serial machine where the cost C_{x+1} is spent all in one. Unfortunately, this sort of "additive cost property" with respect to state is not likely to hold. For instance, if a value in a packet, let's say the TTL, has to be checked against a counter, the cost of the comparison would not be additive at all: the cost of comparing $C_{c\geq?k+1} \neq (C_{c\geq?k} + C_{c\geq?k+1})$. However, if the state we are handling is a list, then we could check later packets just against the new values added to the list, and in this case we will have $C_{c\in L|T} \approx (C_{c\in L} + C_{c\in V})$ (the symbol | represent the concatenation operator).
If the cost of scanning later packets against the new loaded state is additive, the cost of scanning the INMB should be neglected from equation (4.6) when considering the performance of the parallel sensor versus the single machine. This is because the cost of scanning the INMB would be the same cost spent on a serial machine that had received the packets in the correct order, and therefore, tested the later packets versus the state modification caused by the earlier ones.

In this case, the formulas for both the serial and the parallel sensors should be corrected to take into account that the scanning cost of matching (and non-matching) packets is not constant, since the state increases after matching. We can replace C_S with its mean value $\hat{C_S}$ averaged on the event stream.
If the cost is not additive, the parallel system incurs in a performance penalty with respect to the single-sensor system. The worst penalty possible happens when the cost for scanning the differential state generated by a match in a later packet is equal to the cost to scan the state itself (see the example of the counter above).
In that case, the *additional* cost with respect to a serial matcher that the parallel sensor has to pay will be a fixed $\hat{C_S}$ multiplied by the *average number of late packets with respect to* x. This number is difficult to estimate, but we can safely say that it is proportional to the number of sensors if the splitter splits the traffic in a round-robin fashion[10]. Therefore, the additional cost would become $S_B \cdot O(N)$
Now we take into consideration the control node. The cost in case of matching is:

$$C_M(C) = C_R + C_{WS}, \tag{4.7}$$

where C_R is the cost of receiving the update message and C_{WS} is the cost spent updating the working state (paid once).
Let us now consider the throughput of the system of N nodes, when every packet matches. The throughput for a single node machine is $\frac{1}{C_S+C_M}$ (where C_M is the actual cost derived by matching, that is, updating the rule state) and with respect to sensors the throughput for a sensor with N nodes is:

$$T_{M_S} = \frac{1}{\frac{C_S+C_sB+C_sU}{N} + S_B + C_{PS} + C_{rU}}. \tag{4.8}$$

For the control node, the throughput is:

$$T_{M_C} = \frac{1}{C_M + C_R} \tag{4.9}$$

therefore, the total throughput becomes:

$$T_M = min(T_{M_S}, \alpha \cdot T_{M_C}), \tag{4.10}$$

where α is a multiplicative constant which denotes how many times the control node is faster than the sensors[11].

[10] Intuitively, suppose we have N sensors numbered from 1 to N and a round-robin splitter which sends packets in increasing order with respect to the sensors. Now, the worst possible case for matching in terms of INMB rescanning happens when the first packet matches. Later packets have already reached other sensors and will be in their INMB. If the first packet matches, then all the other sensors have to scan exactly one packet in their buffer, thus making the whole INMB scans a $O(N)$ operation.

[11] This constant is needed to take into account that sensors and control node can be different kind of machines, possibly with different hardware.

α comes handy if we want to decouple mathematically the two sensors and take into consideration a possible increase in the computational power of the control node. If the control node doubled its processing

For a mixture of traffic with matching probability p the total throughput of the parallel machine is (as a function of the probability p):

$$T(p) = min\left(\frac{1}{(\frac{C_S+C_sB}{N} + S_B + C_{PS}) \cdot p + (\frac{C_S}{N}) \cdot (1-p) + \frac{C_{sU}}{N} + C_{rU}}, \frac{\alpha}{p \cdot (C_M + C_R)}\right) \tag{4.11}$$

These equations confirm the hypothesis done in Section 4.1 about what the bottleneck in such a designed parallel system is. The non-scaling part of the equation, apart from the *earliestNeeded* update, which under some hypothesis can be neglected, is the $S_B + C_{PS}$ part, regarding the sensor state update, which has to be accomplished on every sensors, and for each match.
Those costs are not parallelizable. According to the Amdahl's law, the total achievable speedup with a N-sensors system is:

$$S = \frac{1}{F + \frac{1-F}{N}} \tag{4.12}$$

where F is the fraction of a calculation that is sequential (i.e., cannot benefit from parallelization), and $(1-F)$ is the fraction that can be parallelized. In this case $F = S_B + C_{PS}$, but if we had not split the rule state in S_P and S_W we would have had $F = S_B + C_{PS} + C_M$ (where $C_{PS} + C_M$ represents the cost of updating *the whole rule state*) in our equations that would have further lowered the achievable speedup.

4.5 Optimizations

The general model described before can be optimized in a real-world setup. In particular, the approach of reducing the cost impact of the sensor state update could be achieved relaxing the assumption (2) and exploiting some *a priori* knowledge to send related packet with respect to a signature to the same sensor.
If such a knowledge exists, then the fact (2) doesn't hold anymore and this means that there is no need to have all the sensors to behave in the same manner (i.e., have the same state). In other words, this means that the sensor state can be partitioned into independent substates that can be updated separately and in a decoupled way. This is made possible by the event space partitioning induced by the signature being matched.
If there is no need for inter-sensor state update, equations 4.11 becomes:

$$T(p) = min\left(\frac{N}{C_S}, \frac{\alpha}{p \cdot (C_M + C_R)}\right), \tag{4.13}$$

because the costs for sending and receiving state updates among the sensors disappears and even the cost for buffer synchronization is not needed anymore.

power we can highlight this either by dividing $C_M + C_R$ by two (because the cost *seen* by it would be half) or doubling α. This second formalism is much clearer.

In this way, we can find the models discussed in Chapter 3 as a special case of the model described here. A very common example of this kind of optimization is to split the traffic so that packets belonging to same *TCP* streams are mapped to the same sensors. These sensors use signature of non cross-connection attacks.
If the signature matched on the sensor is *TCP*-connection-friendly, then all the matching process can be performed in parallel on the sensors, in addition the control node can be avoided, therefore equation 4.13 would become:

$$T(p) = \frac{N}{C_S}, \tag{4.14}$$

which denotes clearly the maximum achievable speedup and is consistent with what we have stated in Chapter 3.
It is remarkable to notice that reintroducing the control node in such a design can still add power to the architecture adding statefulness to *TCP*-flow-related signature. For instance, the sensor could run a stateless connection-wise signature that tags as "possibly dangerous" every *HTTP* payload containing an operation regarding cookie manipulation. Then, the control node could correlate the (possible huge) relevant traffic in order to find misuse in cookie manipulations (such as cookie tampering). This two-step hierarchical approach has been successfully employed in a number of real-world architectures [Roe99, Pax98].
It has to be noticed that the above optimizations are possible only discarding the general assumptions on which our model is based in favor of more specialized ones. In the general case, the equation (4.11) still holds and gives us an upper bound of how far an *IDS* can be parallelized for a generic signature.
It remains to be said that in the described architecture are still applicable all the single-sensor optimizations discussed in Chapter 3, namely techniques to optimize the rule matching by reducing the single sensor analysis run time such as [MMK05] and [LZL+06] or to enhance the packet splitter capabilities [XCA+06].

Chapter 5

Implementation

5.1 System Implementation

In this chapter, we will describe how the model proposed in Chapter 4 has been implemented in a demonstrative -yet working- real-world system. We will focus on the implementation of the sloppy rule matcher only.
We start by outlining the domain where the parallel rule matcher is designed to work. We tried to implement the system in the most generic way possible, but, since the implementation employs low level network primitive, we had to specialize it in order to handle a specific link-level protocol, that is Ethernet. Nevertheless, the proposed implementation could be easily adapted to other kinds of network technologies (e.g., Myrinet [Myr]) since the Ethernet-specific code is encapsulated into a general-purpose library.
In the same fashion, in order to work with raw sockets at the link-level, we used the PF_PACKET/SOCK_RAW family socket interface available in GNU/Linux >=2.2. As before, our implementation can be easily ported to other *OS*s that have features similar to the ones provided by the packet sockets since the *OS*-dependent code is well encapsulated in a well defined library.
The implementation has been supported by a network emulation environment that helped ease the development and testing of the system. Having the possibility to seamlessly test the whole system on a single laptop, with all the logging information from different nodes merged into a single buffer was instrumental in understanding the subtle issues related to concurrency that are very important when designing a parallel system.
Some effort has been spent to have the emulated network testbed exposing the same interface of a real network testbed in order to be able to deploy the same parallel system in both the emulated and the real-world environment without any changes in configuration or building.
An overview of the available implementation choices follows. In this chapter, the emphasis will be put on the software part of the implementation, that is, the implementation of the parallel protocol described in Chapter 4. In the next chapter, we will provide a more precise description of the issues related to the deployment of the system in both the emulated and

the real-world network testbed.

5.2 Possible Implementation Choices

The choice of how to implement the parallel sensors and the control node has been a crucial one. Loosely speaking, two possibilities were available:

1. writing the parallel intrusion detection system from scratch, or
2. enhancing a current intrusion detection system with the parallel protocol and algorithms described in the previous chapter.

Both solutions have pros and cons. Solution 1 is simpler to implement since it would not have required any knowledge of code written by others. In addition, since designed from scratch as a parallel engine, *ad hoc* optimizations and features could have been implemented seamlessly without having to struggle with the lack of flexibility of an already consolidated centralized architecture. As a downside, a from-scratch solution would not have gained from the collateral features that usually come with the commercial *IDS*, that is, a wide range of plugin and signature descriptions that the developer community or the vendors provide to the end user in order to transform a simple detector engine into a real-world, effective intrusion detection system. On the other hand, Solution 2 allows us to reuse some common functionalities of intrusion detection system such as the packet capture framework, the logging architecture, and the protocol decoding. For these reasons, in addition to the possibility to be able to exploit a great deal of plugins and signature descriptions, we have chosen the Solution 2. Even thought this choice has implied some additional efforts in order to understand the previous work and to adapt it to a parallel environment, it has also allowed us to save lines of code exploiting the *IDS* common functionalities. In addition, as we will see in Section 6.3, choosing an existing system has allowed us to exploit and enhance a real world signature such as a port scan.
As reported in Chapter 2, there are many *IDS*s that we could have leveraged for our parallel engine. We have chosen to implement the parallel sensor through a custom version of the `Snort` *IDS* [Roe99]
The choice of `Snort` has been lead by the importance that this engine has gained in recent years in the open source intrusion detection community.
`Snort` is actively developed and has a wide community that provides collateral support, such as signatures descriptions (rules) [JA], plugins, and documentation.
To sum it up, we have chosen `Snort` because, using the words that appear on the `Snort` homepage: "with millions of downloads to date, Snort is the most widely deployed intrusion detection and prevention technology worldwide and has become the de facto standard for the industry"
In addition to the popularity that `Snort` has gained in the open source community, `Snort` is the system that the scientific community employs as a benchmark for detection accuracy measures (see for instance: [SYB06]) or as the basis to implement new features (see for instance: [CM06]).

5.3 Snort

`Snort` is an open source intrusion detection and prevention system capable of performing traffic analysis on *IP* networks. As reported by the `Snort` web-site "it can perform protocol analysis, content searching/matching and can be used to detect a variety of attacks and probes, such as buffer overflows, stealth port scans, *CGI* attacks, *SMB* probes, *OS* fingerprinting attempts, and much more."
`Snort` is free software, available with full source code under the terms of the GNU GPL and it can be downloaded from http://www.snort.org/
`Snort` has three primary functional uses. It can be used as a normal packet sniffer like `tcpdump`, as a packet logger (useful for network traffic debugging, etc.), or as a full blown network intrusion detection and prevention [1] system.
The major strong points of `Snort` are its flexible rule language utilized to describe signatures and the modular plugin architecture that allows detections and reporting subsystem to be extended.

Snort Subsystems

As reported in [Roe99] `Snort` is made up of three primary subsystems.

The packet decoder subsystem is based on the `libpcap` sniffing library and is organized around the layers of the protocol stack present in the supported data-link and *TCP/IP* protocol definitions. Each subroutine in the decoder in charge of decoding a specific protocol is called in the order determined by the current protocol stack, from the data link layer, through the transport layer, up to the application layer. The majority of the work of the decoder routines is to set pointers into the data structure that represents the decoded packet for later analysis by the detection engine.

The detection engine subsystem is the `Snort` core. It performs rule matching against the decoded packets.

`Snort` maintains its detection rules in a two dimensional linked list of what are termed Chain Headers and Chain Options. These are lists of rules that have been condensed down to a list of common attributes in the Chain Headers, with the detection modifier options contained in the Chain Options. These rule chains are searched recursively for each packet in both directions. The detection engine checks only those chain options that have been set by the rules parser at run-time. The first rule that matches a decoded packet in the detection engine triggers the action specified in the rule definition and returns.

The logging and alerting subsystem provides real-time alerting capability by, incorporating alerting mechanisms for syslog, a user specified file, a UNIX socket, or WinPopup messages to Windows clients using Samba's `smbclient`.

[1] For an overview of the intrusion prevention capabilities, see the `Snort` inline mode [Roe]

Program configuration, rules parsing, and data structure generation takes place before the sniffer section is initialized, keeping the amount of per packet processing to the minimum required to achieve the base program functionality.

Plugins

`Snort` architecture is based on plugins. Plugins allow the detection and reporting subsystems to be extended. There are three types of plugin currently available in `Snort`, the following list sketches them briefly.

Detection plugins check a single aspect of a packet for a value defined within a rule and determine if the packet data meets their acceptance criteria. For example, the *TCP* flags detection plugin checks the flags section of *TCP* packets for matches with flag combinations defined in a particular rule. Detection plugins may be called multiple times per packet with different arguments.

Preprocessors are only called a single time per packet and may perform highly complex functions like *TCP* stream reassembly, *IP* de-fragmentation, or *HTTP* request normalization. They can directly manipulate packet data and even call the detection engine directly with their modified data. They can perform less complex tasks like statistics gathering or threshold monitoring as well.

Output plugins are called when an event or an alert are generated. For instance, an output plugin could save alerts and logs into a database instead of writing them into a text file.

Since version 2.6, `Snort` was enhanced with dynamically loadable modules. They can be loaded via directives in `snort.conf` or via command-line options and can perform the same types of operation of their static counterpart but, in addition, can be loaded at runtime without rebuilding the engine.

5.4 Patching Snort

`Snort` has been taken as a basis for the development of the distributed rule matcher described in Chapter 4 both for the sensors and the control node.
Even though sensors are different from the control node from the implementation point of view[2], we decided to base both components on `Snort`. This choice has been taken in order to maximize the code reuse in the implementation. As a matter of fact, both the sensors and the control node use some common functionalities provided by `Snort`, such as the log system, and both use preprocessors, even though different ones [3]

[2]For instance, while sensors have to process traffic and communicate with each other, control nodes need just to passively receive and process the signals sent by the sensors.

[3]We will see in the next chapter that for the proof-of-concept incarnation of parallel port scan detection the sensors will be equipped with the `flow` preprocessor while the control node will run the `sfportscan` preprocessor.

Here follows a general description of the methodology used to transform `Snort` into the parallel rule matcher. A detailed description of the features implemented for the specific tasks of sensors and control node will be provided in the following sections.
As a first step, we downloaded and installed the sources of a vanilla `Snort` package, version 2.6.0.2, with all the preprocessors, and put the sources under version control (using the `subversion` system).
In order to get acquainted with the `Snort` codebase, we found it very useful to run `doxygen` on `Snort` sources. `doxygen` [vH] is a well-known documentation generator for C++, C, Java, Objective-C, Python, IDL and is usually used to generate online documentation from a set of documented source files. However, `doxygen` can be extremely helpful, as in this case, to extract the code structure of an application from undocumented source files. This is useful to quickly find the way in large source distributions.
The relevant part of a sample `doxygen` configuration useful to get acquainted with a large codebase is reported below.

```
OPTIMIZE_OUTPUT_FOR_C  = YES
EXTRACT_ALL            = YES
EXTRACT_STATIC         = YES
SHOW_DIRECTORIES       = YES
RECURSIVE              = YES
SOURCE_BROWSER         = YES
INLINE_SOURCES         = YES
ALPHABETICAL_INDEX     = YES
GENERATE_TREEVIEW      = YES
HAVE_DOT               = YES
CALL_GRAPH             = YES
INCLUDE_GRAPH          = YES
INCLUDED_BY_GRAPH      = YES
DIRECTORY_GRAPH        = YES
```

Some of the more helpful meta-data have been the call, include, and included_by graphs that `doxygen` can create if used in conjunction with the `dot` package to draw graphs. By using those graphs, it has been rather easy to follow the journey of a packet from its capture through the decoding steps and to the detection part, and, therefore, this has eased the task to transform `Snort` into a parallel rule matcher.
Before branching the `Snort` codebase in two separate repositories and develop the sensors and the control node functionality independently, two features were added to the main codebase:

1. a common debug functionality, and

2. a common way for the sensors and the control node to recognize each other.

The feature 1 enhances `Snort` debug system with a new functionality. `Snort` debug verbosity is driven by the environment variable SNORT_DEBUG. Every time in the `Snort` code appears a call to the debug macro "DEBUG_WRAP" as the following:

```
DEBUG_WRAP(DebugMessage(DEBUG_DIST, "Setup pcap_loop\n" ));
```

`Snort` performs the bitwise AND between the first parameter of "DebugMessage" and the value of the environment variable SNORT_DEBUG and logs the correspondent output to the console if the result of the operation is not null.
We have added a new debug constant:

```
#define DEBUG_DIST 0x08000000 /* 134217728 */
```

that is the lowest power of two not used as other debug constant. We pass it as first parameter of the "DebugMessage" function every time we to log something related to the parallel matcher add-ons. In order to visualize the debug information during runtime, `Snort` has to be started after having setting the environment variable SNORT_DEBUG to a relevant value (in our case, we wanted to have printed only debug information regarding to the parallel rule matcher so we set SNORT_DEBUG=134217728 (the decimal value of DEBUG_DIST)
Feature 2 was required because both the sensors and the control node need to know each other, which means that each of them needs to know:

1. how many sensors are collaborating;
2. *who* the collaborating sensors are.

The reason why the sensors cannot be "cooperation oblivious", that is they cannot ignore what the players of the parallel detection system are, is subtle. As explained in Section 4.3.1, each sensors can remove a match from the *pendingMatches* priority queue only when *all* the sensors have acknowledged it. In order to keep track of *which* sensors have acknowledge the matching, each sensor needs to know which sensors are going to acknowledge messages, or, said in other words, which sensors are running.
Note that, in an unacknowledged communication system, sensors would not need to know who the other sensors are, since a broadcast send primitive allows a single sender to reach all the receivers on a network, regardless of their number and identity.
Since sensors communicate to each other by means of Ethernet packets (we will describe this more precisely in the following chapters) a simple way to identify sensory is using their *MAC* address of the interface connected on the control network.
For this reason `Snort` has been enhanced with a new command option:

```
{"mac-listfile", LONGOPT_ARG_REQUIRED, NULL, MAC_LIST_FILE},
```

by means of which it is possible to specify the name of a file containing a list of Ethernet *MAC* addresses: the addresses of the sensors that are part of the rule matcher. By convention, the first *MAC* appearing in the file is the control node *MAC* address (needed by

sensors to send information to the control node) followed by a list of the *MAC* address of the other sensors. All the sensors are fed with the same file, in order to have all the same picture of the players in the communication.
The relevant logic to parse the file and load the information is trivial and has been implemented but it is not reported here.
After these two modifications the repository has been split in two branches: the snort-sensor and the snort-control branches.

5.4.1 Logical Network Architecture

In order to go on with the description of the implementation details for sensor and control nodes we first need to understand how the sensors are connected to each other.
From the networking point of view, each sensor is provided with two network interfaces: the usual interface used by the *IDS* to perform packet capture and another interface used to communicate with other sensors and the control node.
The capture interface is connected to the distribution network (only Ethernet level, without *IP* support) of the splitter. The control interfaces are connected to another Ethernet network, which also contains the control node.
In the next chapter, a more detailed and hardware-oriented description of how the network has been deployed will be given. For now, it is sufficient to know the aforementioned details in order to understand the following considerations.

5.4.2 Sensors

The sensor has been the component that required the most development time because it contains most of the logic of the parallel rule matcher.
The main modification to the `Snort` packet flow, which is the main entry point for the parallel rule matcher logic, was represented by the implementation of the inter-node communication system.
This has first required `Snort` to be enhanced with a new command option that specifies which interface should be used to communicate with other sensors, named "control-interface" (the code snippet follows).

```
{"control-interface", LONGOPT_ARG_REQUIRED, NULL, CONTROL_INTERFACE},
```

This option has to be passed to `Snort` via the command line in order to specify the interface from which it should expect control packets (e.g., `-control-interface=eth1`).
From now on, we will refer to the packets exchanged between the sensors in order to implement the sloppy rule matching algorithm as "control packets" and to the network device used by the sensor to send/receive control packets as "control interface".
Sensors must handle the packet capture and the inter-sensor communications at the same time. This implies that the sensors must process packets coming from the two interfaces concurrently, which is a major difference from the standard way of operating of `Snort`.

There are many solutions to implement this concurrent behavior. A list of the most often used solutions follows.

1. Instruct the `libpcap` library, upon which `Snort` is based, to capture packets from both the interfaces. Some userland demultiplexer then performs control or detection operations according to the type of the decoded packets.

2. Use two separate threads to read from the two interfaces and then employ condition variables to synchronize between the two.

3. Do not use any concurrency. The two streams of packets are time-multiplexed by simply polling the control socket (set in non-blocking mode) every time the `pcap` callback function is called to deliver a new captured packet.

Solution 1 is be the simplest to implement, since the management of the concurrency remains concealed in the `libpcap` library, which would be in charge of merging control packets and traffic and make them appear to the `Snort` process as a single stream of packets. In that way it is easy to implement the logic to handle control packets diverting them from the main packet stream without synchronization problems because the kernel socket buffers ensure that there are no lost packets while the sensor is taking care of the control operations.
Unfortunately, in order to make the `libpcap` library capture from two or more interfaces it is necessary to configure it to listen on the Linux “any” interface. The Linux "any" cooked-mode interface gets all the packets received (or transmitted) on any interface (including the localhost loopback interface) in both directions (incoming and outgoing packets). Since the different interfaces do not have the same link-layer header (e.g., for localhost loopback there is no Ethernet header) the “any” interface uses a special pseudo-link-layer header, described in `libpcap/sll.h` and a different data link type. The problem with the Linux cooked mode interface is that it is extremely inefficient because all the packets, both in incoming and outgoing directions, are cooked and copied to the userland. The unnecessary copies can be avoided by setting up an appropriate `pcap` filter, however the time spent for cooking packets is not negligible.
The Solution 2 is effective and does not suffer from the drawbacks of the previous one. Nevertheless, it implies a heavy refactoring of the `Snort` main loop in order to accommodate the two reader threads, and the use of thread condition variables to prevent races between traffic and control packets that could be originated by a match.
Solution 3 is good under different points of view. First of all, it does not require any kind of concurrent management of the two interfaces. In addition it is very little invasive, since it requires just to add a single entry point in the main `Snort` loop inside the `pcap` callback function, *before* the captured packets are processed.
In details, the code snippet below shows how the hooks for the processing of the control packets are plugged into the callback function of the main `pcap` capture loop.

```
1 void PcapProcessPacket(char *user,
```

```
2       struct pcap_pkthdr * pkthdr, u_char * pkt)
3 {
4     PROFILE_VARS;
5     PREPROC_PROFILE_START(totalPerfStats);
6
7     /* First thing we do is process a Usr signal that we caught */
8     if( sig_check() )
9     {
10        PREPROC_PROFILE_END(totalPerfStats);
11        return;
12    }
13
14    /* Process all the control packets first */
15
16    pc.total++;
17    if (!skip_forward)
18      {
19        receive_control_packet(pv.control_socket);
20        DEBUG_WRAP(DebugMessage(DEBUG_DIST,
21                   "After receiving control packet\n" ););
22      }
23   ...
```

The `receive_control_packet` function, at line 19, is the main entry point (thus not the only one) of the parallel rule matcher. We will discuss later why it is executed conditionally under the `!skip_forward` guard.
Inside `receive_control_packet` a poll is performed on a packet socket opened in non-blocking mode and listening on the control interface, so that *all* the control packets that are waiting in the control socket buffer queue are read and processed.
Note that this protocol is not starvation-free since if the control node gets congested only control packets will be processed and the traffic will probably be discarded.
It has to be noticed that this condition should never occur since the sloppy rule matching protocol has been designed in order to minimize the communications among sensors, so we can say that we can safely take this risk.
This solution also helps eliminate races among traffic and control packets. Control packets have always the priority upon traffic packets and the actions that they trigger, if any, are processed thoroughly before any other traffic packet is being analyzed.
In addition to the one just described, there are other two entry points in the `Snort` packet capture control flow where the control is passed to the parallel rule matcher logic, which are:

- the asynchronous access to parallel primitives called from inside the matching rules in order to, among other things, notify the matching to other nodes, and
- the primitives called by `Snort` to hand out a (hopefully shallow) copy of the received packets to the parallel INMB.

We first take into consideration the latter since is the most easily explainable.
A slight modification has to be done to the `Snort` core engine in order to have each sensor saving into its INMB the traffic packets captured by `Snort`. Recall that the parallel matcher needs a buffer to store packets already analyzed in order to perform detection operations on them with respect to new rules added remotely by other sensor. `Snort` *per se* discards each packet right after processing it, so we need to implement the logic to store the packets into the INMB instead of disposing them after processing.
We have implemented the INMB as a circular buffer, ordered by packet id (the order is enforced at insertion time). Each packet is saved into this buffer after being decoded and before being passed to the preprocessor and detection chain.

```
1  void ProcessPacket(char *user,
2       struct pcap_pkthdr * pkthdr, u_char * pkt, void *ft)
3  {
4      Packet p;
5
6  ...
7      (*grinder) (&p, pkthdr, pkt);
8
9  ...
10     switch(runMode)
11      {
12
13 ...
14          case MODE_IDS:
15
16 ...
17              if (!skip_forward) put_buffer(&p);
18
19              /* start calling the detection processes */
20              Preprocess(&p);
21
22 ...
```

The code snippet above shows that the `put_buffer` routine is called by the function `ProcessPacket`, which is in turn called by `PcapProcessPacket`, which we have seen in the previous snippet. `put_buffer` is called after the packet has been decoded (the function pointer dereferencing on line 7) and only if `Snort` is running in *IDS* mode.
The logic inside `put_buffer` is fairly simple: a copy of the data structure `Packet` used by `Snort` to store information about a decoded packet is copied into the buffer performing boundary checking.
The third and last hook into the `Snort` core is the most blurry and the least elegant one. In the current setup, since we still do not have signatures written in order to support the parallel rule matcher we had to rely on `Snort` signatures, and, therefore, we adapted them to work in the parallel rule matcher environment.

In section 6.3 we will explain more in details what we sketch here.
In our experiments we have adapted the detection plugins and made them parallel-aware. In particular, we have enhanced a `Snort` preprocessor in order to process traffic and keep state and, in addition, to communicate its state changes to other sensors with the possibility, at the same time, to allow its state to be modified by communication coming from other sensors. This means that the preprocessor cannot operate transparently, or at least not thoroughly, but has to notify its state changes and being able to receive and perform changes communicated by other. We tried to implement this logic in the least invasive way possible.
Each time the preprocessor changes its state, it calls a notification function provided by the parallel *API* that we developed. This functions performs the operations described by the sloppy rule matching algorithms, i.e., crafts a packet with the relevant information and broadcasts it to other sensors (plus ancillary operations such as adding the match to the $pendingMatches$ queue).
The modification of the plugin state as a result of the requests coming from other sensors, is more difficult to implement. As prescribed by the sloppy rule matching algorithm, a request for state change is broadcasted from a sensor on the control network and every sensor needs to perform the actions specified in order to change its state according to the information received. The best way to implement this behavior would be to add a hook inside the plugin to perform the state change, but this would have implied a lot of modifications inside the plugin itself. For this reasons, we have implemented the functionality by "luring" the plugin and passing a fake packet crafted in order to trigger the requested state change as it was read from the capture interface.
This loosely speaking means that upon a match message is received by the control interface, the system creates a packet filled with the relevant information and hands it out to the relevant preprocessor, which is unaware from which interface the packet is coming.
The fake packet will then cause the relevant state changes to happen, but this would imply that a new matching notification is triggered by the plugin. In order to avoid this situation, which would cause the sensors to continue mutually forwarding the same information in a sort of "resonance", when a sensor hands out a packet created from a message received from the control interface it sets the global variable (`skip_forward`), which prevents the packet from being communicated (forwarded) to other sensors.
We know that this implementation, which prescribes to craft a fake packet in order to enforce the state changes in the plugin, is not optimized. We will draw a better way to handle this situation in the future work.
The interconnection between `Snort` and the additional logic needed by the parallel rule matching described above are summarized in Figure 5.1.

Features Implemented

After the general sketch of the architecture and an explanation of how the logic for the parallel rule matcher is integrated with `Snort`, we now give a more precise description of the most important implementation details.

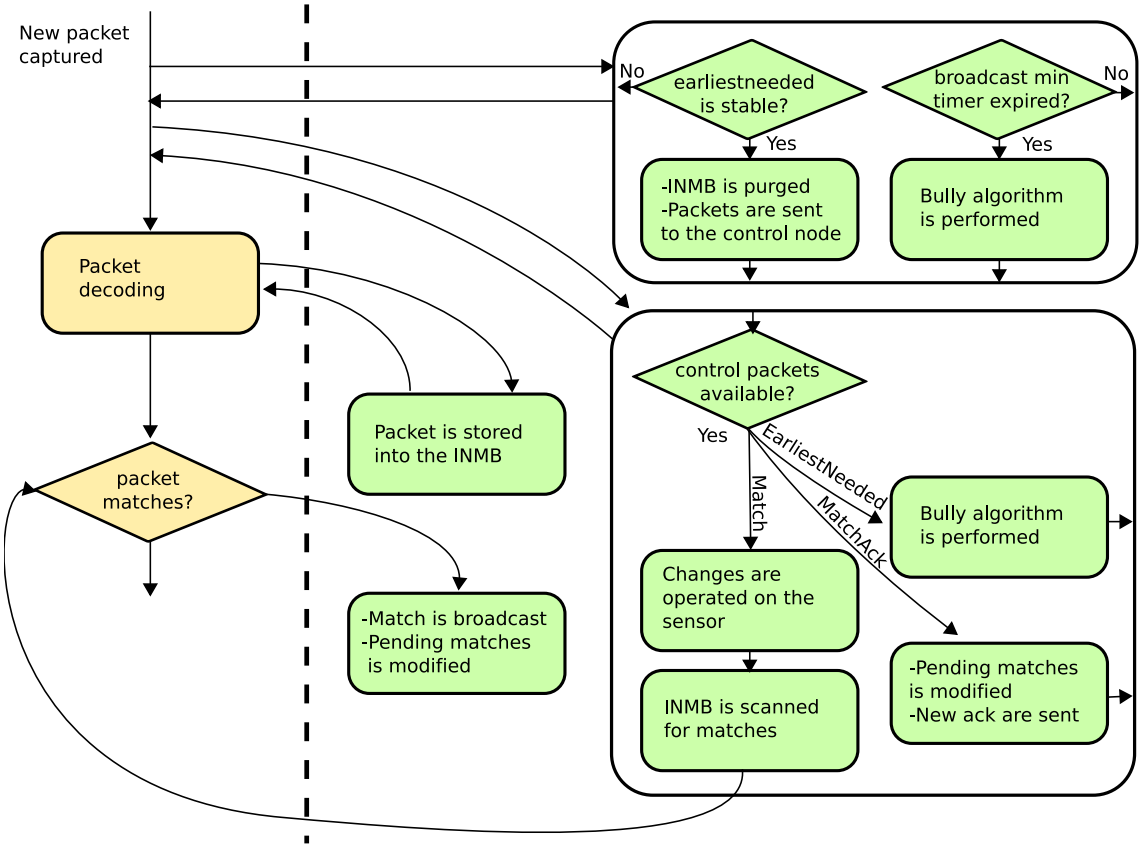


Figure 5.1: Logic interconnection between `Snort` and the parallel rule matching functionalities. On the left is represented the standard `Snort` control flow, while on the right is a representation of the additional logic required by the parallel rule matcher to work. The control is passed from `Snort` to the additional logic in four points (eight arrows cross the dashed line, in both directions).

Let us start from the changes in the data structures used by `Snort`. We enhanced the structure `Packet`, used by `Snort` to represent a decoded packet by adding some additional fields.

```
1 typedef struct _RuleMatchInfo
2 {
3   int rule_id; // rule identifier
4   uint32_t ack_host_bitset;
5   /* bitset of host that acknowledged the match: 0 means 'akc'd' or
6   'no need to ack' thus the semantic of use for this field is :
7
8   -the i-th bit correspondent to the i-th host is set when a rule is
9    matched to indicate that an acnowlegdement for host i is needed
```

```
10
11    -when and host acknowledge the rule/packet pair the correspondent
12      bit is unset
13
14    -if all the bit are unset the packet can be removed from the packet
15      queue.
16    */
17
18      // ruleid of the match that triggered the current one;
19      int trigger_rule_id;
20
21      // packetid of the match that triggered the current one;
22      unsigned int trigger_packet_id;
23
24      // senderid of the match that triggered the current one;
25      unsigned int trigger_sender_id;
26  } RuleMatchInfo;
27
28  typedef struct _Packet
29  {
30
31  ...
32      RuleMatchInfo * match_list;
33      unsigned long int packet_id;
34      unsigned int n_matches;
35      unsigned int n_nacked_matches; /* number of matches that need to be
36                                        acknowledged */
37
38  ...
39  } Packet;
```

Every packet is enhanced with an array of structures `RuleMatchInfo` one for each rule that was matched on that particular packet. The `RuleMatchInfo` structure maintains information about a match, namely:

rule_id: the identification of the rule that triggered the match

ack_host_bitset: a bitset that represent the sensors that acknowledged the match. Not all the rules need explicit acknowledgements.

trigger_rule_id: the rule identifier of the match that triggered the current one (if any).

trigger_packet_id: the packet identifier of the match that triggered the current one (if any).

trigger_sender_id: the sensor identifier of the match that triggered the current one (if any).

The array of `RuleMatchInfo` is dynamically allocated the first time that a packet matches. This prevents wasting memory in a preallocated array that would not be used in the majority of the cases (it is expected that the majority of the packets will not match any rule).
Other auxiliary fields added to the `Packet` structure are:

packet_id: the packet id as assigned by the splitter.

n_matches: the number of rules that matched this packet.

n_nacked_matches: the number of rules that have not been acknowledged for this particular packet.

Rules are numbered with increasing id and each rule is associated with some information, that is:

```
1  typedef void (*transform_list_t)(const Packet * captured_packet,
2   unsigned char**  crafted_packet, int * p_len,
3   const uint16_t rule_code);
4
5  typedef void (*update_state_list_t)(const unsigned char * control_packet,
6   const int p_len, const int rule_id, const uint32_t,
7   const packet_id, const int sender_id );
8
9  ...
10 transform_list_t sensor_transform_list[MAX_TRANSFORM_LIST_LEN] = {...};
11 transform_list_t control_node_transform_list[MAX_TRANSFORM_LIST_LEN] = {...};
12
13 update_state_list_t update_state_list[MAX_UPDATE_LIST_LEN] = {...};
14
15 uint16_t control_node_rule_codes[MAX_TRANSFORM_LIST_LEN] = {...};
16 uint16_t sensor_rule_codes[MAX_TRANSFORM_LIST_LEN] = {...};
17
18 unsigned int need_match[MAX_TRANSFORM_LIST_LEN] = {...};
```

More precisely, each rule is described by three functions:

control_out: a function that transforms the matching packet into the information to be broadcast to other sensors. This function is of type `transform_list_t`. The packet returned by this function is ready to be sent once padded with the relevant Ethernet source (the sensor's MAC address) and destination (the control node *MAC* address) addresses. The array `control_node_transform_list` (line 11) is a list of function pointers of the type specified above, one for each rule.

control_in: a function that reads a control packet that contains the matching information for a certain rule and applies the transformation that the rule prescribes to the sensor's state. This could be considered in some sense as the "inverse" of

the `countrol_out` since the function `countrol_out` creates a control packet from a change of state while this function operates a change of state in response to a control packet.

This function is of type `update_state_list_t`(line 5), that is, a function that takes as input an `unsigned char *` (the control packet) and some information about the rule (`sender_id`: the id of the sensor that broadcasted the rule, `rule_id`: the id of the rule that matched on the other sensor, and `packet_id` the id of the packet that matched on the other sensor) and does not return anything but simply changes the state of the sensor. The array `update_state_list` (line 13) is a list of function pointers of the type specified above, one for each rule.

output: a function that transforms the matching packet into the information to be sent to the control node. This function has type `transform_list_t`(line 1 in the snippet above), that takes as input a `Packet *` (the packet that matched the rule) and returns a new raw packet `unsigned char *` with the relevant information folded in. The function takes another input parameter, that is, the rule id, needed to craft the packet and give in output the length of the crafted packet too. The packet returned by this function is ready to be sent, once padded with the relevant Ethernet source (the sensor's *MAC* address) and destination (Ethernet broadcast) address. The array `sensor_transform_list` (line 10) is a list of function pointers of the type specified above, one for each rule.

In addition to the information above, some other ancillary information is associated to each rule, that is:

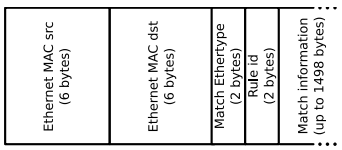
1. A 2 byte code that has to be written on each packet broadcast to the other sensor. The array `sensor_rule_codes` (line 16) contains this information for every rule.

2. A 2 byte code that has to be written on each packet sent to the control node. The array `control_node_rule_codes` (line 15) contains this information for every rule.

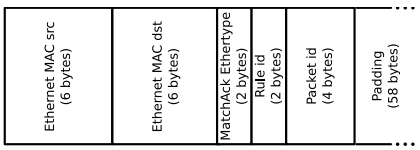
3. a flag that tells whether the rule needs to be acknowledged by other sensors or not. The array `need_ack` (line 18) contains this information for every rule.

Control packets are sent/received through the packet socket interface [mp] set in non-blocking mode. Packet sockets allow packets to be passed to and from the device driver without any changes in the packet data. When transmitting a packet, the user supplied buffer should contain the physical layer header. For this reason we defined a format for packets that are exchanged between sensors and from sensors to the control node.
In Figure 5.4.2 is shown the format of the three types of packet exchanged among sensors and sent by the sensors to the control node. It has to be noticed that the protocol type code (ethertype), is custom-defined using available values [IEE] .

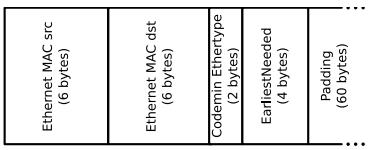
```
#define CODE_MIN_BROADCAST 0x9090
#define MATCH_ETHERTYPE 0x0300
#define MATCH_ACK_ETHERTYPE 0x0301
```



(a) Match packet format



(b) MatchAck packet format



(c) Earliest needed packet format

Control packets are standard Ethernet packets with a custom ethertype address at bytes 12-14 that identifies the type of information that the packet carries. More precisely,

- *Match* control packets have a 2 byte rule code encoded at bytes 14-15, followed by specific rule parameters.
- *MatchAck* control packets have the value of the matched packed id encoded in bytes 16-19.
- *earliestNeeded* packets have the value of *earliestNeeded* encoded in bytes 14-17.

It has to be noticed that packets that could result having a length smaller than the minimum required by the Ethernet standard (64 bytes) have to be padded.

Putting It All Together

Now we put all the previous consideration together by following step by step the data flow of a traffic packet through the parallel rule matcher.
As explained before, once a packet is captured from the traffic, it is decoded by the `Snort` decode engine and then put in the sensor INMB. The function in charge of doing this is `put_buffer`, which inserts the packet in a circular buffer, ordered by PacketId.
After this step, the packet is passed again to `Snort`, which check it against its preprocessors. If the packet matches a rule on a preprocessor parallel-enabled then the preprocessor itself will call the notification function `notify_change` passing the id or the rule that matched. The `notify_change` function knows implicitly that the packet that matched the rule is the last packet inserted in the buffer (we record into a variable the last id inserted) and subsequently does two operations: it modifies the *pendingMatches* queue and it broadcasts the information regarding the match, as the bully algorithm prescribes.
It has to be noticed that in our implementation there is not any data structure that represents the pending matches. The *pendingMatches* queue is implicit, and the information about matches is stored in the relevant `Packet` structure of the matching packets that are in the INMB. Since the INMB is ordered by PacketId and that all the packets matching at least one rule have to be in the INMB (the sloppy algorithm prescribes this), this way of storing information about matches is identical to the one that an explicit *pendingMatches* queue would provide.
The function that should update the *pendingMatches* queue then simply adds a new $RuleMatchInfo$ to the matching packet in the sensor INMB.

After that, a packet is crafted to store the information about the match in the way described above and then it is broadcast to other sensors.
The captured packet then continues its journey through the `Snort` preprocessors and detection engine.
In addition to the mechanism described above, every time a new packet is received an event timer (i.e., an event counter that fires some action periodically) is updated. The events associated with the event counter are related to the management of the *earliestNeeded* and INMB synchronization, more precisely:

- If the sensor is the current holder of the minimum *earliestNeeded* (elected through the bully algorithm) then it broadcasts its *earliestNeeded* every time a defined event-time expires (50 packets received, in our setup). Sensors broadcast their *earliestNeeded* by crafting and sending a single Ethernet frame with a custom ethertype and the *earliestNeeded* value padded just right after the header (see Figure 5.4.2).

- For each sensor, if a given event-time is expired since the last change seen in the advertised *earliestNeeded* operated by other sensors (or itself) then the *earliestNeeded* seen is considered stable (i.e., becomes the *discardPtr*) and packets earlier than it in the INMB are discarded. Before being discarded, each packet that matched at least one rule is encoded and sent to the control node.

When a packet is received on the control interface its ethertype is checked, this can have 3 different values:

- If (ethertype=CODE_MIN_BROADCAST) then the received control packet carries the *earliestNeeded* sent by some sensor. In this case, the bully algorithm comes to play and if the receiving sensor has a lower *earliestNeeded*, then that value is broadcast back, otherwise, the value is stored. If after a suitable time this value remains stable, it will be considered the new *discardPtr* and used to prune the INMB buffer. While pruning the INMB, each packet that should be discarded is checked for rules that matched it. For each rule found, a message is sent to the control node to signal that the match occurred. In this situation, the `control_node_transform_list` array comes to play. A new raw packet is created using the function associated with the matched rule that extracts the relevant information about the match from the packet and sends them, crafted in a new raw Ethernet frame, to the control node.

- If (ethertype=MATCH_ETHERTYPE) then a match notification is received and the correspondent changes to the state are performed. Those changes do not have to trigger a new notification to the other sensors. If the packet id associated with the match is earlier than *lastInspected* then the INMB buffer is scanned after having updated the state and any new match caused is broadcast. Packets that have already matched a determined rule are prevented to match the same rule again. If no new packets have matched then a *MatchAck* packet is sent back in unicast to the sensor that broadcasted the match.

- If (ethertype=MATCH_ACK_ETHERTYPE) then an acknowledgement to a match is being received. The sensor searches for the packet id whose match is being acknowledged, and, for that packet, the RuleMatchInfo with the correspondent rule id is searched. When the correct `RuleMatchInfo` entry is found, its field `ack_host_bitset` is updated with the id of the sensor that acknowledged the match. If all the sensors have acknowledged the match an acknowledgement is sent to the sensor that caused the match that triggered the current match to occur.

From the point of view of the communication among the sensors, it has to be noticed that:

- *earliestNeeded* and *Match* control packets are sent to Ethernet broadcast address and have as a source address the *MAC* address of the sensors that sent them.
- *MatchAck* packets are sent in unicast to the sensor that broadcast the match only and have as Ethernet source address the *MAC* (Media Access Control) of the acknowledging sensor.

We have given a precise characterization of how the sloppy rule matching algorithm has been implemented for the sensor nodes. In the next section, the logic of the control node will be taken into consideration.

5.4.3 Control Node

The logic implemented in the control node is rather simpler compared with the one taken into consideration for the sensors.
As for the sensors, the control node is based on `Snort`, in order to share the debug functionalities and the sensor identity management with the sensors' codebase, but actually it exploits a negligible part of `Snort` itself.
Loosely speaking, the control node can be seen as a sensor that does not capture traffic and works only by processing packets received on the control interface.
The control interface receives packets through a packet socket set in blocking mode (we do not need the non-blocking mode here, since there are no concurrent captures on multiple interfaces to be performed).
The control node must process the packets ordered by packet id, and, since packets can arrive out of order to the control node because they followed different paths through the sensors, the control node needs to reorder the packets upon receiving them and before processing them.
This task is accomplished using a hybrid event-timeout/real-timeout based mechanism.
For each packet, we assume that no other packets with a smaller packet id can arrive after a certain event-span [4] (currently 100 packets). This means that we use a queue in which we store the last 100 "valid" packet received sorted by packet id. Every time a new packet

[4] as for the sensors, we measure the time as number of event occurred, in this case number of packet received

is received, it is inserted into the queue and if the queue size becomes greater than 100 then the oldest packet is removed from the queue and processed. If a received packet is older than the oldest one it is declared "not valid" and discarded. Therefore, each received packet is passed to the sensors with a delay of 100 packets. This could be too slow in case of signatures that do not match often and could cause a reaction lag of the system over acceptable limits. In order to avoid this, a pure timeout system enforce the depletion of the queue if no packets have been arrived for a fixed timeout (0.5 seconds in the current setup).

Chapter 6

System Evaluation

6.1 Emulation

The development and the earlier tests of the parallel rule matcher have been performed in an emulated environment.
As stated in Chapter 5, this environment was designed to mimic a real network deployment in a transparent way. Neither the sensors nor the control node processes should be aware of being run in an emulated network environment. In this way, once tested in a user-friendly manner in the emulated environment, the system can be transposed into a real network without changing its configuration.
The emulated testbed was useful during development and debug sessions since it is self-contained and a single laptop can seamlessly emulate a system with thousands of sensors. Of course, the emulated system does not help study the performance and scaling properties of the model since emulated processes, hubs and switches do not work in real concurrence. Therefore, time measures are not relevant.
In order to have relevant measurements with respect to time and performance on an emulated system we should have introduced a virtualized time notion to take into account the fact that concurrence in the emulated system is not real, but, instead it is emulated by multiprocessing/multithreading. In such a system, the time as perceived by the single processes has not anything to do with the "real" time so direct time measurements on the emulated system are meaningless from the performance point of view. Adding the concept of a virtualized time would have turned our emulation into a *simulation* and would have made things harder to implement. One of the greatest issues with time virtualization would have been the estimation of the time spent in active network components, such as hubs and switches, which would have implied having a mathematical model for those. Such kind of simulation problems are an open research field and implementing a model from scratch for our environment, only for testing purposes, would have been overkill.
Following the above considerations, we have chosen to use the emulated environment only for debug and consistency checking, while we take into consideration a real-network deployment to perform measurements regarding performance and scaling properties.

6.1.1 Network Virtualization

There are different tools, available in the open-source community to build emulated network environments. Research on network virtualization has recently become more and more important, fostered by the wide development of machine-virtualization techniques, such as Xen [Bar03, MST+05] and Qemu [Bar06], and the increasing development of "userland networking" techniques (e.g., OpenVPN [Fei06]).
Although many network virtualization techniques are available on the market and in the open source community for the aforementioned reasons, it has been difficult to find a solution suitable for our needs, because of various reasons.
Almost all virtualization systems are designed and tweaked towards layer III networking (e.g., *IP*) and above, while our system is Ethernet-based. This is for instance the case of the `python twisted` framework [Kin05], which is a powerful networking engine written in Python, supporting numerous protocols but designed mainly for application-level protocols. Twisted includes some tools for raw level II networking (see the plugin formerly called `twisted-pair`) but this part is not actively developed anymore.
In addition to that, we were looking for a system that would be easily-customizable in order to emulate communication failures, slow downs, etc. This has excluded solutions such as OpenVPN [Fei06], which is a full-featured *SSL VPN* solution that can accommodate a wide range of configurations, but is a well-established large project, which would be too complicated to customize.
As a last constraint, we wanted a system easy to setup and install, with a small memory footprint, and low *CPU* utilization. This requirement of course avoids the utilization of any full-blown *OS* virtualization systems, such as *UML* (User Mode Linux) [Dik06], or over-featured virtual network infrastructure such as VINI [BFH+06].
Instead, we found very interesting for our needs a small (less than 10K lines) C toolkit called *VDE* (Virtual Distributed Ethernet). Before describing the toolkit and the enhancements we have implemented, we first draw some consideration of what the fundamental parts of an emulated network are.

6.2 Emulated Network Topology

In brief, a network is made of nodes and links, so we need to emulate both these elements in our virtual network framework. In our setup the nodes are the network interfaces from which traffic packets and control packets are captured while the links are connections of two types:

- one-to-one connections between two interfaces,
- one-to-many connections that pass through an active network component, such as a hub or a switch.

TUN/TAP Device

In order to emulate network interfaces we used the TUN/TAP device driver available for Linux>=2.2. In particular, we used the TAP Ethernet network device. An excerpt of the TUN/TAP device driver documentation follows.

> TAP driver was designed as low-level kernel support for Ethernet tunneling. It provides to userland applications two interfaces:
>
> **/dev/net/tun:** a character device;
>
> **tapX:** a virtual Ethernet interface.
>
> Userland applications can write Ethernet frames to `/dev/net/tun` and the kernel will receive this frame from `tapX` interface. At the same time, every frame that the kernel writes to the `tapX` interface can be read by userland application from the `/dev/net/tun` device.

Using the TAP devices we can have up to 255 interfaces on a single machine. Those interfaces are similar to real Ethernet interfaces and we can read from them using packet sockets or capture raw Ethernet frames by means of `libpcap`.
Intuitively, TAP devices are a good fit for emulated network interfaces and we made sensors and the control node read from those interface. Now it remains to implement in the virtual framework the link abstraction. This has been achieved using the Virtual Distributed Ethernet.

Virtual Distributed Ethernet

VDE is a software component developed by Prof. Renzo Davoli (University of Bologna). Using the author's words [Dav04]:

> VDE is a kind of Swiss knife of emulated networks. It can be used as a general Virtual Private network as well as a mobility support technology, a tool for network testing, a general reconfigurable overlay network, a layer for implementing privacy preserving technologies and many others.

Among the advantages of *VDE*, we have that it is designed following the KISS principle verbatim. It is a collection of simple and modular utilities implemented in a clear and stripped down -yet fully functional- way. It has been very easy to modify *VDE* source code in order to introduce our enhancements.
The main components of *VDE* are described below.

vde_switch is a component that mimics the behavior of a physical Ethernet switch. It has several ports that can be connected with Ethernet-compliant equipment, and, therefore, with a TAP interface.

vde_cable is the virtual counterpart of an Ethernet cable and can be used to connect together two vde_switches.

It is easy to see that `vde_switch` fits perfectly the role of a virtualized network link, and, as a matter of fact, we used it to connect the virtual interfaces to each other.
More precisely, we deployed a virtual network architecture where each sensor reads control packets and capture traffic from two TAP interfaces and the control node receives packets from a TAP interface.
All the control interfaces have been connected to each other with a `vde_switch` and the traffic is sent from the splitter to the sensors' capture interfaces through another `vde_switch`.

Enhancements to VDE

As described in Chapter 4, in our setup packets are obtained from a tap (T) into the high-speed network link(s). The tap copies the packets and sends them to a packet numbering unit (PNU), which in turns, forwards them to the splitter. The splitter sends the traffic to the sensor nodes in charge of performing the detection operations.
In order to perform the round robin distribution (or any other form of distribution) of the traffic we used a combination of an Ethernet frame rewriter and a switch.
The frame routing to the sensors is accomplished by rewriting the frames Ethernet destination address with the *MAC* address of the sensor that should take care of its analysis and then by sending them to an opportunely configured switch. For instance, given that mac_array is an array that contains the sensors' capture interface *MAC* addresses and N is the number of sensors deployed, by rewriting the i-th frame destination address with the value of $mac\_array[i\%N]$ one can partition the traffic in a round-robin fashion.
The Ethernet frame rewriter mechanism can be incorporated in the packet numbering unit, which modifies the Ethernet header as well. As a matter of fact, the packet numbering unit tag frames with an increasing number but cannot add an additional field to the packets in order not to incur into fragmentation for packets that have size close to the link *MTU*.
In summary, by using a frame rewriter and a switch we can perform the packet splitting as described in Chapter 4.
The traffic-distributer switch has to be configured with static routes. The switch needs to know in advance the Ethernet *MAC* addresses of the sensors capture interfaces and does not need any auto-learning mechanism because packets flow only in one direction (i.e., from the splitter to the sensors). Indeed, an auto-learning mechanism would be harmful because the frame Ethernet source address does not represent a real *MAC* address (it has been rewritten by the PNU), and, therefore, the switch would spend time and resources in order to remember the supposed-to-be sources addresses, resulting in a performance decrease.
Since version of *VDE* that we used (vde2-2.1.1) does not implement static routes for the `vde_switch` tool, we implemented this feature.
More precisely, a new command line option (`static_route`) has been added to `vde_switch` in order to set the software switch in static route mode.
Second, the logic to acquire the *MAC* addresses and to use them to forward frames has been added. Since `vde_switch`, when connecting to TAP interfaces, needs to know the name of

the interfaces it connects to, we exploited this knowledge in order to have it retrieving the *MAC* address of the interfaces automatically.
In details, we modified the function `open_tap`, whose task is to open the `/dev/net/tun` device and to associate it to an interface specified in the command line, in order to also retrieve the interface *MAC* address. The code snippet below show how this is done.

```
1  int open_tap(char *dev, unsigned char * mac)
2  {
3          struct ifreq ifr;
4          int fd, s;
5
6  ...
7          if ((s = socket(AF_INET,
8               SOCK_RAW, htons(ETH_P_ALL))) < 0)
9            {
10             perror("socket");
11             return -1;
12           }
13
14         memset(&ifr, 0, sizeof(ifr));
15         strncpy(ifr.ifr_name, dev, sizeof(ifr.ifr_name) - 1);
16         if (ioctl(s, SIOCGIFHWADDR, &ifr) == -1) {
17           printlog(LOG_ERR, "SIOCGIFHWADDR failed %s",
18                    strerror(errno));
19           return(-1);
20         }
21         close(s);
22
23         memcpy(mac,&ifr.ifr_hwaddr.sa_data,ETH_HEADER_SIZE);
24 ...
```

The function prototype has been modified in order to take an additional output parameter: the *MAC* address of the tap interface opened. The logic in charge of retrieving the interface *MAC* address is in lines 7-21.
After the retrieval, a simple linked list is created with all the *MAC* addresses of the TAP interfaces, and, upon packet forwarding, if the `vde_switch` is working in `static_route` mode, the list is scanned to find the port to which the incoming frame has to be forwarded according to its Ethernet destination address.
We are aware that a liner search on a linked list is not very efficient and that a hash table would perform better but since we are operating with a number of sensor lesser than 10 the cost of a linear search is totally affordable.
Aside from the logic to handle the list of *MAC* address, the only modification that remained to be performed on the `vde_switch` source code was to change the logic in packet forwarding when `static_route` mode is set.
In details, the switch memory update (performed by calling the `find_in_hash` function)

has to be avoided if `static_route` mode is on (see the additional disjunction in the if condition at line 2):

```
1 if(    !((IS_BROADCAST(packet->header.src)) || (pflag & HUB_TAG)
2        || static_route ) )
3  {
4
5   int last = find_in_hash_update(packet->header.src,vlan,port);
6  ...
```

The `find_in_hash` function, whose task is to find the port to which forward frames when the switch works in dynamic mode, has been modified in order to search in the static route list, instead of the address-port hash table, if the switch is working in `static_route` mode. The code that performs that in the snippet below is the condition at line 4 that forces the lookup in the static route list if the relevant flag is set.

```
1  int find_in_hash(unsigned char *dst,int vlan)
2  {
3    int p;
4    if (!static_route || (p=lookup_static_route(dst))<0 )
5      {
6        struct hash_entry *e = find_entry(extmac(edst,dst,vlan));
7        if(e == NULL) return -1;
8        return(e->port);
9      }
10   else
11     return p;
12 }
```

Thanks to the simplicity and flexibility of the *VDE* codebase it has been easy to patch it in order to add the aforementioned functionalities.

6.2.1 Python Automatic Test Framework

We setup a couple of Python scripts that could help creating a development testbed for the parallel rule matcher deployed in the virtual network framework.
We now describe the functionalities of those scripts without entering in the implementation details.
The first script creates the virtual network testbed, resembling the infrastructure defined in Chapter 4 and 5. It is basically driven by a single parameter: the number of sensors to be created, say N. The script creates $2N + 2$ TAP interfaces ($tap_i \ldots tap_{i+2N+2}$, where i is the first identifier available) and two `vde_switch`. Optionally, it is possible to assign a specific address to the TAP interfaces. If not provided, *MAC* addresses will be assigned to the virtual interfaces in increasing order starting from `00:00:00:00:00:01`. The two `vde_switch`es created by the script mimic the functionalities of the distribution switch and of the control switch. The distribution switch is started in daemon mode, with static

routes and connects together interfaces tap_i and $tap_{i+1}, tap_{i+3} \ldots tap_{i+2N-1}$, while the control switch, started in daemon mode and with static routes as well, connects together interfaces $tap_{i+2}, tap_{i+4} \ldots tap_{i+2N}$ and tap_{i+2N+1}. Interface tap_i is used by the splitter to inject frames, interface tap_{i+2N+1} is the control interface of the control node, and other interfaces arex the capture and control interfaces of the sensors, as described in Figure 6.1

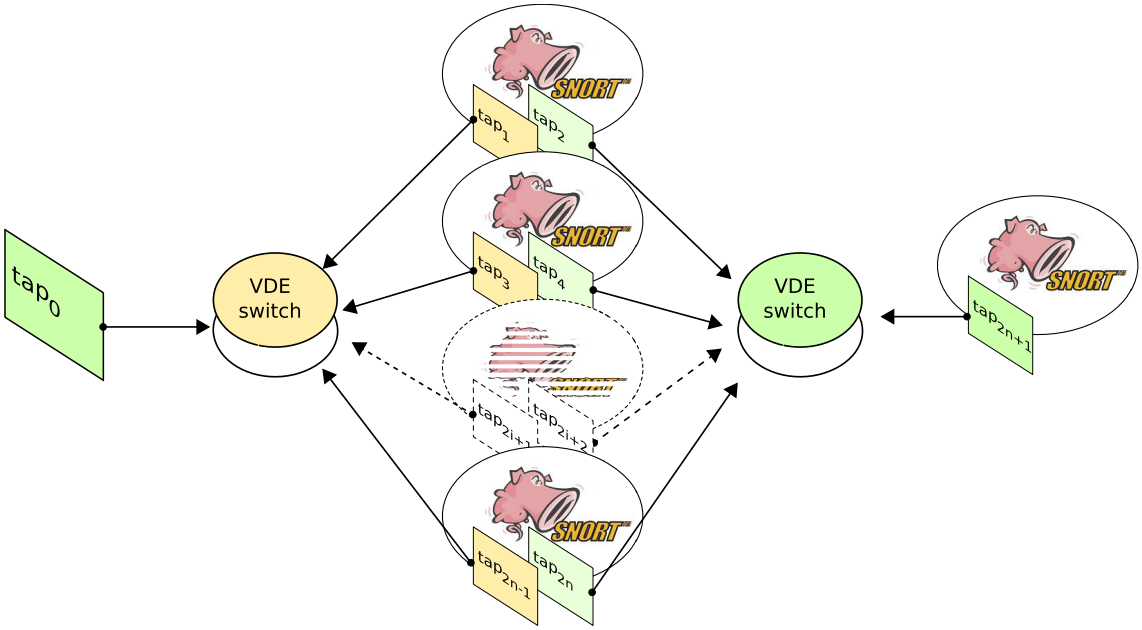


Figure 6.1: Virtual topology for the rule matcher, as created by the helper scripts.

In addition, the script logs operation to console, performs clean exit on errors, and save its working state (i.e., `vde_switch` daemons' PID, interfaces names, etc.) along with a description of the network topology created, to a configuration file. This make it possible for the script to stop the testbed by killing the `vde_switch` daemons and to delete the tap interfaces previously created. The script is designed in such a way that it is possible to run concurrently more than one testbed on the same host (provided that the total number of TAP interfaces created is less than 255).
Another script is used to effectively start the `Snort` sensors and control node concurrently attaching them to the virtual topology previously created. This script reads the network topology description created by the previous script and actually runs the sensors and the control node by passing the right parameters (in terms of which interfaces to reads or capture from) to the `Snort` processes. By means of the `select` primitive, this script multiplexes output and errors logged by all the nodes into a single stream of logs, making it possible to debug the concurrent system as it was a serial one. In addition, the script can run all the `Snort` processes into `valgrind` (extremely useful when debugging and chasing memory leaks).
Both the scripts described helped ease a lot the development processes.

Testing the System

To test whether the system was functioning properly during development we used the aforementioned scripts in conjunction with two powerful tools: `scapy` [bio] and `tcpdump` [MLJ02, tcp02].
`tcpdump` is a well-known and widely-used tool for network monitoring and data acquisition. `tcpdump` uses `libpcap` to capture traffic. The traffic of interest can be specified by command line arguments. `tcpdump` has been very useful in the development of the parallel rule matcher to understand which packets were arriving or departing from an interface. `tcpdump`'s powerful filter capabilities were useful for instance to select only packets coming from a given address, or class of address, in order to have a partial view of the system.
The other extremely important tool used during development and testing is `scapy`, a tool by Philippe Biondi. To use the words on the `scapy` website:

> Scapy is a powerful interactive packet manipulation program. It is able to forge or decode packets of a wide number of protocols, send them on the wire, capture them, match requests and replies, and much more

A strong good point in favor of `scapy` is that it is implemented in Python and comes with an interactive shell that resembles the default Python shell. In this way it is very straightforward to forge packets of any kind using a simple and intuitive syntax. We mainly used `scapy` to inject raw Ethernet frames into the various interfaces and see what the results were. For instance, with this simple command:

```
for i in [1,3,5,7]:
    sendp(Ether(dst='AA:BB:CC:DD:EE:0%d'%i,src='aa:00:00:00:00:0%d'%i),
    iface='tap0')
```

four Ethernet frames are forged and sent to four different *MAC* addresses. Without `scapy`, the same functionalities could have been implemented in C using the `libnet` [Foo] library in a far more complex way.

6.3 Real-world Network Setup

As we said in Section 6.1, the emulated network testbed was developed mainly for testing and debugging purposes, and not for a thorough performance analysis. In order to have such an experimental verification, we setup a real-world network testbed. From the networking point of view, the system has been built following verbatim the architecture described in Chapter 4 and already implemented in the emulated environment. More precisely the system was composed of two networks:

1. the distribution network, through which packets are sent from the splitter to the sensors. This is an Ethernet networks with no *IP* (Internet Protocol) stack level. The splitter has a high throughput network interface connected to a switch configured with static routes from the interface to which the splitter is connected towards the outgoing sensors interfaces.

2. The control network, through which the sensors exchange control packets with each other and with the control node. Sensors and the control node are connected through a hub since we do not have any high throughput requirements for control traffic, only a low latency is desirable.

Port Scan Specialization

From the software point of view, we installed a stripped down -yet representative- version of the algorithm described in Chapter 5.
On the sensors' side, `Snort` listens on the traffic interface. The `Snort` running on the sensors have two parallel-enabled preprocessors loaded, namely: `stream4_reassemble` and `sfportscan`, configured as below:

```
preprocessor stream4_reassemble
preprocessor sfportscan: proto  all
  scan_type  all
  watch_ip  X.Y.Z.W/24
  memcap  10000000
  sense_level  low
  logfile  /var/log/snort/scan.log
```

These two preprocessors have been modified in order to support some of the features described in Chapter 5. We analyzed their behavior by following the journey of a packet captured from a live interface.

1. When a new packet is captured on the live interface it is passed to the `Snort` decoding engine.

2. Before sending the packet to the preprocessor (`stream4_reassemble` and `sfportscan`) each sensor listens (in non-blocking mode) for incoming packets on the control interface and process them, if any.

3. Packets received from the live interface are sent to the `stream4_reassemble`. If the packet changes a state of the reassembly machine (i.e., initiates, tears down, or modifies the state of some connections), which in this example incarnates the scanning state described in our theoretical framework described in Chapter 4, then it is broadcast to the other sensors as a *Match* beacon.

4. Packets received from the control interface (coming from other sensors) are injected into the stream reassembly preprocessor in order to re-create the same state changes and then discarded.

5. Packets sent through the stream reassembly preprocessor are then handed out to the port scan preprocessor. At the beginning of the process, some tests are performed on the packet in order to exclude benign packets (test for necessary conditions for

attack). One of this tests consists in checking if the packet is part of a legitimate connection. If this is the case, the packet surely cannot be part of a scan and thus no further check is performed on it.

Here is where the scanning state of the `stream4_reassemble` preprocessor is exploited. Packets that are not tagged as benign at this stage do not go through a complete analysis against port scan detection heuristics but are simply forwarded to the control node that takes care of them. Sensors do not maintain any state associated with the port scan plugin, such as scanner and scanned lists.

6. The control node has only the `sfportscan` preprocessor loaded and does nothing more than letting the packets received from sensors going thoroughly through it. Control node does not know anything about the preprocessing steps carried on at the sensor level and uses the port scan preprocessor thoroughly, maintaining the state associated to it.

It is easy to recognize that the aforementioned setup is a particular case of the general architectural model described in Chapter 4. The preprocessor `stream4_reassembly` on the sensors keeps the predicate state, while the `sfportscan` plugin performs the rule predicate evaluation based on that. The working state is only kept in the control node `sfportscan` preprocessor in the form of all the relevant structures maintained by the plugin itself, such as scanner hosts and scanned hosts lists.
Note here that this experimental setup is a very stripped-down, yet working, incarnation of the theoretical framework proposed, and, in particular:

- There is only one signature (port scan) that is checked; the `stream4_reassemble` preprocessor does not check the traffic against any signature but only keeps the state, but from the modeling point of view and the subsequent analysis, we consider them both as rules, but with different side effects.

- The semantic of the inter-node communication is the same proposed in algorithm 4 but the syntax is slightly simplified. In particular, transformation functions as described in Chapter 5 have singular values, which can be seen in Table 6.3.

	control_out	control_in	output
stream reassemble rule	identity	identity	null
port scan rule	null	null	identity

The *identity* function takes a packet as input and returns the same packet in output, while the *null* function takes a packet as input and returns the null string ξ in output. The stream reassembly rule broadcasts state changes to other sensors by simply forwarding to broadcast Ethernet address the packets that have matched; in this sense, the function applied to packets before being broadcast is the identity function. In the same fashion, sensors process the control packets forwarded by other

sensors by simply handing them through their instance of the `stream4_reassembly` preprocessors with no changes, that is, they evaluate an identity function on those packets.

Packets that cause a `stream4_reassembly` change are not sent to the control node, thus formally we can say that the output function for that rule is null. In the same fashion, since portscan rule is no more than a predicate evaluation, it does not cause packets to be broadcast to other sensors (null control_out and control_in functions) and forwards matching packets to the control node unchanged (identity output function)

- There is no *matchAck* for the match messages, or said it in other words, rules have the flag *need_ack* set to off.
- There is no INMB on the sensors. Since packets that can update the stream reassembly machine usually belong to the *TCP* three-way handshake, those are supposed to be to usually fairly distant in time from each other, compared to the times involved in the communications among the sensors. This simply means that we assume that the state-changing packets are processed by the sensors in the right order, as they would be processed by a serial machine, and, if this assumption holds, there is no need for the INMB. More formally, we can say that even thought the packet stream can undergone a local reordering in being concurrently processed by sensors, the subsequence of the state-changing packets comes to the sensors in order. We will see later on this chapter that this assumption does not hold under certain conditions.

The above considerations show that the evaluated module is an instance of the general one proposed in Chapter 4 and implemented in Chapter 5.

Evaluation Dataset

In order to evaluate the system, we have crafted an artificial dataset by merging the standard evaluation dataset IDEVAL[Lab98, Lab99, LFG+00, McH00] (attack free) with a dump of a port scan performed against a host inside the home network.
The port scan has been performed by the tool `nmap` (see Chapter 3) against the host X.Y.Z.W issuing the following command:

```
nmap -p20-25,80,443 X.Y.Z.W
```

The IDEVAL dataset used was attack-free and composed by 1.37M packets of various protocols. It has been merged with the port scan dump by means of a modified version of `tcpmerge` [Bev].
`tcpmerge` merges two `libpcap` traces with respect to time. The two traces we merged are not time related and port scan dump would have been appended in queue to the IDEVAL traffic, since recorded years later (IDEVAL traffic used was recorded in 1999). In order to avoid this, we have modified the `tcpmerge tool` to make it merge traces regardless the

time and distributing the attack packets evenly over the attack-free traffic. This resulted in having approximately one attack packet every 65K attack-free packets.
As a last artifact modification of the traffic packets, we rewrote the *MAC* address of every frame in order to induct different scattering pattern on the splitter. More precisely, we created two testsets with different characteristics:

1. Ethernet frame destination addresses of the traffic image have been rewritten in order to cause the distribution switch to send the packet round-robin to sensors. This has been accomplished by setting the destination address of the i-th frame as $mac[i\%N]$ where $mac[\cdot]$ is a list of the Ethernet *MAC* address of the sensors' capture interfaces and N is the number of the sensors.

2. Frame destination addresses have been rewritten according to a hash calculated on the *TCP* source and destination port and *IP* source and destination address. More precisely, the destination address of the i-th frame has been set to $mac[hash(P_i)\%N]$ where P_i are the aforementioned parameters for the the i-th frame and $hash(\cdot)$ is a hash function that returns integer numbers as a result.

The *MAC* rewriting operations have been accomplished by the `tcprewrite` [Turb] tool. `tcprewrite` is a tool to rewrite packets stored in `pcap` file format and provides many ways to modify different packets fields. We modified `tcprewrite` in order to rewrite Ethernet destination address in the manner described above.
More precisely, we added a function that take as input packets as read by `tcprewrite` and returns in output an integer hash [Wan07] of the sum of the *TCP* source and destination ports and the *IP* source and destination addresses [1].
The program has been also enhanced with a new command line option to specify a file that holds a list of newline-separated *MAC* addresses, which are the sensors' *MAC* addresses. Additional logic has been put into work in order to actually write to the frames the addresses read according to the functions described above.
It has to be noticed that the aforementioned task could have been carried out using the by far more user-friendly `scapy` at almost no cost. Unfortunately the version (1.0.4.1beta) available at the time of the experiments had problems in reading the IDEVAL traces (bug in the *NTP* packets decoding).

6.3.1 Hardware

We set up four Linux boxes sensors running `Snort` 2.6 [Roe99] with the aforementioned preprocessors loaded in.
The function of the control node has been accomplished by another Linux box running a patched `Snort` with a vanilla version of the `sfportscan` preprocessor.

[1] We applied a hash function to scramble the bits of the aforementioned sum, which, taken as is and applied to the packets of the IDEVAL, resulted being strongly biased towards odd numbers (65% of the processed packets resulted in having an odd sum).

Another Linux box was used to send the traffic into the testbed through a SysKonnect SK-98xx Gigabit Ethernet network card connected to a 32-bit/66MHz PCI bus.
Each sensor box mounts two 3Com network cards, one to receive the traffic from the splitter and another one to communicate with other sensor and the control node, mimicking closely the virtual topology reported in Figure 6.1
Sensors are connected to the splitter box through a Cisco Catalyst 3500 XL switch configured with static routes to the sensors capture interfaces. The switch has a 1000-BaseT GBIC module that receive the traffic from the splitter output interface and is connected to sensors through FastEthernet ports.
The sensors control interfaces are connected to each other through an ordinary FastEthernet hub. Since we do not need high performance on the control networks but only low latency and, given that the traffic on that network should be fairly low, we can avoid using a switch in favor of a hub.

Splitter

Some words have to be spent about the splitter and the way we injected packets into the parallel rule matcher.
The traffic source activity has been emulated through `tcpreplay` [Tura], which was run on the splitter box to replay the traffic datasets through the Gigabit Ethernet output interface at a configurable rate. Note that the whole traffic dataset needs to be load into memory before being replayed[2] in order to avoid the communication between disk and memory, which could represent a speed bottleneck[3]. Even though we took into consideration this issue, in the following experiments the maximum throughput used will never be higher than 50Mb/s, since it was already sufficient to confirm our claims.

6.4 Experiments and Preliminary Results

We ran a simple experiment to validate the model. Even though the results are very promising, more testing is needed in the future to evaluate the system in different configurations. We compared the parallel rule matcher setup with a single instance of a vanilla `Snort` with the `stream4_reassemble` and `sfportscan` plugin loaded, patched and configured as explained in the previous sections. We expected that at a given traffic rate the single `Snort` would not be able to keep up with the traffic and thus would miss the port scan attack, while the parallel version of the sensor should still be able to catch it. We performed also other tests in order to evaluate the goodness of our findings against a control test set. A list of the test performed follows.

[2] In Linux this task can be accomplished transparently by copying the image traffic file into `/dev/shm/` and reading it from there. `/dev/shm` is a tempfs filesystem that lives in RAM. It has to be noticed that its maximum size has to be set consistently with the data that are copied into it.

[3] Disk performance is often the primary bottleneck to network performance. Disk bandwidth for ordinary *SATA* or *EIDE* disk is around 50 MB/s while a Gigabit Ethernet is in theory capable of $\approx$ 120 MB/s

single Snort: in this setup, the traffic has been replayed to a single box mounting a Gigabit Ethernet interface via a cross cable, first at low rate and then at a speed higher enough to cause packet losses.

parallel architecture with a single sensor: in this setup, the traffic has been replayed to a single-sensor parallel machine equipped with the aforementioned preprocessors. This deployment has been performed for testing purposes only. We aimed at testing the system eliminating concurrence issues among sensors while retaining the two-tier architecture layout.

In this case, as before, traffic has been replayed first at low rate and then at a higher speed to cause packet losses.

parallel architecture with four sensors: in this setup, we implemented the full-blown parallel architecture. This configuration has been tested in several ways, namely both with traffic sliced according to a hash calculated on the *TCP/IP* headers and simple round-robin, and both with low and high rate traffic.

We found that when traffic is replied at a low rate, all the tested configurations give the same output, that is, they successfully identify the port scan introduced into the data, regardless of the fact that if the traffic splitter was working in a round-robin or hash-based fashion.
It has to be noticed that other attacks have been identified, which have to be considered as false positives, since the traffic utilized should be attack free. With traffic replied at a lower rate the false positives identified are still the same. This gives us a strong hint that, regardless the topology and the setup, under light load, the systems behave in the same way.
When increasing the rate of replayed traffic, in the parallel setup with only one sensor 6.4, the sensor approximately forwards to the control node the 10% of the total packet count (148Kpackets) because recognized as "possibly dangerous", before starting missing packets. Recalling Equation 4.11, this would mean a degree of exploitable speedup of order 10, in the assumption that we have sensor and control node installed on the same kind of hardware (α=1, as expressed in Chapter 4).
Unfortunately, the set up with four sensors, i.e., the one that resemble more closely a real world incarnation of the system, does not show the same scaling properties when tested with the round-robin traffic.
As the traffic rate increases, each sensor forwards an increasing fraction of the received packets reaching 1/2 of the received traffic at a rate of 50Mb/s. Even though the sensors nodes do not get congested, since each of them receive 1/4 of the total throughput, the control node ends up processing an aggregate throughput of 1/2 of the one outgoing from the splitter. In this is situation, the control node becomes the real bottleneck of the architecture.
It is remarkable to understand that the performance decrease of the parallel engine are associated with a decrease of communication efficiency between the sensors, which occurs

	Low speed traffic (≈ 100Kb/s)	High speed traffic (≈ 50Mb/s)
Single `Snort` instance	port scan detected	port scan missed
Parallel architecture: 1 sensor	port scan detected	port scan missed
Parallel architecture: 4 sensors round-robin traffic	port scan detected	port scan missed
Parallel architecture: 4 sensors hashed traffic	port scan detected	**port scan detected**

Table 6.1: System evaluation results. It can be see that the only configuration able to keep up with 50Mb/s paced traffic is the parallel one, with four nodes (in bold).

when the traffic rate increases. This hypothesis has been confirmed by the fact that the experiment where the hash-based splitting was employed with the full-blown architecture went well, producing more or less the same performance of the single node `Snort`, that is, 1/10th of the traffic forwarded from each sensors to the control node, which therefore finds itself processing 1/10th of the total throughput.
The reason why the hash-based splitting method obtains better performances is subtle and depends on the state information that sensors exchange with each other. This state information is relative to initiated *TCP* flows, and helps the port scan plugin to avoid forwarding to the control node for inspection packets belonging to a legitimate connection. It is easy-understandable that if miscommunication about *TCP* flow information occurs among sensors , they will be led into having partial information about initiated connections and, therefore, will be less confident in discarding benign traffic. In the hash-based splitting configuration, each sensor is fed with packets that surely belong to the same *TCP* flow. Therefore, even though there are communication errors between the sensors, each sensor will still have the necessary information to decide if a processed packet is benign. In this case, the sensor will not forward it to the control node, since the check "does the packet belong to any legitimate flow?" is done using locally-acquired knowledge only, and not using knowledge coming from other sensors. It has to be noticed that this situation, in which sensors have different state, contradicts Fact 2, which states that the same scanning state has to be present on every sensors in the same form, but this is still acceptable, since Assumption 2, which states that the system does not rely on any predefined mapping between packets and sensors, and upon which Fact 2 is based, does not hold anymore.
Results are summarized in Table 6.1 and 6.2
Having said that, we have however a proof that the parallel machine can easily keep up without losing packets to a throughput of 50Mb/s, while, at the same throughput, the single sensor configuration drops packets and misses the `nmap` scan [4]. This clearly demonstrates

[4] As reported by other [KVVK02, Der05], we know that 50Mb/s is not the state of the art for intrusion

	Low speed traffic (≈ 100Kb/s)	High speed traffic (≈ 50Mb/s)
Parallel architecture: 1 sensor	10%	heavy packet loss
Parallel architecture: 4 sensors round-robin traffic	10%	50%
Parallel architecture: 4 sensors hashed traffic	10%	**10%**

Table 6.2: System evaluation results. The tables represents the percentage of the input throughput that reaches the control node. The parallel architecture with four nodes achieve the best performance leading to a factor of exploitable parallelization of 10.

the capabilities and the scaling properties of the parallel engine with respect to the single sensor configuration even though the specific reasons why the round-robin splitting system performed badly have not been clarified yet. We further investigate this point in the next section.

6.4.1 Further considerations on the traffic

Further analysis of the traffic utilized for our experiments unveiled some pitfalls in the dataset.
We have noticed that IDEVAL traffic is low-throughput, hence the probability of one or more connections to get interleaved in the dump is rather low. This in turn causes that the probability for a packet of being adjacent to another packet of the same connection to be very high.
This result in the fact that, very frequently, the relevant packets for a *TCP* three-way handshake are usually close to each other, if not adjacent, in the traffic image we used.
Therefore, if the traffic is replayed fast, in order to simulate a high-throughput link, and the splitter performs the round-robin distribution, the relevant packets that modify the *TCP* flow state of a sensor for the `stream4_reassemble` preprocessor will usually get to different sensors. In this situation each sensor hit by a state-change packet will broadcast its match to other sensors that, in most of the cases, will have already received and processed the other relevant packets. Since in our setup sensors did not have any INMB they cannot reconstruct the right order of scanning and therefore, in most of the cases they will result having "a reordered view" of the relevant packets for a connection initiation. These reordered

detection speed on a single sensor and we did nothing to tweak sensors in order to increase it. The emphasis of this work is put on the analysis of the scaling properties of a parallel architecture with respect to its centralized counterpart. The only assumption that matters here is that both the architectures are configured in the same way, no matter if optimized or not, in order to have a common baseline to perform the comparison.

packets will then be discarded by the `stream4_reassemble`, which simulates the *TCP* state machine, because they are considered bogus. As a result, the `stream4_reassemble` preprocessor misses the newly-initiated flows and the `sfportscan` preprocessor does not have enough information to consider following packets as benign since apparently they do not belong to a legitimate *TCP* flow.
The above considerations make us understand that replaying a low-throughput traffic faster is not equivalent to replaying real-world, high-throughput traffic. In real-world traffic, the physical speed limitations of network links and devices would have prevented that the packets part of the three-way handshake arrive at the sensors at the same time.
This phenomenon violates our assumption made in 6.3, which states that we suppose that the state-changing packets are processed by sensors in the right order, and confirms the importance of having the sensors equipped with an INMB managed as prescribed by the sloppy rule matching algorithm to simulate a quasi-ordered[5] view of unordered events.

[5]For a true ordered view the exact rule matching algorithm, as described in Chapter 4 is needed.

Chapter 7

Conclusion and Future Work

In the present work we have analyzed the feasibility and described a proof-of-concept implementation of a parallel, stateful intrusion detection system for high speed networks.
Our work starts in Chapter 1, with an outline of the problem of computer security, in which we have stressed the ever-increasing importance that this field has acquired over time.
In Chapter 2, we have provided an overview of the current solutions to computer security issues, focusing on misuse network-based intrusion detection. In this analysis, we put the emphasis on the fact that network speeds have increased faster than the speed of processors, and therefore, centralized solutions have reached their limit.
In Chapter 3, we have presented an in-depth analysis of the current research on high speed intrusion detection, underlining the fact that the majority of the solutions proposed so far are based on custom hardware and cannot be considered as a long-term solution. In addition, we have stressed that these solutions are not capable of in-depth, stateful intrusion detection analysis of traffic.
In order to overcome those limitations, in Chapter 4 we have proposed a new model based on the idea of dividing the traffic volume into smaller portions that can be thoroughly analyzed by intrusion detection sensors. This idea is not novel, since it has often been advocated by the high-performance research community as a way to distribute the service load across many nodes. The problem with this kind of solution is that any approach that uses a static partitioning algorithm needs to over-approximate the event space associated with a signature and, therefore, it is not able to effectively partition the traffic in the case of complex stateful signatures.
The key fact of our model is that the traffic partitioning is done without taking into consideration the signatures used by the intrusion detection sensors; therefore, the sensors themselves need to coordinate with each others in order to share the information necessary to detect multi-step attacks. In order to implement this concept, we have proposed a two-tier architecture composed of a group of intercommunicating sensors and a control node. We decomposed the stateful actions necessary to evaluate a signature and possibly update the sensors' state in two parts, one parallelizable and one inherently serial, and proposed an architecture in which we moved all the non-parallelizable operations to the control node. By doing this, we were able to exploit the highest possible degree of parallelism. We have

also proposed a protocol for the communications among sensors and between a sensor and the control node, and algorithms to support state synchronization. In addition, we have described a precise characterization of the bottlenecks that are inherent to the parallel matching system in the most general case.
In Chapter 5 we have provided and discussed an implementation of the proposed architecture based on the `Snort` intrusion detection engine, and developed optimizations to reduce the impact of the bottlenecks identified in the model characterization. The evaluation of the prototype implementation, described in Chapter 6, has been carried out on both an emulated and a real-world network environment. The architecture has been deployed on a real network using four Linux boxes as sensors and it has been tested for port scan detection in various configurations. The result of the evaluation have confirmed the validity of the theoretical model, since the parallel rule matcher, composed of four sensors, has successfully outperformed a single `Snort` instance, which we used as a baseline for comparison, performing the same kind of detection on the same dataset.

7.1 Future Work

The present work analyzes the problem of stateful high-speed intrusion detection under a new perspective. The present research in this field is based on the assumption that traffic can be sliced in a way that keeps all the evidence necessary for detection in one slice.
Under this assumption, no communication is needed among the sensors.
As stated more than once in this work, we have tried to remove from our model the previous assumption because many real-world multi-step attacks cannot match those conditions.
The novel model we have taken into consideration provides new solutions but, at the same time, opens new problems. We sketch limitations and further enhancements about the model, the architecture, the implementation, and the testing of the proposed system.
In our model, even thought we have been able to parallelize the major bottleneck, which is the traffic scanning process, we still have an inherently serial component, that is, the control node. The control node processing activities are usually less resource-demanding than the ones performed on the sensors but they still could become overwhelming for a single machine. A further improvement of our architecture would be to enhance the parallel model presented making it capable of distributing the load of control activities on more than one control node.
A more long-term improvement of the model would require redesigning it based on a strong characterization of the semantics of the signatures that it will encompass.
As a matter of fact, we have defined our model with performance in mind and in order to make if fit the known signatures, but it is not guaranteed to satisfy any possible existing signature. In order to make our system model more general, it would be helpful to construct it starting from a model of the semantics of attack signatures that systematically enumerates the different aspects that characterize attack signatures. In his work [Mei04], Meier proposes such a model, adapting Zimmer's model [ZU99] to benefit from already-gained insights in the Active Database domain. The first step towards defining a general-purpose

architecture for parallel intrusion detection should start from enhancing such a kind of semantic model, identifying in it the inherently non-parallelizable constructs, and extending them in order to support parallel operations. With such a kind of checklist for the development of a signature specification language in mind, it would be straightforward to design the parallel system model and architecture. We know that this work would require a great effort to be accomplished, but it could give a solid basis on which to build a signature description language implementable, with no additional efforts, on a parallel architecture. Another task leaved as a future work is to implement and evaluate the "exact rule matching" algorithm proposed in Chapter 4 with the same accuracy devoted to the analysis of the "sloppy rule matching" algorithm.

Focusing on the implementation of the architecture, even though, as described in Chapter 5, most of the skeleton of it has been already developed, we implemented only one signature. Some efforts should be spent in order to implement more rules to match the signatures provided by `Snort` in the parallel-enabled architecture. This does not mean that one has to take into consideration all the `Snort` rules present in the default ruleset, since those are stateless and therefore would not require sensors to communicate. Following this considerations, it would be very desirable to have a working rule implementation for co-ordinated port scans detection tailored on the work described by Gates [Gat06]. Since the event space of a co-ordinate port scan cannot be partitioned in any way, we could exploit the fact that our system is oblivious of how the traffic is partitioned to effectively implement it. In this fashion, sensors could be used, as in the proof-of-concept implementation of Chapter 5, to filter out benign traffic while the complex state required by Gates' algorithm (a heuristic inspired to the solution of the set coloring problem) could be maintained on the control node.

Last but not least, the proposed system should be more extensively tested, but, before doing that, it is necessary to implement more parallel-enabled rules.

Acknowledgements

As every step in our lives, this work would have never come to an end without the aid of the people who surrounded me giving help in many different ways, silently or loudly, upon request or spontaneously.
I would like to say thank you to everyone who helped and assisted me during this period. In particular I want to thank my advisor, Professor Giovanni Vigna, for having given me the possibility to work at the Computer Security Group of the Computer Science Department at the University of California, Santa Barbara, and to utilize the laboratory equipment for my experiments.
And for all the mornings spent surfing together at C street, too :)
Many thanks to Dr. Ashish V. Thapliyal, who spent a lot of time with me thinking and designing the subject of the present work, for his enlightening advice and all the fruitful time we spent discussing this subject. A big "thank you" goes to the Computer Security Group crew, that is Greg, Marco, Vika, Bob, Wil, Fredrik and Davide for being always kind and supportive to me, even when heavily bugged...
In Santa Barbara I found such a warm and homely place to live and at *UCSB* in particular such a peaceful, serene yet motivating working environment. I wish everyone could live the same experience and I hope that the Engineering department of the University of Pisa will foster this kind of experience in the future.
Another thank you is deserved by the Italian crew in Santa Barbara that is Marco, Davide and Giovanni again, Marco's wife, Francesca, and Fabio who with their genuine "Italianity", and sometime supported by delicious food coming from overseas, never let me miss my native country. A special thanks to Freebirds which fed me with its delicious quesadillas during the nights spent at the lab (Well, maybe my liver is not so thankful, though...)
Getting back to the eastern side of the Atlantic Ocean, I would like to thank the Scuola Superiore Sant'Anna for having supported me as a pupil during the last five years and to have let me meet some of the most important people in my life. Here I found Lucas, who supported me tirelessly during the time spent abroad and who has a dream, exactly like me, thanks to Pier with whom I shared not only rooms this last years, to Anto "the bunny" waiting to visit him in Sardegna, Ludo, the wizard, who helped me with LaTeX, C/C++, networks and everything is "keyboard driven" and to all the "take-it-easy" part of Sant'Anna: Ele, Ale, Sciacca, Thomas, Simona, and il Pornosauro.
Thanks also to the big family of Ask.com for which I have been working for the last two years and in particular to Tony, AG, GPD and Navid. I have appreciated very much when

you have "overlooked" my absences when I was busy with exams or writing my thesis.
A "big thank you" goes to all of my old friends from Romagna: we seldom see each other but in front of a bottle of wine (or more) everything is like back to high school. Thanks to "Il Dalimunt" for his irony and ability of being always kind and helpful when you are in need. Thanks to Stef for all those chatting during the last ten years and for every time I realized that his doubts and his uncertainties sounds always like mine, thanks to Cass, Neri, Duri, la Vale, la Titti, l'Alice, we'll see you at Eno', sometime.
Thanks also to Prof. Sebastiano Vigna, Prof. Roberto Grossi, Flavio, Giuseppe, Maurizio, Dario and all the friends I met at the IOI for letting me always feel like at home and for having opened to me such a huge amount of opportunities, which I would have never achieved without their help. Everything begun at the IOI.
I would like to thank Professor Gianluca Dini of the University of Pisa for having accepted to supervise me in this thesis together with Prof. Giovanni Vigna.
Last but not least, I need to say "thank you" to my parents who always taught me to never let it go and, saving the best for the last, to my beautiful girl Lara who had the strength not to kill me all the times I have gone out of mind in these last months (really, not that few...)

Bibliography

[Amd67] G. Amdahl, *Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities*, Proceedings of the AFIPS Conference, 1967.

[And80] J.P. Anderson, *Computer Security Threat Monitoring and Surveillance*, James P. Anderson Co., Fort Washington, April 1980.

[Ans04] Google Answers, *Facts / Statistics about COMPUTER SECURITY and LOSS OF DATA* , http://answers.google.com/answers/threadview?id=304308, 2004.

[Ant98] Antirez, TCP *idle scanning*, http://seclists.org/bugtraq/1998/Dec/0079.html, 1998.

[Axe99] S. Axelsson, *Research in Intrusion Detection Systems: A Survey*, Tech. Report 98-17, Dept. of Computer Engineering, Chalmers University of Technology, Sweden, August 1999.

[Axe00] S. Axelsson, *Intrusion Detection Systems: A Taxomomy and Survey*, Tech. Report 99-15, Dept. of Computer Engineering, Chalmers University of Technology, Sweden, March 2000.

[Bar03] P. Barham, *Xen and the art of virtualization*, Proceedings of Symposium on Operating Systems Principles (SOSP) 2003., 2003.

[Bar06] Daniel Bartholomew, *Qemu: a multihost, multitarget emulator*, Linux J. **2006** (2006), no. 145, 3.

[Bev] Rob Beverly, *tcpmerge.c*, http://momo.lcs.mit.edu/code/tcpmerge.c.

[BFH+06] Andy Bavier, Nick Feamster, Mark Huang, Larry Peterson, and Jennifer Rexford, *In vini veritas: realistic and controlled network experimentation*, SIGCOMM Comput. Commun. Rev. **36** (2006), no. 4, 3–14.

[bio] Philippe biondi, *scapy website*, http://www.secdev.org/projects/scapy/.

[Bis02] M. Bishop, *Computer Security: Art and Science*, Addison-Wesley, 2002.

[CER06a] CERT, *CERT/CC Statistics 1988-2006*, http://www.cert.org/stats/cert_stats.html, 2006.

[CER06b] CERT, *Secure Coding*, http://www.cert.org/secure-coding/, 2006.

[CIS01] CISCO, *CISCO Intrusion Detection System*, Technical Information, Nov 2001.

[CM06] Michele Colajanni and Mirco Marchetti, *A parallel architecture for stateful intrusion detection in high traffic networks*, september 2006.

[CSI06] CSI, *CSI/FBI Computer Crime and Security Survey*, http://i.cmpnet.com/gocsi/db_area/pdfs/fbi/FBI2006.pdf, 2006.

[Dav04] Renzo Davoli, *Vde: Virtual distributed ethernet*, Tech. report, 2004.

[Der05] Luca Deri, *ncap: Wire-speed packet capture and transmission*, Proceedings of 3rd IEEE/IFIP E2EMON, May 2005.

[Des02] Mamata D. Desai, *Distributed intrusion detection*, Master's thesis, January 2002.

[Dik06] Jeff Dike, *User mode linux*, 1st ed., Prentice Hall Ptr, April 2006.

[EVK00] S.T. Eckmann, G. Vigna, and R.A. Kemmerer, *STATL: An Attack Language for State-based Intrusion Detection*, Proceedings of the ACM Workshop on Intrusion Detection Systems (Athens, Greece), November 2000.

[Fau05] Sacha Faust, LDAP *injection*, 2005, SPI Dynamics.

[Fei06] Markus Feilner, *Openvpn: Building and integrating virtual private networks*, May 2006, pp. 1–258.

[Foo] George Foot, *libnet homepage*, http://libnet.sourceforge.net/.

[FV01] Mike Fisk and George Varghese, *Fast content-based packet handling for intrusion detection*, Tech. report, La Jolla, CA, USA, 2001.

[Fyo07] Fyodor, *Nmap – the network mapper*, http://www.insecure.org/nmap/, 2007.

[Gat06] C. Gates, *Co-ordinated Port Scans: A Model, A Detector and An Evaluation Methodology*, Ph.D. thesis, Dalhousie University, Halifax, Nova Scotia, February 2006.

[GM82] H. Garcia-Molina, *Elections in a Distributed Computing System*, IEEE Transactions on Computers (1982).

[IEE] IEEE, *Ieee list of ethertype values*, http://standards.ieee.org/regauth/ethertype/eth.txt.

[Inf06] Infosyssec, *The Security Portal for Information System Security Professionals*, http://www.infosyssec.net/infosyssec/security/intdet1.htm, 2006.

[ISS01] ISS, *BlackICE Sentry Gigabit*, http://www.networkice.com/products/sentry_gigabit.html, November 2001.

[JA] Matt Jonkman and James Ashton, *Bleeding edge threats*, http://www.bleedingsnort.com/.

[JMA00] Alan Christie John McHugh and Julia Allen, *Defending yourself: The role of intrusion detection systems*, *IEEE* software (2000).

[JPBB04] J. Jung, V. Paxson, A. Berger, and H. Balakrishnan, *Fast portscan detection using sequential hypothesis testing*, 2004.

[KHQ] Ying Chen Kai Hwang, Min Cai and Min Qin, *Hybrid Intrusion Detection with Weighted Signature Generation over Anomalous Internet Episodes*, IEEE Transactions on Dependable and Secure Computing.

[Kin05] Ken Kinder, *Event-driven programming with twisted and python*, 2005, p. 6.

[Kle] Amit Klein, *Blind xpath injection.*

[Kol] Mitja Kolsel, *Session fixation vulnerability in web-based applications*, February.

[KVVK02] C. Kruegel, Fredrik Valeur, G. Vigna, and R.A. Kemmerer, *Stateful Intrusion Detection for High-Speed Networks*, Proceedings of the IEEE Symposium on Research on Security and Privacy (Oakland, CA), IEEE Press, May 2002.

[Lab98] MIT Lincoln Lab, *The 1998 DARPA Intrusion Detection Evaluation*, http://ideval.ll.mit.edu/1998_index.html, 1998.

[Lab99] MIT Lincoln Laboratory, *DARPA Intrusion Detection Evaluation*, http://www.ll.mit.edu/IST/ideval/, 1999.

[LFG+00] R. Lippmann, D. Fried, I. Graf, J. Haines, K. Kendall, D. McClung, D. Weber, S. Webster, D. Wyschogrod, R. Cunningham, and M. Zissman, *Evaluating Intrustion Detection Systems: The 1998 DARPA Off-line Intrusion Detection Evaluation*, Proceedings of the DARPA Information Survivability Conference and Exposition, Volume 2 (Hilton Head, SC), January 2000.

[LZL+06] H. Lu, K. Zheng, B. Liu, X. Zhang, and Y. Liu, *A Memory-Efficient Parallel String Matching Architecture for High-Speed Intrusion Detection*, IEEE Journal on Selected Areas in Communication **24** (2006), no. 10.

[McH00] J. McHugh, *Testing Intrusion Detection Systems: A Critique of the 1998 and 1999 DARPA Intrusion Detection System Evaluations as Performed by Lincoln Laboratory*, ACM Transaction on Information and System Security **3** (2000), no. 4.

[Mei04] Michael Meier, *A Model for the Semantics of Attack Signatures in Misuse Detection Systems*, Information Security, 7th International Conference, ISC (K. Zhang and Y. Zheng, eds.), 2004.

[MLJ02] S. McCanne, C. Leres, and V. Jacobson, *Tcpdump 3.7*, Documentation, 2002.

[MMK05] Sebastian Schmerl Michael Meier and Hartmut Koenig, *Improving the efficiency of misuse detection*, Proceedings of the Second International Conference, DIMVA 2005, Vienna, Austria, July 7-8, 2005., Springer Berlin / Heidelberg, 2005, pp. 188–205.

[mp] Linux man page, *packet(7) - linux man page*.

[MS03] Kevin D. Mitnick and William L. Simon, *The art of deception: Controlling the human element of security*, John Wiley & Sons, Inc., New York, NY, USA, 2003.

[MSK01] S. McClure, J. Scambray, and G. Kurtz, *Hacking exposed: Network security secrets & solutions*, third ed., Osborne/McGraw-Hill, 2001.

[MST+05] Aravind Menon, Jose R. Santos, Yoshio Turner, (john) G. Janakiraman, and Willy Zwaenepoel, *Diagnosing performance overheads in the xen virtual machine environment*, VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments (New York, NY, USA), ACM Press, 2005, pp. 13–23.

[MVKV06] D. Mutz, F. Valeur, C. Kruegel, and G. Vigna, *Anomalous System Call Detection*, ACM Transactions on Information and System Security **9** (2006), no. 1, 61–93.

[Myr] Myricom, *Myrinet network technology*, http://www.myri.com/myrinet/overview/.

[Nes01] *Nessus homepage*, http://www.nessus.org/, 2001.

[NN02] S. Northcutt and J. Novak, *Network intrusion detection*, third ed., New Riders Publishing, August 2002.

[oSTIPM03] National Institute of Standards and Massachusetts Institute of Technology Lincoln Laboratory: Richard Lippmann Josh Haines Marc Zissman Technology ITL: Peter Mell, Vincent Hu, *An overview of issues in testing intrusion detection systems.*, http://csrc.nist.gov/publications/nistir/nistir-7007.pdf, June 2003.

[otCSotI90] Standard Coordinating Committee of the Computer Society of the IEEE, *IEEE Standard Glossary of Software Engineering Terminology*, 1990.

[pan05] *An experimental evaluation to determine if port scans are precursors to an attack*, Yokohama, Japan, june 2005.

[Pax98] V. Paxson, *Bro: A System for Detecting Network Intruders in Real-Time*, 7th Usenix Security Symposium, 1998.

[PN98] T.H. Ptacek and T.N. Newsham, *Insertion, Evasion and Denial of Service: Eluding Network Intrusion Detection*, Tech. report, Secure Networks, January 1998.

[Pro02] The Honeynet Project, *Know your enemy*, 2002.

[Roe] M. Roesch, *Snort inline mode*, http://www.snort.org/docs/snort_manual/node7.html.

[Roe99] M. Roesch, *Snort - Lightweight Intrusion Detection for Networks*, Proceedings of the Large Installation System Administration Conference (LISA) (Seattle, WA), November 1999.

[SBD+91] S.R. Snapp, J. Brentano, G. Dias, T. Goan, T. Heberlein, C. Ho, K. Levitt, B. Mukherjee, S. Smaha, T. Grance, D. Teal, and D. Mansur, *DIDS (Distributed Intrusion Detection System) – motivation, architecture, and an early prototype*, Proceedings of the 14th National Computer Security Conference (Washington, DC), October 1991.

[Ser05] Arturo Servin, *Security attacks and intrusion detection*, http://www-users.cs.york.ac.uk/~aservin/docs/attacks_ids.pdf, 2005, presentatin slides.

[SGVS99] R. Sekar, V. Guang, S. Verma, and T. Shanbhag, *A High-performance Network Intrusion Detection System*, Proceedings of the 6th ACM Conference on Computer and Communications Security, November 1999.

[SHM02] S. Staniford, J. Hoagland, and J. McAlerney, *Practical automated detection of stealthy portscans*, Journal of Computer Security **10** (2002), no. 1-2.

[sno04] *Snort - The Open Source Network Intrusion Detection System*, http://www.snort.org, 2004.

[SP05] R. Sommers and V. Paxson, *Exploiting Independent State For Network Intrusion Detection*, Proceedings of ACSAC, 2005.

[SPW02] Stuart Staniford, Vern Paxson, and Nicholas Weaver, *How to 0wn the internet in your spare time*, 2002, To Appear in the Proceedings of the 11th USENIX Security Symposium (Security '02).

[ST] Matteo Selmi and Enrico Tassi, *Intrusion detection system snort and tripwire.*

[Sto] Clifford Stoll, *The cuckoo's egg*, Pocket Books.

[SYB06] Avinash Sridharan, Tao Ye, and Supratik Bhattacharrya, *Connectionless port scan detection on the backbone*, Malware workshop, held in conjunction with IPCCC (Pheonix, AZ), April 2006.

[tcp02] *Tcpdump and Libpcap Documentation*, http://www.tcpdump.org/, June 2002.

[Top01] *Toplayer networks*, http://www.toplayer.com, November 2001.

[Tura] Aaron Turner, *tcpreplay homepage*, http://tcpreplay.sourceforge.net.

[Turb] Aaron Turner, *tcprewrite trac page*, http://tcpreplay.synfin.net/trac/wiki/tcprewrite.

[vH] Dimitri van Heesch, *Doxygen*, http://www.doxygen.org/.

[VM01] J. Viega and G. McGraw, *Building secure software: How to avoid security problems the right way*, Addison Wesley, October 2001.

[Wan07] Thomas Wang, *Integer hash function*, http://www.concentric.net/~Ttwang/tech/inthash.htm, 2007.

[WF07] Patrick Wheeler and Errin Fulp, *A taxonomy of parallel techniques for intrusion detection*, ACM-SE 45: Proceedings of the 45th annual southeast regional conference (New York, NY, USA), ACM Press, 2007, pp. 278–282.

[Wik07a] Wikipedia, *Network processor — wikipedia, the free encyclopedia*, 2007, [Online; accessed 17-April-2007].

[Wik07b] Wikipedia, *Receiver operating characteristic — wikipedia, the free encyclopedia*, 2007, [Online; accessed 14-April-2007].

[Wik07c] Wikipedia, *Vulnerability (computing) — wikipedia, the free encyclopedia*, 2007, [Online; accessed 8-April-2007].

[Wik07d] Wikipedia, *Zero day — wikipedia, the free encyclopedia*, 2007, [Online; accessed 11-April-2007].

[XCA+06] Konstantinos Xinidis, Ioannis Charitakis, Spiros Antonatos, Kostas G. Anagnostakis, and Evangelos P. Markatos, *An active splitter architecture for intrusion detection and prevention*, IEEE Trans. Dependable Secur. Comput. **3** (2006), no. 1, 31.

[ZU99] D. Zimmer and R. Unland, *On the Semantics of Complex Events in Active Database Management Systems*, Proceedings of the 15th International Conference on Data Engineering, IEEE Computer Society Press, 1999, pp. 392–399.

[1] http://tango.freedesktop.org/
[2] http://creativecommons.org/licenses/by-sa/2.5/