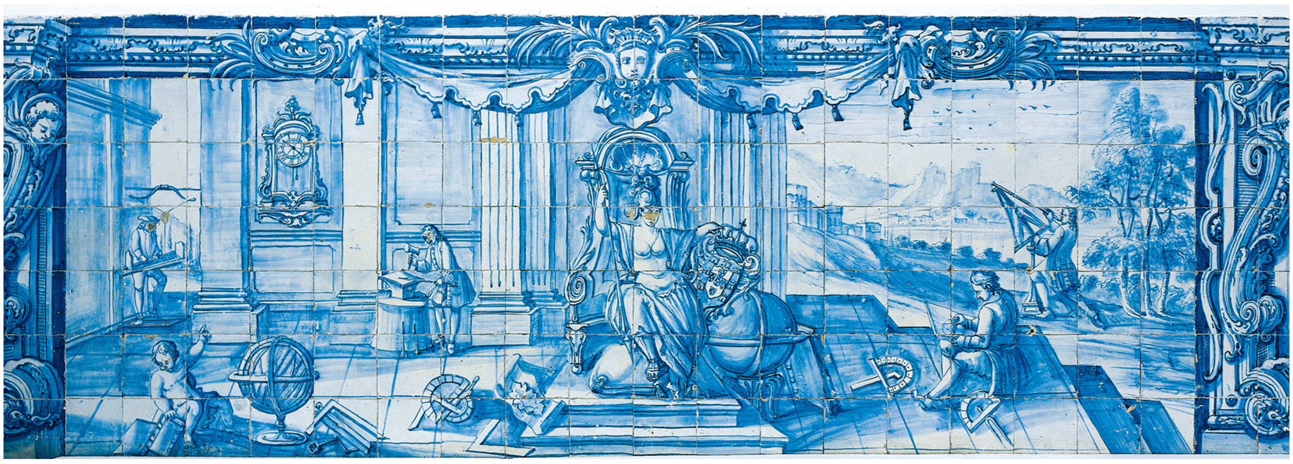
DECLARATIVE DOMAIN-SPECIFIC LANGUAGES AND APPLICATIONS TO NETWORK MONITORING

Pedro Dinis Loureiro Salgueiro

Tese apresentada à Universidade de Évora
para obtenção do Grau de Doutor em Informática

ORIENTADOR: *Salvador Pinto Abreu*

ÉVORA, Julho de 2012



INSTITUTO DE INVESTIGAÇÃO E FORMAÇÃO AVANÇADA

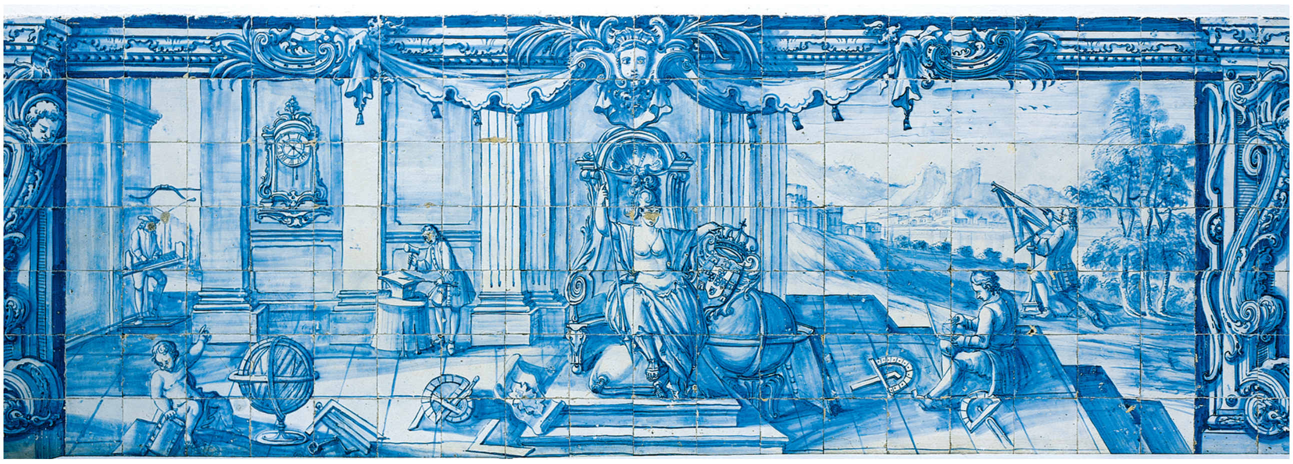
DECLARATIVE DOMAIN-SPECIFIC LANGUAGES AND APPLICATIONS TO NETWORK MONITORING

Pedro Dinis Loureiro Salgueiro

Tese apresentada à Universidade de Évora
para obtenção do Grau de Doutor em Informática

ORIENTADOR: *Salvador Pinto Abreu*

ÉVORA, Julho de 2012



INSTITUTO DE INVESTIGAÇÃO E FORMAÇÃO AVANÇADA

Dedicated to Célia and my Parents with love

Acknowledgments

First of all, I would like to thank my fiancee Célia and my parents for their support over all these years of work which helped me carrying out this work to the end.

I thank my supervisor, Prof. Salvador Abreu, which guided me through all this work, always pointing me in the right direction, providing me with new ideas and encouraging me to take new steps, which without him would be impossible to finish this work. I also thank him for reviewing all the papers produced during this work as well as this thesis, making this work understandable and readable.

I also want to thank Prof. Daniel Diaz of University of Paris 1-Sorbonne and Prof. Isabel Brito of Instituto Politécnico de Beja, which I had the privilege to work during this work and guided me in very important stages.

I also want to acknowledge Departamento de Informática of Universidade de Évora for providing me excellent conditions to work.

I would also like to acknowledge CENTRIA (Centro de Inteligência Artificial) for providing funds which allowed me to participate in some conferences.

I also would like to acknowledge IBM Corporation for donating the IBM QS21 dual Cell/BE blades used in this work, in the context of a SUR (Shared University Research) grant awarded to Universidade de Évora and CENTRIA.

I also would like to thank all my friends who helped me in many occasions.

Financial Support

I would like to acknowledge Fundação para a Ciência e a Tecnologia (FCT) for financially supporting me with scholarship SFRH/BD/35581/2007 by Programa Operacional Potencial Humano (POPH) of Quadro de Referência Estratégico Nacional (QREN), which allowed me to develop the work described in this thesis.

FCT Fundação para a Ciência e a Tecnologia
MINISTÉRIO DA EDUCAÇÃO E CIÊNCIA

POPH

QREN QUADRO DE REFERÊNCIA ESTRATÉGICO NACIONAL PORTUGAL 2007.2013

UNIÃO EUROPEIA
Fundo Social Europeu

Abstract

Network Intrusion Detection Systems (NIDSs) are in use probably ever since there are computer networks, with the purpose of monitoring network traffic looking for anomalies, undesired behaviors or a trace of known intrusions to keep both users, data, hosts and services safe, ensuring computer networks are a secure place to work.

In this work, we developed a Network Intrusion Detection System (NIDS) called NeMODe (NEtwork MOnitoring DEclarative approach), which provides a detection mechanism based on Constraint Programming (CP) together with a Domain Specific Language (DSL) crafted to model the specific intrusions using declarative methodologies, able to relate several network packets and look for intrusions which span several network packets.

The main contributions of the work described in this thesis are:

- A *declarative approach* to Network Intrusion Detection Systems, including detection mechanisms based on several Constraint Programming approaches, allowing the detection of network intrusions which span several network packets and spread over time.
- A Domain Specific Language (DSL) based on Constraint Programming methodologies, used to describe the network intrusions which we are interested in finding on the network traffic.
- A compiler for the DSL able to generate multiple detection mechanisms based on Gecode, Adaptive Search and MiniSat.

Linguagens Específicas de Domínio Declarativas aplicadas à Monitorização de Redes

Sumário

Os Sistemas de Detecção de Intrusões em Redes de Computadores são provavelmente usados desde que existem redes de computadores. Estes sistemas têm como objectivo monitorizarem o tráfego de rede, procurando anomalias, comportamentos indesejáveis ou vestígios de ataques conhecidos, por forma a manter utilizadores, dados, máquinas e serviços seguros, garantindo que as redes de computadores são locais de trabalho seguros.

Neste trabalho foi desenvolvido um Sistema de Detecção de Intrusões em Redes de Computadores, chamado NeMODe (NEtwork MOnitoring DEclarative approach), que fornece mecanismos de detecção baseados em Programação por Restrições, bem como uma Linguagem Específica de Domínio criada para modelar ataques específicos, usando para isso metodologias de programação declarativa, permitindo relacionar vários pacotes de rede e procurar intrusões que se propagam por vários pacotes e ao longo do tempo.

As principais contribuições do trabalho descrito nesta tese são:

- Uma abordagem declarativa aos Sistema de Detecção de Intrusões em Redes de Computadores, incluindo mecanismos de detecção baseados em Programação por Restrições, permitindo a detecção de ataques distribuídos ao longo de vários pacotes e num intervalo de tempo.
- Uma Linguagem Específica de Domínio baseada nos conceitos de Programação por Restrições, usada para descrever os ataques nos quais estamos interessados em detectar.
- Um compilador para a Linguagem Específica de Domínio fornecida pelo sistema NeMODe, capaz de gerar múltiplos *detectores* de ataques baseados em Gecode, Adaptive Search e MiniSat.

Contents

List of Figures

List of Tables

List of Acronyms

DSL	Domain Specific Language
GPL	General Purpose Language
IDS	Intrusion Detection System
NIDS	Network Intrusion Detection System
CP	Constraint Programming
CSP	Constraint Satisfaction Problem
CBLS	Constraint-Based Local Search
AS	Adaptive Search
GC	Gecode
SAT	Boolean Satisfiability Problem
CNF	Conjunctive Normal Form
CLP	Constraint Logic Programming
LAN	Local Area Network
WAN	Wide Area Network
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
ARP	Address Resolution Protocol
IP	Internet Protocol
FTP	File Transfer Protocol
FPGA	Field-programmable gate array

GPU	Graphical Processor Unit
GAC	Generalized Arc Consistency
DFS	Depth-first search
MCH	Min Conflicts Heuristic
4GL	Fourth-Generation Language
DSEL	Domain-specific embedded languages
APT	Abstract Parse Tree
SSH	Secure Shell
DNS	Domain Name System
DHCP	Dynamic Host Configuration Protocol
MITM	Man in The Middle
MAC	Media Access Control
SHA	Sender Hardware Address
SPA	Sender Protocol Adress
THA	Target Hardware Adress
TPA	Target Protocol Adress
IPV4	Internet Protocol version 4

Preface

This thesis represents the work of almost 4 years of research (2007-2011) in the context of my PhD program, which I decided to pursue mostly as a personal challenge and as an opportunity of self-development.

Over these years of work, many lessons were learned. Perhaps the most important thing to recall from this work, is that in research, nothing is guaranteed, the work is constantly evolving and changing according to the results, and the final goals of a research work can easily change and become completely different from the initial goals.

Chapter 1

Introduction and Motivation

This Chapter introduces the work presented in this thesis and briefly mentions the tools and techniques used therein: Intrusion Detection Systems (IDSs); Domain Specific Language (DSL) and Constraint Programming (CP).

1.1 Introduction

This thesis is about a declarative approach to Network Intrusion Detection, called NeMODe (NEtwork MOnitoring DEclarative approach), which relies on a declarative programming paradigm to both describe the network intrusions as well as to perform the detection of such intrusions.

The system provides a declarative description Domain Specific Language to model the network intrusions to be found by the system. It is primarily designed to ease the description of such intrusions in a very descriptive way. The language includes a compiler able to generate three different detection mechanisms, based on different Constraint Programming (CP) approaches.

The detection mechanism provided by NeMODe relies on Constraint Programming, providing three different intrusion detection mechanisms based on different approaches to Constraint Solving: Propagation, Constraint-Based Local Search (CBLS) and Boolean Satisfiability Problems (SAT), allowing for the detection of intrusion signatures which spread across several network packets over a period of time.

Part of the work presented in this thesis appeared before in joint publications with Prof. Salvador Abreu(my supervisor), Prof. Daniel Diaz and Prof. Isabel Brito. I thank all of them for letting me use the following common work: [1, 2, 3, 4, 5, 6, 7, 8].

1.2 Intrusion Detection Systems

IDSs are the first line of defense in present computer systems, essential to keep network users as well as data safe from bad intentioned people which take advantage of some service vulnerabilities to gain access to private data or perform some other kind of attack.

Over the years, most of the work related to IDSs focused on performance, trying to cope with increasingly higher speed of computer networks, leaving for second plan both the intrusion description and detection mechanisms. Existing IDSs often rely on complex pattern-matching algorithms capable of matching multiple patterns at once, looking for the desired patterns in both network packet headers and payload. Custom rule-based languages are commonly used to describe specific intrusions, stating what patterns should not be found in a network packet.

Such systems, are designed to detect network intrusions whose signatures can be found in a single network packet, and in most cases, are not able to detect network intrusions with signatures spread over time or several network packets.

Although some IDSs allow the specification and detection of intrusions which span several network packets, they usually need to resort to filters or plugins to do so.

1.3 Constraint Programming

Constraint Programming (CP) is one approach to declarative programming, by allowing the description of the problem in terms of "how" the problem is modeled, instead of "how" the problem is solved, widely used to solve combinatorial problems.

While using CP, a problem is modeled as a set of variables with a specific domain, over which a set of relations and restrictions are specified, according to the needs of the problem. Once the problem is solved, each variable is instantiated with a value from its domain, respecting all restrictions and relations which have been specified, thus reaching a valid solution.

Over the years, several approaches to CP have been developed and used. In this work we use Constraint Programming systems based on Constraint Propagation; CBLS and Boolean Satisfiability Problems (SAT).

1.4 Domain Specific Languages

Domain Specific Languages (DSLs) are small, very expressive, programming languages, specifically created for a specific application domain, contrary to General Purpose Languages which are designed to be as versatile as possible.

DSLs are designed with the purpose of facilitating the way the problems are modeled, enabling for non programmer users to write valid and efficient programs.

DSLs try to catch the *essence* of a specific application domain by capturing the pertinent semantics, abstractions and notions, providing a descriptive way of modeling problems, leading users proficient in the application domain to use such programming language in an easy and effective way.

There are two main types of DSLs; *internal* and *external* languages. Internal languages are DSLs which are build into a General Purpose Language (GPL), using libraries to extend the GPL to allow the use of the notions of the application domain, but limited to the constructs and capabilities of the base GPL. External languages are small programming languages built from scratch for a specific application domain, thus free to use the desired language constructs, using standard programming language development tools.

1.5 Using Constraints on Intrusion Detection

To maintain the quality and integrity of the services provided by a computer network, some aspects must be verified in order to maintain security.

The description of those conditions, together with a verification that they are met can be seen as an Intrusion Detection task. These conditions, specified in terms of properties of parts of the (observed) network traffic, will amount to a specification of a desired or an unwanted state of the network, such as that brought about by a system intrusion or another form of malicious access.

Those conditions can naturally be described using a declarative programming approach, such as Constraint Programming, enabling the description of these situations in a declarative and expressive way. Using Constraint Programming in intrusion detection allows to specify intrusion signatures as relations between several network entities, enabling an easy way to describe and perform the detection of attacks that span several network packets.

1.6 Thesis outline

Chapter 1 introduces the work presented in this thesis and presents a brief introduction on the concepts used in this work; Intrusion Detection System (IDS); Domain Specific Language (DSL) and Constraint Programming (CP). We also motivate for the use of Constraint Programming in Network Intrusion Detection.

We make a brief survey on the concepts used in this work, presenting some approaches to Intrusion Detection System and Constraint Programming in Chap(s). 2 and 3, respectively.

Chapter 4 describes how to model and perform Network Intrusion Detection using Constraint Programming methods. It presents the architecture of the system as well as the details of each network situation recognizer available in NeMODe.

In Chap. 5 we describe the Domain Specific Language provided by NeMODe which allows the description of specific network attack signatures. We provide the specification of the language and present some examples to demonstrate its use. We also include a brief survey of Domain Specific Languages.

Chapter 6 presents an evolution of NeMODe, allowing the use of an adaptive network traffic window, an important step towards live network traffic monitoring.

In Chap. 7 we present the experimental results of the work described in this thesis. We present the results for each test case analyzed while using all recognizers available in NeMODe. We also evaluate NeMODe and perform a comparison against other Intrusion Detection Systems.

Last, we conclude and present possible future work lines in Chap. 8.

1.7 Roadmap

In this section we provide a “roadmap” for reading this thesis which provides different reading paths:

Table 1.1: Reading paths

Subject	Chapter
Survey on Intrusion Detection Systems	1- 2
Survey on Constraint Programming	1- 3
Survey on Domain-Specific Languages	1- 5.1
Using Constraints for Intrusion Detection	1- 4- 5- 6- 7
A Domain-Specific Language for IDS	1- 5
All	1 to 8

Chapter 2

Network Intrusion Detection Systems

This Chapter introduces Network Intrusion Detection Systems. A brief history of IDSs is presented, allowing some insight into the most important evolutionary steps of these systems. We also present the most important types of IDSs, their characteristics as well as some important IDSs, both historic and presently used systems.

2.1 Introduction

Computer networks are getting more complex, larger, and faster, providing more services to their users to satisfy the demanding needs of users, which increasingly depend on networks.

Due to the increasing use of computer networks, there is the need to ensure that both users and data are safe while working in systems connected to a computer network.

There are a number of tools and approaches to ensure the safety of computer networks. Firewalls are the first line of defense against network attacks, filtering network traffic and performing access control, but they can are not sufficient for a large number of intrusions.

Network Intrusion Detection Systems (NIDSs) are at another level of defense, being one of the most important tools to prevent network intrusions, attacks or other type of malicious actions which could compromise the safety of users or data. NIDS focus on traffic monitoring, trying to inspect network traffic, looking for anomalies or undesirable communications.

Intrusion Detection Systems (IDSs) have been the subject of study for years, with the purpose of preventing and detecting intrusions to ensure the safety of computer networks. Most of the academic work related to IDS consists in the development of faster detection methods [9], still, over the years a number of different approaches to IDSs have been used, with focus on new methods both to describe and detect network attacks.

2.1.1 Approaches to Intrusion Detection Systems

A number of approaches to Network Intrusion Detection Systems have been introduced and evolved over time. Although several approaches can be used, the first academic works related to IDSs classifies them in two major categories [10, 11]:

1. signature based
2. anomaly based

Intrusion Detection based on Signatures

Signatures are the central aspect of signature based IDSs, also known as *misuse detection* [11], since the desired network attacks to be monitored are described through a representation of specific properties which identifies the desired network attack.

Each network intrusion present specific characteristics, leaving a trail which identifies and makes proof of the attack. These characteristics can be either specific data in the payload of network packets, specific data in the headers of network packets, or relations between several network packets with special characteristics. These characteristics which identify the network attacks, are called *intrusion signatures*.

Intrusion signatures are used to represent what is considered to be a *licit* or *illicit* behavior in the network.

The signatures are then matched against the network traffic in order to perform the detection of the designated network situations, a method analog to the detection of virus in computer systems.

IDSs based on signatures present some disadvantages, the most important being that they are not able to detect unknown intrusions: those which have no signatures in the IDS database [10].

With this approach to IDS, signatures which model attacks are explicitly *programed*, usually in terms of rules, containing the description of what should be found in the network traffic to recognize the specific signature, thus triggering the corresponding alarm. This approach to Network Intrusion Detection is very similar to a *default permit security policy* [12].

Several approaches have been used in signature based IDSs, either in the representation of the network attack signatures or in the detection mechanism itself.

Follows the most widely used approaches [10]:

State modeling The underlying idea of *state modeling* is to describe the desired network intrusions as a set of states. Such states are then looked for in the network traffic, which, when found, identify the specific intrusion. Several types of *state modeling* exist, *State transition* and *Petri nets* are some, just to mention some.

Expert systems IDSs based on expert systems rely on a set of rules describing the desired attacks or the behavior of the attacker. The expert system is then responsible to reason about the security of a given network. This method is very powerful, but IDSs performance is low.

String matching Intrusion Detection Systems based on *string matching* use regular string matching techniques and algorithms to perform sub-string matching in data found on the network traffic, which identify specific network signatures. The signatures are string patterns which identify the desired attack, which should be found in the network traffic. Usually complex and efficient multi-pattern matching algorithms are used, making this approach very efficient, but not very flexible.

Anomaly Based IDSs

Anomaly or behavior analysis is another widely used type of Intrusion Detection Systems, approaching the problem from another point of view. Instead of looking for specific or known network intrusions, it searches for anomalies in the network traffic.

On Anomaly based IDSs, the *normal* behavior of the network is usually modeled through the use of statistical methods and data mining approaches. According to the normal network behavior model, the current network behavior is analyzed, and if it deviates from the model more than a predetermined level, then there is a likelihood that the network is under attack.

This type of IDSs is very flexible but presents some problems, the most significant is the incapacity to detect network attacks which don't change the behavior of the network in a sufficient scale to trigger the alarms. One other issue, is the large number of false positives, due to a more "strict" network behavior discipline which may be used to identify as much "abnormal" behavior as possible [10].

Anomaly based IDSs need to acquire the network behavior and transform it into a model, which can be compared with the current network behavior. Several methods exist to acquire such behavior, but most of them fall in 3 main classes [10]:

Self-Learning Anomaly based IDSs which follows a Self-Learning approach observe and analyze the network traffic over a period of time, learning from example the normal behavior of the network, simultaneously building a model which represents the normal behavior of the network.

Programmed The *programmed* anomaly based IDSs rely on someone who is capable of *teaching* the system, by *programming* the desired anomalous events. In this approach, the system administrator is responsible for deciding which events should be considered anomalous and understood as a network attack.

Default deny Anomaly based Intrusion Detection Systems which fit in the *default deny* class resemble in many aspects the *default deny security policy*, which formulates the events which are *allowable*. Every other event is forbidden or considered illegal. In this class of IDSs, the circumstances which identify the *allowable* events are explicitly stated, everything else is considered as an intrusion.

2.2 History of Intrusion Detection Systems

Before the introduction of Intrusion Detection Systems, the conventional security methods which were used to secure hosts relied in “closing” the hosts to secure them from the “outer” world, enclosing them in a protective “shield”.

These security methods relied on access control, using *Identification & Authentication* mechanism requiring users to identify and authenticate themselves to gain access to the system [13].

As an alternative to those security methods to improve security, the Intrusion Detection concept was introduced around the mid-’80s [14], upon which many Intrusion Detection Systems were created.

2.2.1 Host based IDSs

These first IDSs were only able to monitor a single host, monitoring audit trails produced by the host operating system, looking for network attacks or other type of undesired access, rather than directly monitoring the network traffic.

IDSs which are not able to monitor network traffic, but are able to monitor more than one host at the same time, are also considered host based IDSs.

Systems such as AT&T’s ComputerWatch [15], the HAYSTACK system [16], IDES [17], the ISOA system [18, 19], MIDAS [20], just to mention some, were some of the first IDS systems, performing host intrusion detection within a single host.

AT&T's ComputerWatch

AT&T's ComputerWatch [15] is a Host Intrusion Detection System, with no real-time capabilities, having as primary purpose to assist the system administrators responsible for the hosts security. Besides assisting the system administrators, it also has some intrusion detection capabilities, although in very limited way.

ComputerWatch manages to assist system administrators by summarizing the large audit trails produced by the hosts without removing relevant information about undesired attacks, thus greatly reducing the amount of information to be analyzed by the system's administrators, indicating which events should be analyzed in more detail. These summarized audit trails can be interactive, analyzed by the system administrator as they are being produced, or produced in a *batch* for later review.

ComputerWatch was designed for the UNIX System V/MLS, and was based on expert systems to both summarize the audit trails, enhancing the security sensitive information and perform intrusion detection, with rules for detecting anomalies or simple intrusions.

The HAYSTACK system

The HAYSTACK system [16] is an Intrusion Detection System based on host audit trails, representing the system user's behavior and was designed for the Unisys (Sperry) 1100/60 mainframe, running the OS/1100 operating system. It summarizes the hosts audit trails, transforming them into much smaller traces, containing only relevant information about user behavior, anomalous events and security incidents, thus allowing the detection of intrusions.

The system was able to detect some types of attacks or security issues; such as break-in attempts, masquerade attacks, penetration of security system or information leakage. To achieve this, HAYSTACK uses *behavioral constraints* representing the desired behavior as well as the security policies, imposed by system administrators.

HAYSTACK provides three different ways to help system administrators in detecting a possible intrusion or attack:

Notable Events Events which might modify the security policy and state, delivered to the system administrator, which then decides whether there is some kind of intrusion or malicious access.

Special Monitoring Special events are marked by the system administrator for monitoring, such as when a particular *user id* is used or when a file is accessed.

Statistical Analysis HAYSTACK also employs statistical analysis to perform intrusion detection; setting "suspicious quotients" which measures how the user behavior resembles a specific intrusion; and looking for a significant deviation on the user behavior when compared to user's recent behaviors.

The IDES system

Intrusion-Detection Expert System(IDES) [17] is a system-independent, real-time Intrusion Detection System, running independently in a central system which processes data audit trails collected from monitored hosts.

IDES relies on expert systems and statistical methods. A rule-based, forward-chaining expert system is used to model known intrusions, systems vulnerabilities and other specific security problems, while statistic methods are used to detect anomalous user behavior.

Both statistical methods and expert system use the same input to produce the reports. Such files are statistical representations of the host audit files.

User behavior is observed, adaptively learning the behavior of the users using statistical methods, and keeping a historic record of the users behaviors. The learned *normal* and historic behaviors are then compared against current user behavior, and if it deviates too much from what is expected, the behavior is considered as an intrusion. Also, if some of the expert system rules which models undesired behaviors, known intrusions or known vulnerabilities are triggered, the behavior is also considered an intrusion and reported.

The ISOA system

ISOA [18, 19] is an Intrusion Detection System designed for Unix workstations, allowing system administrators to perform automated and interactive audit trails, helping with the detection of intrusions by presenting graphical alerts, advices and explaining the possible threats.

It is based on the notion of *indicators* which are the records of the audit trails. The *current indicators*, representing current events found in audit trail are correlated with *expected indicators*, representing the expected events of both users and hosts. This correlation is achieved with an expert system and statistical methods.

The *indicators* are organized as a hierarchy of security level concerns, so that when the *indicators* are found in the audit trails, the security level concerns rises. When the security level concern reaches some predetermined level, the system makes a more detailed analysis of the given user or host.

Profiles representing the expected user and hosts behaviors which include historical events, are checked against the current events through the use of statistical methods to identify the deviation between expected and verified behaviors, according to predetermined threshold defined in such profiles.

ISOA provides two types of Intrusion Detection, one in real time, performed at the same time the audit data is being produced, triggering further investigations if certain events are found and comparing the current parameters or behaviors with the ones that should be found. A second type of intrusion detection is performed after the user terminates the session, by checking session statistics against the predetermined profiles.

The MIDAS system

The MIDAS system [20], which stands for Multics Intrusion Detection and Alerting System, is a real time Intrusion Detection System based on expert systems and statistical analysis, which was used in the Multics operating system.

This system is used to perform a characterization of the *normal* behavior of both users and systems using audit files. Both system and users are then monitored for a significant deviation of their behavior.

The behaviors and activities of both user and system are kept over time, represented as statistical profiles. This allows MIDAS to compare the current behavior and activities with the historic of both users and system, deciding if there is a significant deviation, indicating the possibility of an attack.

The expert system used in MIDAS is organized in a series of chained hierarchical rules. These rules can trigger more specific rules of a higher hierarchical level, denoting an increase of the probability of an attack. The expert system rules used to model the desired network intrusions are organized in 3 types:

Immediate Attack Rules Rules with no statistical information, identifying individual events which are known to be anomalous.

User Anomaly Rules Statistical profiles are used by such rules to determine if there is a significant deviation from previously, historic, observed user behaviors.

System State Rules Rules are similar to the *User Anomaly Rules*, regarding the entire system instead of users.

USTAT

USTAT, presented in [21], is a real-time Intrusion Detection System based on State Transition Analysis for UNIX systems, based on STAT [22], a previous State Transition Analysis Tool, which, when introduced, was a new way for modeling intrusion, and actually used in the development of real-time intrusion detection tools.

USTAT was developed for SunOS 4, and relied in audit trails produced by the C2 Basic Security Module of SunOS as source to detect the desired attacks. USTAT has the particularity of only keeping track of critical actions that must occur for a specific intrusion to be successful.

In USTAT, the attacks were described by a set of goals and transitions based on state-transition diagrams, and, if a specified event is triggered, an attack state is considered an intrusion.

2.2.2 Network based IDSs

Host based IDSs were limited to monitoring one host at a time and weren't able to analyze network traffic. A second generation of IDSs were introduced, capable of monitoring several hosts connected through a network. These systems became known as Network Intrusion Detection Systems.

Systems such as IDES and ISOA, described in Sect. 2.2.1, rely on audit trails generated by the target host, and thus unable to monitor network traffic, they were also considered NIDSs, since they are capable of monitoring several hosts connected through a network, by transferring the audit trails to a centralized location. Such systems can be considered as either host or network based IDSs.

Other systems apply different approaches to Network Intrusion Detection, using either network traffic monitoring only, or combining network traffic monitoring with audit trails. Systems as such as NADIR [23], NSM [24], DIDS [25], are representative IDSs with network traffic monitoring capabilities. We now detail these further.

The NADIR system

NADIR [23] is an IDS based on an automated expert system with built-in rules which describe the *illegal* events, producing reports which help system administrators to review audit files while looking for *anomalous* events.

NADIR was implemented as the Integrated Computing Network(ICN) of Los Alamos National Laboratory(LANL), which was comprised of hundreds of users using hundreds of hosts, connected to different parts of the network operating at different security levels. Such network sections are connected by three main nodes which provide user authentication and access control, store data, authenticate and record all file access. Each of these nodes produce audit records, which are then sent over to NADIR.

Based on these records, NADIR generates weekly reports for both user and network activity, which are then compared against built-in expert rules modeling the security policy as well as *illegal* or suspicious behaviors. Such rules are developed by security experts. NADIR assigns a *level of interest* for each rule that has been triggered, presenting the results to the system administrators in either a detailed *raw* format, or in a graphical format which highlights the most suspicious events. It also provides tools for further investigations of the events that deserve more attention.

The NSM system

The Network Security Monitor(NSM) [24] is a significant departure from other Intrusion Detection Systems of its time as the detection mechanism relies entirely on network traffic monitoring instead of audit trails analysis.

This type of intrusion detection presents a variety of advantages over systems which relies on audit trails analysis [13]. These are the most significant:

1. Systems based on audit trails tend to work only with one Operating System at the same time, since audit trails are Operating System dependent. Network traffic monitoring allows to monitor network protocols used my most Operating Systems, thus, it allows to monitor hosts using different Operating Systems at the same time.

2. Audit trails usually are not readily available, the Operating System can delay its writing or might need to be transferred to be analyzed. Using network traffic monitoring, the system has immediate access to the data, as soon as it gets on the network.

3. Audit trails are vulnerable, as they can be tampered with, or the administrator might simply turn them off. The use of network monitoring avoids this problems, since the user has no control over the system.

NSM models the network in a hierarchically structured Interconnected Environment Model(ICEM), which is composed by 6 interconnected layers, from the lower level bit stream layer, up to the higher level layer which models state of the entire network.

These layers are all interconnected in order to relate all possible information that can be acquired from the network traffic, relating, network packets, unidirectional data streams, bidirectional data streams, network activities of single hosts, how the hosts are connected and the behavior of the entire system.

Expert systems are used by NSM to analyze network traffic, using as input a number of network related information, such as the network model(ICEM), profiles with expected traffic behavior consisting of data paths and service profiles, knowledge about each service available on the network, the authentication level required by each service, the level of security of each host and signatures of past attacks.

DIDS

The Distributed Intrusion Detection System(DIDS) [25] was the first IDS combining distributed network monitoring and audit trail reduction with a centralized data analysis.

DIDS was designed having as its main priority to overcome the deficiencies presented by NSM, such as not being able to monitor attacks which access the network through a dial-up line, and extends the monitoring from a Local Area Network (LAN) to the Wide Area Network (WAN) level.

In a network monitored by DIDS, a monitor is installed on each host connected to the network, and each network segment has an individual monitor, thus augmenting the network data with data captured in each host.

Both host and network monitors are responsible for collecting evidence of any suspicious activity, reporting “interesting” events, which could be related to network attacks to a central “director”. Based on expert systems, the “director” analyzes the reports, evaluating the evidences collected by the monitors, and decides if there is any kind of threat.

DIDS relies on aggregate information to perform intrusion detection, i.e.: even if the activity on a single host doesn’t seem suspicious, when aggregated with data from other hosts, it may reveal a questionable event.

NetSTAT

NetSTAT, described in [26], is much like USTAT [21], previously described in Sect. 2.2.1, also following the approach of STAT [22], using state transition analysis to approach intrusion detection.

The primary goal of NetSTAT was to improve USTAT, extending the state transition analysis to network based-intrusions, allowing to use the USTAT IDS approach no monitor network traffic.

NetSTAT follows an approach based on formal models, where the network attacks are modeled as state transition diagrams, and each state and transition are specific to the network environment, modeled through the use of hypergraphs.

NetSTAT is a distributed application, capable of operating on complex networks composed of several sub-networks. It is composed by: a network fact base, which stores and manages the relevant information about the network security; the state transition scenario database, which manages the state transition representation used to model the desired intrusions; the probes which monitor the network traffic at specific network locations; and the analyzer, used by the network administrator to configure all the system.

2.3 *Modern* IDSs

Presently, Snort [27, 28] is one of the most widely used Network Intrusion Detection Systems, providing a rule-based description of the network attacks and relying on multi-pattern matching algorithms to perform intrusion detection.

Bro [29] is another modern and frequently used Network Intrusion Detection System, although not as much as Snort. Bro follows an approach slightly different from Snort, combining two of the most used approaches in Network Intrusion Detection Systems: misuse and anomaly detection, bringing together the best of the two approaches.

Both Snort and Bro are rule-based systems, since the known network attacks or network state are described through the use of rules.

2.3.1 Snort

Snort [27, 28] is a widely used, open-source, lightweight and cross-platform IDS, based on the `libpcap` [30] sniffer and logger, providing real time alerting capabilities to inform the network administrators of a possible intrusion.

Snort relies on efficient pattern-matching techniques to detect the desired intrusion signature, being designed to monitor small TCP networks, where it is not feasible to use large and expensive commercial IDS. In particular, Snort uses a multi-pattern matching algorithm, an optimized version of the Aho-Corasick algorithm [31], which allows the efficient detection of multiple network attack signatures at once, without sacrificing the performance of the system.

To describe the signatures of the desired network intrusions or attacks, Snort provides a simple rule based language which relies on specifying tests and actions over single network packets. Snort *ships* with a very complete set of rules which covers most of the known network intrusions or attacks, making it very easy to start using Snort.

Snort presents some pre-processors that help relate separate network packets; `Stream4` is such a pre-processor: it gives Snort the ability to be stateful, allowing the trace of network packets on its session and use its state on the given session to create a rule that describes the desired signature. The `Flow` pre-processor also allows snort rules to relate with other rules by using the *flowbits* keyword, allowing one rule to set some flag, and another other rule can check if that flag is set, and, if so, complete the rule to describe the desired signature.

These two pre-processors help Snort to describe network attack signatures that span several network packets, but they do so in a very limited way, not allowing the description of more complex relations between packets, such as the temporal distance between two packets. Also, the way that the relation between several rules is expressed is awkward, counter-intuitive and error-prone.

2.3.2 Bro

Bro [29] is another a widely used, open source IDS, with a clear separation between the mechanism and policy and with a high degree of extensibility.

Bro is a peculiar IDS, as it is neither fundamentally anomaly based nor signature based: it supports both approaches to Intrusion Detection. Bro is event driven, abstracting the network packets. An event engine performs the low-level processing, which then passes the events to a higher level policy layer, where the network administrator defines the specific network policies using a specific scripting language. It also provides a signature matcher, allowing the use of Snort rules by means of a specific signature converter.

Bro's signatures are described in a specially built and flexible signature language, script based, having in mind the network intrusion detection domain, providing specific data types for intrusion detection, such as Internet Protocol (IP) addresses and port numbers.

This signature language allows one to describe network attack signatures mostly as a pattern of bytes, which should be found in the network packets payload to detect the specified signature. The use of such network signatures is a straightforward process for some attacks, but in other cases, it's harder to specify signatures of this type, leading to a much broader specification, which could result in a high number of false positives.

To prevent this scenario, Bro makes uses of *context* [32], providing two levels of context, one at the low level, using regular expressions instead of fixed patterns, and another, higher level, which takes advantage of semantic information which is brought by both protocol analysis and the scripting language.

Bro provides application layer analyzers for some protocols/applications; such as Finger, File Transfer Protocol (FTP), Portmapper, Ident, Telnet and Rlogin [33]. Besides these protocols, it can easily be extended to provide other application specific processing, allowing the use of specific properties of these applications either in the policy or the signature specification. Bro also provides analyzers for specific network situations, such as Scan Detection, SYN-Floods, and Stepping Stones.

Bro manages to reconstruct TCP sessions as well as track the connection state, thus having the capability of relate several network packets, although in a limited way. It is also able to perform time and event based correlation [34], thus allowing for a finer-grained relationship between packets, although this is made at the event level, being a complex and laborious task.

2.4 The quest for performance

Most of the work related to Intrusion Detection Systems tend to focus on performance improvement in terms of detection speed, allowing to perform intrusion detection in present computer networks, with increasingly higher speeds, reaching several Gbit/s.

To cope with such network speeds and still be able to perform Intrusion Detection in a live network link, several approaches have been taken. Attig and Lockwood [35] implemented a framework capable of scanning network packet headers and payload content in dedicated hardware, using Field-programmable gate arrays (FPGAs), allowing the intrusion detection in fast networks.

Gnort [36] is another approach to the increasingly higher speeds of today's computer networks. In this case, Graphical Processor Units (GPUs) are used to improve the pattern-matching operations of Snort, allowing it to deal with faster network speeds.

The quest for performance in Network Intrusion Detection Systems to monitor fast networks has also been addressed by using different pattern-matching algorithms or, in some cases, combining several algorithms to obtain a faster result. The work presented by Coit et al. [37] implements a variation of the multi pattern-matching algorithm Aho-Corasick in Snort, combining it with the Boyer-Moore algorithm. The new algorithm allows Snort to monitor live network traffic at much higher speeds than it would otherwise.

We now proceed with a more detail description of these systems.

2.4.1 Using FPGAs in IDS

Attig and Lockwood [35] proposed a framework with the capability of scanning network packet headers and payload content using FPGAs.

In particular, this work is capable of processing Snort rules in hardware. This framework allows future modules to be added so that all necessary Snort features are supported. These capabilities allows one to use all Snort features to perform rule-based processing at multi-gigabit speeds.

This work combines rule processing together with content scanning, defining the rule processing. This combination is achieved through a rule processing unit, operating in real time, determining if there is a match against any rule, which receives as input both packet header and content match information.

This system forwards the traffic through several modules, first through a *TCP Flow Assembler*, which re-assembles and orders the TCP packets. These packets are then passed to the *Header Processors*, verifying the header of each packet; then to the *Content Scanners*, verifying the packets payload.

If there is a matching header or signature, both packet header and content are forwarded to the *Rule Processor*, which checks if some rule is matched against the network packets. Here, a rule consists of an action, an header, and 0 to n signatures. A rule match happens when the header and all associated signatures are detected.

To model the system, the rules are programmed dynamically using special control packets from an administration console on the network.

2.4.2 Gnort

Gnort [36] explores the capabilities of modern GPUs, to perform Intrusion Detection, more specifically by offloading the pattern-matching operations from the CPU onto the graphic card, which are becoming increasingly powerful.

Gnort is based in the Snort open-source IDS, adapted so that its pattern matching operations are transferred to the Graphics Processing Units(GPUs), thus increasing the throughput of the system.

The system uses the multi-pattern matching Aho-Corasick algorithm, also used in Snort, ported to run on graphic cards, boosting the performance up to a factor of three, when compared to Snort running in a computer equipped with a 3.40 GHz Intel Pentium 4 and 2GB of RAM.

Gnort comprises three main tasks: first the network packets are transferred to the

GPUs into batches; then the GPUs perform the pattern matching through the use of several threads; and finally, the results of the pattern matching techniques are transferred back to Snort.

Gnort was implemented to run in NVIDIA GeForce 8 series(G80) video cards, more specifically a NVIDIA GeForce 8600GT using the Compute Unified Device Architecture (CUDA) SDK.

2.4.3 Speeding up Snort

Coit et al. [37] implemented a variation of the Aho-Corasick algorithm, used in Snort to perform pattern matching, by combining the Aho-Corasick and the Boyer-Moore algorithms in order to improve the overall performance of Snort.

More specifically, this is a Boyer-Moore algorithm applied to a set of keywords held in an Aho-Corasick-like keyword tree, which overlays common prefixes of the keyword.

This algorithm, which the authors call AC_BM, can improve the performance of Snort, regarding the speed of network traffic analysis, from 1.02 up to 3.22 times, depending on the network traffic type and rule set used. Such performance gains have the cost of an increase in memory use, 3 times more that in the Snort original pattern matching algorithm.

2.5 *Other* approaches

One limitation which is found in most Intrusion Detection Systems, more evident in widely used systems, is the lack of a method to describe intrusion signatures which span a long period of time, or which spread across several network packets.

Still, there are some systems which allow the specification of such signatures, but in a very limited way and with a very counter-intuitive description, usually achieved by means of pre-processors or built-in modules, purposely built for some tasks.

These approaches are very distant from the conceptual level of the application, with focus in a single problem, using a very specific terminology, instead of focusing in the overall abstract problem of Network Intrusion Detection.

Despite the focus of the IDS community on performance, some works have been developed which draw on other aspects, some using declarative approaches, either to model the desired intrusions or even for the detection mechanism.

We now present some of these: a declarative stateful intrusion detection based on temporal logic [38]; Sutekh [39], a rule-based system using logical operations to model the intrusions; IDIOT [40, 41, 42], a pattern-matching IDS based on Colored Petri-Nets; and LAMBDA [43], a specially built language to describe attacks, through the use of logical formulas.

2.5.1 Declarative Stateful Intrusion Detection

A declarative approach to stateful Intrusion Detection based on temporal logic is presented by Couture et al. [38]. It keeps track of context, thus allowing to model network intrusion signatures involving several network packets, a feature which lacks in most other network intrusion detection systems.

This work is based in a propositional logic extension, which allows the specification of temporary properties, properties between a given set of events which can be understood as knowledge gathered from the events.

Intrusions are modeled as a sequence of events, where events are packets with specific properties considered relevant for the given attack.

This system is capable of expressing timing, safety, and repetition properties, very relevant features to intrusion detection, thus allowing the expression of signatures which span several network packets by allowing a relation between them to be stated.

To express intrusion signatures, the authors use the syntax of propositional logic, extended with a temporal operator used to reach past network packet in the network traffic. Using this syntax, the expression of the desired intrusion signature is very compact, but yet remains challenging for network administrators.

2.5.2 Sutekh

In [39], Pouzol and Ducasé propose a declarative signature specification language named Sutekh, with precisely described semantics, allowing to reproduce rules for existing rule-based IDSs, using an algorithm based on the construction of a state-transition diagram.

Sutekh is a declarative signature specification language providing sequence, alternative, partial order, negation, event correlation via logical variables unification, condition verification and alert triggering, using a declarative semantics to describe the sequence of events in the network traffic corresponding to a network signature.

Sutekh does not provide an intrusion detection mechanism, instead, it produces rules for existing rule-based Intrusion Detection Systems from the rules specified in the Sutekh language. To do so, the authors defined an algorithm to build a state-transition diagram, called *SigGraph*, representing the evolution of the detection process. Based on this diagram, rules for different existent IDSs can be generated, such as for ASAX [44] or EMERALD [45].

2.5.3 IDIOT - Intrusion Detection with Colored Petri Nets

In [40, 41], Kumar and Spafford present an approach and an implementation of a model to perform Intrusion Detection using pattern matching based on graphs, providing a clean separation of the various components.

The system can be viewed as three basic abstractions: the *Information Trail*, which encapsulates the audit trail; the *Signature Layer*, providing a system independent signature representation; and the *Matching Engine*, which encapsulates the pattern matching detection technique. The evolution of this work was latter named IDIOT [42].

In this case, the intrusion signatures are represented as specialized graphs, more specifically, using an adaptation of Colored Petri Nets, having guards which are the context where the signatures are matched, and vertices being the system states. The intrusion signatures represent the sequence of events and the context of the desired network signature.

The signature representation is achieved in a straightforward syntax which directly maps the signature in the graph.

2.5.4 LAMBDA

LAMBDA [43] is a language to describe attacks in general, not only, but also including network attacks or intrusions. Attacks in LAMBDA are described by means of conditions and effects through the use of logical formulas related to the computer systems state, providing a description of the desired attack from the attackers point of view.

This language allows the description of generic attack operations, independently of the detection mechanism and the operating system used. This generic description is then complemented with detection mechanism elements.

The attacks are described as a combination of actions and a set of statements related to the target system. More specifically, the main components of the attack description are: a set of *conditions* which must be verified for the desired attack to be succeeded;

the *effects* of a successful attack, which are the consequences of one attack to the system; and a *scenario*, describing how the attacker combines the actions necessary to achieve the desired network attack. This description of the attacks is very powerful, allowing the description to be from the attacker's point of view. Besides these main components, the description of an attack also includes the actions to take when an attack is detected.

One of the main concerns with LAMBDA was to use a declarative approach in the definition of the language. Another concern of LAMBDA was to be modular, allowing the use of rules previously defined in other network attacks, to describe new ones.

2.6 Conclusion

In this chapter we introduced the concept of Intrusion Detection, covering the most relevant types of IDSs, types of detection and characteristics of some IDSs, and gave a brief account of the history of IDSs.

We saw several types of Intrusion Detection Systems, using different approaches to describe the desired network signatures or network status as well as for the detection mechanisms. However, most of these systems fall in two major types of types of approaches: the signature based and the behavior based.

Although there are many approaches to IDSs, the most widely used systems are often rule-based, where the intrusions or attacks are described though the use of rules, relying on pattern matching techniques as a detection mechanism, usually resorting to complex multi-pattern matching algorithms, allowing the simultaneously detection of multiple signatures without sacrificing performance.

Another conclusion which can be drawn from this chapter is that, over the years, the studies in the area of IDSs have focused in enhancing the current pattern-matching algorithms to be able to cope with the increasingly network speeds, either by combining several pattern matching algorithms or by implementing pattern matching algorithms in alternative hardware. Still, there is work in either the description or detection mechanisms underlying the IDS, but in much smaller numbers.

Although there are effectively some Network Intrusion Detection Systems which provide native mechanisms, designed from scratch to allow the description of intrusions which spread across several network packets, most IDSs are either capable of using signatures which only involve one network packet, or, when allowing to specify a relation between several network packets, this relation is achieved in a very basic and limited way, often using ad-hoc tools, specifically made to achieve those relations.

Chapter 3

Constraint Programming

In this Chapter we introduce the Constraint Programming (CP) paradigm, an approach to Declarative Programming. We present several instances of CP, including Propagation Based Solvers, Constraint-Based Local Search and Boolean Satisfiability Solvers, as well as the constraint solvers used in our work.

3.1 Introduction

Constraint Programming (CP) [46, 47, 48] is an approach to declarative programming paradigm, used to solve large and complex real world problems, mostly, but not only, of combinatorial search nature. This approach to declarative programming is widely used in the areas of planning and scheduling, but also used in a variety of areas.

Constraint Programming consists in the formulation of a solution to a problem as a CSP [46], in which a number of variables are introduced, with well-specified domains, describing the desired state of the system. A set of relations, called *constraints*, is then imposed on the variables which make up the problem. These constraints are understood to have to hold true, resulting in a *solution* to the CSP.

CP relies on a declarative way to model problems, in a way that we describe the problem instead of specifying how the problem should be solved. Such description is achieved by specifying a set of relations and properties that must be verified, a key aspect characteristic of Constraint Programming.

The word "constraint" is central to Constraint Programming, since the major part of modeling a CSP is achieved by stating constraints over a set of variables. A constraint is simply a way to specify logical relations between several entities, variables, each taking values from specific domains.

3.1.1 History of Constraint Programming

Constraint Programming originated back in the sixties and seventies in Artificial Intelligence area. One of the first problems modeled as a Constraint Satisfaction Problem problem was probably the *scene labeling* [49], with the goal of identifying 3D scenes using lines from 2D drawings. Another first application of constraints was Sketchpad *interactive graphics* [50], an interactive graphics application, which allows the user to draw and manipulate constrained geometric figures, contributing in a large scale for the development of local propagation methods and constraint compiling. Other early important contributions to CP were the systems Alice [51], CONSTRAINTS [52] and ThingLab [53]; systems which back on those days were able to provide the most important features of CP [54], including:

- Declarative problem modeling
- Propagation of effects
- Efficient search

Although Constraint Programming could already be found in such systems, the most important step towards CP was taken when Gallaire [55] and Jaffar and Lassez [56] reached the conclusion that logic programming was a particular type of Constraint Programming, since the central idea of Logic Programming and Declarative Programming is that problems should be modeled not by specifying how they should be solved, but rather describing the problem itself, the same underlying ideology of CP. This conclusion led to combining Constraint Programming and Logic Programming, originating the concept of Constraint Logic Programming (CLP), one of the widely used approaches to Constraint Programming. CHIP(Constraint Handling in PROLOG) [57] and GNU Prolog [58] are examples of CLP systems.

Although Constraint Logic Programming was one of the first successful implementations of Constraint Programming, it is also possible to use CP in general purpose programming languages or special declarative programming languages. Ilog Solver [59] was an example of such a system, a library for Constraint Programming in C++.

3.1.2 Discrete and Continuous CSPs

There are two major types of Constraint Satisfaction Problem: discrete and continuous [46]. Discrete CSPs are commonly used to solve problems of a combinatorial nature. Continuous Constraint Satisfaction Problems or CCSPs, are also widely used, but in a different application, mostly for solving non-linear systems and optimization problems.

The primary differences of these types of constraints is the domain of the variables of the CSP: in discrete CSPs, variable have a finite and discrete domain; while in the continuous CSPs, also known as interval CSPs, variables have a continuous domain, a continuous range of values, usually an interval of real numbers.

Another major difference between these two types of Constraint Satisfaction Problems is the techniques used to solve the problems. Discrete constraints rely mostly on graph theory and integer programming, while continuous constraints are fundamentally based in numerical analysis, using numerical or symbolic algorithms and interval-based techniques.

Although these two types of CSPs use different techniques to solve problems, they share some aspects, since propagation algorithms are commonly used in both approaches to reduce the domain of the variables [60].

3.1.3 Modeling in Constraint Programming

The formulation of a problem as a Constraint Satisfaction Problem is intrinsically related to the approach used by the type of the constraint solver to be used, since several approaches to CP are significantly different from each other. The chosen solver can it self influence how a problem is modeled as a CSP. Although different solvers and approaches to model a CSP require different techniques, the concept to model the problem remains the same, independently of the tool being used.

Modeling a problem as a CSP, consists in deciding which are the variables of the problem; the domain of each variable; and which properties and relations among the variables should hold to model the problem.

When a solution to a CSP is found, it consists of an assignment of values to each variable, taken from their individual domain, respecting both properties and relations stated over the CSP. According to the problem and the needs of the user, one or more outcomes for the solving process can be sought for:

- one solution, with no preference for which one
- the best solution, according to some objective function
- all possible solutions
- whether there exists a solution

Listing 1 presents a formal representation of a CSP $P = (V, D, C)$, a triple of variables V, domains D and constraints C representing the problem. V represents the variables of the problem; D the domains of each variable; C the constraints which restrict the values of the variables and establishes relations between them. Expression 3.5 requires each variable $V_i \in V$ to take values from the respective domain $D_i \in D$.

Listing 1 CSP - Formal representation

$$P = (V, D, C) \tag{3.1}$$
$$V = \{V_1, \ldots, V_n\} \tag{3.2}$$
$$D = \{D_1, \ldots, D_n\} \tag{3.3}$$
$$C = \{C_1(V_i, \ldots, V_j), \ldots, C_m(V_i, \ldots, v_j)\} \tag{3.4}$$
$$\forall\ V_i \in V \Rightarrow V_i \in D_i \tag{3.5}$$

3.1.4 Constraint Solving Techniques

To reach a solution of a CSP, several techniques can be used, systematic search and consistency techniques are the ones most widely used.

Systematic search is an exhaustive method to reach a solution of a CSP, making a systematic exploration of the search place. Although a trivial algorithm is possible, it's not efficient, however, such algorithms are very important in Constraint Satisfaction, as they are the basis for more advanced and efficient ones.

Consistency-checking technique is yet another important approach to reach a solution of a CSP. This approach is based in removing values from the domain of the variables which are inconsistent with the constraints specified to model the problem. There are many consistency algorithms, but most of them are not complete, as such, usually this technique is never used alone, being complemented with other approaches.

These two approaches to Constraint Satisfaction can be used independently, but a common approach is to combine the two: search and consistency techniques. Constraint Propagation is such a technique [46].

The algorithms used to solve constraints can be either complete or incomplete. Complete algorithms are capable of finding a solution if it exists, and if doesn't exist, are capable of stating and proving that there is no solution to such problem, i.e. they explore the entire search space. As for the incomplete algorithms they are usually capable of finding a solution, although if there is no solution to such problem, they are not capable of proving there is no solution. Usually Constraint Propagation, which may rely on backtracking search is complete, while other approaches, such as local search are incomplete.

3.2 Propagation Based Solvers

Propagation [46] is one of the approaches most widely used in Constraint Programming. Using Propagation based solvers, the problem is described by stating constraints over variables. A constraint states what values are allowed to be assigned to each variable. The constraint solver will then propagate all the constraints and reduce the domain of each problem variable in order to satisfy all the constraints and instantiate the variables of the problem with valid values, thus, reaching a solution to the CSP.

Constraint Satisfaction Problems are usually NP-complete, normally solved through the use of backtracking search techniques, trying to fix a partial solution by extending it into a global, consistent solution. The idea of Constraint Propagation is to transform a given CSP into a smaller, tighter and simpler but equivalent CSP, thus reducing the search space used in the search mechanism. Constraint Propagation is achieved by repeatedly reducing the domain of variables, always keeping the equivalence between the original and resulting CSP by ensuring that constraints do not get falsified.

3.2.1 Constraint Propagation Algorithms

To achieve the reduction of variable domains, there are a number of known *atomic* reduction steps. An *atomic* reduction step is scheduled by the constraint propagation algorithm which tries to select the best reduction step for the purpose of reaching a property called *local consistency* [61].

The concept of *local consistency* is a central aspect to constraint programming. If a CSP is *locally consistent* it contains some parts which are considered consistent according to the constraints which defines the CSP.

The most popular definition of *local consistency* is called *hyper-arc consistency*, also known as *Generalized Arc Consistency (GAC)*, which, when dealing with binary constraints, is known as *arc consistency.* Basically, this definition means that for each variable in each constraint, all values of the variable's domain be part of a tuple which satisfies the constraints, in other words, each value of each domain must be part of a solution of the CSP. If all constraints in a given CSP are *arc consistent*, the CSP is also *arc consistent.*

There are many *arc consistency* algorithms, the most well known is AC3 [53], first presented as a binary system, and latter extended to an hyper-arc consistency or GAC algorithm, originating the GAC3 [62]. AC3 is not optimal regarding time complexity, thus, a new improved algorithm was later presented, the AC4 [63, 64]. While AC3 needs to redo much of the work when revisiting a node, AC4 stores a maximum amount of information, avoiding recomputation, keeping track of all values supported for the constraints. AC6 [65] is an evolution of AC4, standing in between AC3 and AC4, using the best techniques of both algorithms, keeping the optimal complexity of AC4 but stopping the search of support values as soon as one is found, just as in AC3.

3.2.2 Search Algorithms

Although some CSPs can be solved by using only Constraint Propagation, the vast majority of problems can not be solved using only this method: there is the need to complement these algorithms with other techniques. Due to this characteristic, Constraint Propagation is interwoven with a *search algorithm* [66] until a solution to the CSP is found.

The *search algorithms* are usually *top-down*, repeatedly expanding a node in the bottom level of a *search-tree* until a failure is found. When this occurs, it backtracks to an upper level in the *search tree* and then resumes the node expansion. The expansion of these nodes is known as the *branching strategy.*

The most basic backtracking search algorithm is very simple and naive, but very inefficient, still it is used as the basis of more complex and efficient algorithms. This algorithm is rather simple; it starts by assigning a value to a variable taken from the variable's domain, and then checks if this value in the current solution violates any constraint. If some constraint is violated, a new value is assigned to the variable. Once all values have been tried and there are still violation to some constraints, the algorithm backtracks to the previous variable, assigning it with a new value which hasn't been previously used. The algorithm terminates when either a solution is found; all solutions are found; or all values have been tested against all variables, indicating there is no solution to the problem.

Forward checking [67] is a look-ahead search algorithm, able to check constraints between past and future variables, opposed to the backtracking algorithm, which is only capable of checking constraints in the current and past variables. When a value is assigned to a variable by the forward checking algorithm, any value of the domain of a future variable which conflicts with the current assignment is temporarily removed from the domain of the future variable. With this approach, if in a given instance, a future variable presents an empty domain, it means the current partial solution is inconsistent, allowing the process to prune the search tree much sooner than the simpler backtracking algorithm.

Maintaining Arc Consistency(MAC) [68] is another look-ahead search algorithm, looking further ahead than forward checking when a value is assigned to a variable. Besides checking values against future variables, it checks values of future variables against each other. This way, any value which is not supported in the domain of other a future variable is immediately removed, as well as the ones which are not supported in the current assignment. This approach reduces the domain of future variables even more than forward checking.

3.2.3 Gecode

Gecode (GC) [69] is a very efficient, award winning constraint solver library based on constraint propagation. It is an open source system, implemented in C++, designed to be interfaced with other systems or programming languages and available for most used systems: GNU/Linux, Windows, and Mac. Gecode is widely used in both research and education, as it is the state-of-the-art in constraint programming and most importantly is an efficient and free open-source platform.

Since Gecode was designed primarily to be interfaced with other systems, several interfaces were developed by third party developers, e.g.: Gecode/J [70], a Java interface, a project which is still available but no longer maintained; Gecode/R [71], a Ruby interface to Gecode; AliceML [72], a dialect of Standard ML with constraint programming capabilities, using Gecode for constraint solving; GeOz [73] a project that integrates Gecode into the Mozart/oz environment; the Monadic Constraint Programming Framework [74] which allows to perform constraint programming in Haskell through the use of Gecode; GeLisp [75], a portable wrapper of Gecode to Lisp; among others.

Gecode is composed of a small and generic kernel which coordinates all constraint propagation needed to reach a solution. It provides simple interfaces for the variable domains, search heuristics and search engines, allowing the use of both Finite Domain Integers and Finite Domain Sets. The kernel interface allows an easy adaption of any these components to the specific needs of the problem being modeled.

Standard search algorithms are used in Gecode, such as depth-first search, limited discrepancy search, branch-and-bound optimization and Depth-first search (DFS) restart optimization. Gecode also allows for parallel search in distributed environments.

Gecode implements the most widely used constraints, enough to model most problems, nevertheless, sometimes there is the need to implement custom constraints to satisfy the needs of a specific problem. Due to the Gecode design architecture, it is very easy to implement new constraints, either by combining the constraints provided by Gecode and/or writing custom propagators which implement constraints specific to the needs of a given CSP.

3.3 Constraint-Based Local Search

Local Search [76] is the approach most widely used to solve combinatorial optimization problems, and it is suitable to solve Constraint Satisfaction Problems. In fact, Local Search is a very important approach to solve CSPs, as it is able to tackle very large and complex problems found in real-life applications.

Constraint-Based Local Search (CBLS) [77] combines Local Search techniques with Constraint Programming, using CP to model the problems in terms of constraints and requires heuristic functions to be defined for use in the Local Search component.

Although these are not complete methods to solve a CSP, as they are unable to guarantee completeness or optimality, these methods are widely used in Constraint Programming, because their sheer performance is sometimes the only way to solve complex problems.

The basic idea of Local Search is quite simple: it starts with an initial, candidate solution to the problem, generally randomly generated or through the use of some heuristics, which is then iteratively improved though minor modifications until a termination criterion is satisfied, according the specification of the CSP. When the criterion is satisfied, a solution is declared found.

The incremental modifications to the candidate solutions are usually guided by heuristic functions related to the constraints which model the problem, helping in the process of choosing the *starting point* for the next minor change to the solution.

Local Search may stagnate in local minima, unable to reach a valid solution. To prevent stagnation, various methods may be used, such as randomized iterative improvement, evolutionary algorithms, simulated annealing or tabu-search, to name a few.

Although CBLS algorithms are relatively simple to implement, they present very good performance figures and flexibility to adapt to changes in the specification of the problem, although the tuning of heuristic functions is a very sensitive aspect.

3.3.1 Local Search Algorithms

There are a number Local Search methods, but the simplest one, the basis for many complex and efficient algorithms, is the *hill-climbing* algorithm: it starts with a candidate solution, and at each step, selects a position to be improved from the current neighborhood through the use of heuristic functions. This process is then repeated until a satisfactory solution is found.

There are many *hill-climbing* based algorithms. The *Min Conflicts Heuristic (MCH)* [46], although the most simple, is a widely used Local Search algorithm: MCH iteratively assigns different values to each variable in order to minimize the number of violated constraints. More specifically, the algorithm starts by assigning random values to the variables, building the initial candidate solution, then, at each step, a randomly chosen variable still involved in conflicts is assigned with the value which minimizes the conflicts over it. This process is repeated until a solution is found, an objective solution is satisfied, or the maximum number of iterations has been reached.

3.3.2 Preventing local minima stagnation

Hill-climbing algorithms suffer from stagnation, as they may get stuck on local minima. This problem is the main limitation of this approach to CSP solving, but many methods can be used to avoid this problem. A simple and widely used method to avoid such stagnation in *hill-climbing* based algorithms is to restart the solving process after a specified amount of *max steps* have been performed.

Randomized Interactive Improvement

Another method which can be used to avoid local minima, is to occasionally allow the algorithm to select a step which doesn't improve the current solution, through random methods. The Randomized Interactive Improvement [46] is an extension to the *hill-clinging* algorithm, where, with a user-specified fixed probability, a randomly chosen step is selected for the next step, instead of selecting the step which best improves the current solution. This method, also called *random walk step* is also applied to many other Local Search algorithms.

Tabu Search

Tabu Search [46] is yet another common method to avoid local minima stagnation in *hill-climbing* based algorithms through the use of short term memory, preventing the search algorithm to visit a recently visited position during a specific number of steps. Although TabuSearch helps *hill-climbing* algorithms in avoiding being stuck at local minima, they present an undesirable side-effect: some parts of the search space can be easily overlooked. This can be minimized by the specification of a criterion which allows TabuSeach to visit recently visited positions, thus, resuming the normal operation of the *hill-climbing* algorithm, contradicting the *tabu-search* on certain occasions.

Dynamic Local Search

Dynamic Local Search [46], also known as Penalty-Based Local Search is yet another method to escape from local minima in *hill-climbing* based algorithms. This technique is obtained by modifying the heuristic functions when the search is about to stagnate in a local minima. Such modifications of the evaluation heuristic functions is achieved by the use of *penalty weights* which are related to specific properties of the candidate solution and to the constraints used to model the CSP.

3.3.3 Adaptive-Search

Adaptive Search (AS) [78] is a generic, domain-independent algorithm, for solving Constraint Satisfaction Problem, based on Local Search [76], not limited to a specific type of constraints and able to use a large class of constraints.

AS takes into account the structure of the problem, using variable-based information to design general heuristics which help solve the problem, guiding the solver more precisely than global objective functions, such as the number of constraints violated. To prevent stagnation on local minima, Adaptive Search uses a memory based mechanism similar to Tabu Search.

Adaptive Search is a peculiar solver, as it is designed to solve problems which can to be modeled as a permutation, i.e. in a problem modeled with `N` variables, each variable domain is $D = \{I, \ldots, I + N - 1\}$, where `I` is the lower value that each variable can take. A solution to such problem is a permutation of D. In effect, AS has an implicit "all-different" global constraint.

The iterative improvements to each candidate solution in AS are achieved by selecting one element of the solution and then swapping its value with that of one of its local neighbors. Both variables are chosen by means of a series of heuristics that match the problem being solved.

In general, Adaptive Search work as follows; start with a randomly generated candidate solution, then, at each step, compute the amount of error of each variable according to the constraints used to model the problem. The variable which presents the highest error is selected. Then, the cost of a new solution is evaluated considering all possible values that can be assigned to the selected variable. The value which provides the best next solution is assigned to the *culprit* variable, by means of a swap, and the process is repeated until a solution is found or a maximum number of steps has been reached. If no better solution is found, the variable is marked *tabu* and is ignored during a predefined number of steps.

In Adaptive Search, the constraints which model the problem are represented as *error functions* which inform "by how-much" each constraint is being violated. Besides the error functions which model the constraints, there are other heuristic functions which help solve the problem, the most important are the *Cost_ of_ Solution* and *Cost_ on_ Variable*. These *error functions* are then used to model the constraints, which guide the solver towards a solution.

Cost_ of_ Solution computes the amount of error of a candidate solution, while function *Cost_ on_ Variable* informs the algorithm about the cost of changing the value of a variable in the present candidate solution. These heuristic functions are used together to help find the *culprit* variable, the variable which brings most violations to the current candidate solution, thus, the variable which must see its value modified. The new value to be assigned to this variable is the one which minimizes the total number of violations, selected through the use of the same heuristic functions which model the constraints.

The specification of a CSP in Adaptive Search is achieved by implementing the heuristic functions:

1. *Cost_ of_ Solution*
2. *Cost_ on_ Variable*

Part of the CSP is modeled in *Cost_ of_ Solution*, where all necessary constraints to model the problem are given. This description of the problem has two purposes: on one hand, inform the amount of error of a given solution, on the other, inform if the solution is valid, i.e. the solution has a zero error value. The rest of the CSP is modeled via *Cost_ on_ Variable*, stating the constraints related to a specific variable given as parameter to the function *Cost_ on_ Variable*, allowing to compute the amount of error of a specific variable in a given candidate solution.

3.4 Boolean Satisfiability Problems

Another method of solving Constraint Satisfaction Problems is to transform the CSP into a Boolean Satisfiability Problem (SAT) [79] in order to take advantage of existing free and efficient SAT solvers [80]. A Boolean Satisfiability Problem consists in determining if there exists a valid assignment to all variables of a Boolean function, also known as a propositional formula, such that the boolean function is satisfiable, i.e. evaluating to true.

SAT is a prominent approach to the specification and solving of complex and practical problems in many areas, such as planing, circuit testing and software verification, automatic test generation, logic synthesis, among others. As a consequence, many efficient algorithms have been proposed and implemented.

These algorithms can be either complete or incomplete. In complete algorithms, the solvers are able to tell if there is a solution to a SAT problem and which one, as well as proving that a solution does not exist, if that is the case. As for the incomplete methods, while they are not capable of determining if there is effectively a solution which satisfies the given SAT, their performance makes them interesting in applications where it is not necessary to prove unsatisfiability.

SAT solvers date back to 1960, when Davis and Putnam [81] proposed an algorithm for Boolean SAT, which became known as Davis-Putman algorithm(DP). This algorithm was then enhanced to solve a problem of excessive memory usage, resulting in the Davis-Logeman-Loveland algorithm, also known as DPLL.

Over the years, many SAT solvers have been developed, but most of them rely on the DP or DPLL algorithms, combined with Local Search methods. This gives rise to new efficient methods to solve SAT problems. Although these algorithms are quite efficient, there are many problems which push them over the limit, leading to the development of new methods to improve the DPLL by optimizing some of its aspects. This brought about a new generation of solvers, such as Chaff [82], which allowed to solve real life problems, industry originated, consisting of millions of variables.

The propositional formula, or boolean formula, used to encode a SAT problem is usually presented in the form of a *product of sums*, also known as Conjunctive Normal Form (CNF) [79], which is a conjunction of clauses, each clause being a disjunction of literals, and each literal a boolean variable or its negation.

3.4.1 SAT Encodings

As previously mentioned, in order to model a problem with SAT, we need to represent it as a CNF formula. This modeling can be achieved in a different number of ways, called *encodings*.

Most the available encodings use the same process to represent variables, they assign a boolean variable to every possible value of each CSP variable [80], so that, for each CSP variable i, and for each value v of its domain, there will be a logical value $x_{i,v}$, which, when true, means the value v has been assigned to the CSP variable i. If variable $x_{i,v}$ has been assigned with a *true* value in a solution to such CSP, the value v has been assigned to variable i. On the contrary, if it has been assigned with a *false* value, it means the variable v can not be assigned to variable i in order to satisfy the CSP.

There are many methods to encode a CSP as a SAT problem, the most common ones being:

1. Direct Encoding
2. Support Encoding
3. Log Encoding

Direct Encoding

One of the most popular encoding is the *direct encoding* [83]. The *direct encoding*, uses the variable representation described above.

The *direct encoding* is composed by 3 types of clauses:

1. *at-least-one* clauses
2. *at-most-one* clauses
3. *constraints* clauses

The *at-least-one* and *at-most-one* clauses are used to model the variables of the CSP. The *at-least-one* clauses are used to indicate which values may be assigned to each CSP variable. In fact, there should be one *at-least-one* clause for each CSP variable i, imposing that at least one value v of its domain should be assigned to i, thus specifying the domain of each variable. Listing 2 represents the *at-least-one* clauses for CSP variable x_i, where $1 \ldots n$ are the values found in the domain of x_i.

Listing 2 *at_least_one clauses*

$$x_{i,1} \lor x_{i,2} \lor \ldots \lor x_{i,n} \tag{3.6}$$

The *at-most-one* clauses are used to make sure that only one value is assigned to a CSP variable, forbidding two values to be assigned to a variable at the same time. Listing 3 represents the *at-most-one* clauses for CSP variable i, where $1 \ldots n$ are the values found in the domain of i.

Listing 3 *at_most_one clauses*

$$(\neg x_{i,1} \vee \neg x_{i,2}) \wedge (\neg x_{i,1} \vee \neg x_{i,3}) \wedge \ldots \wedge (\neg x_{i,1} \vee \neg x_{i,n}) \tag{3.7}$$

$$(\neg x_{i,2} \vee \neg x_{i,3}) \wedge \ldots \wedge (\neg x_{i,2} \vee \neg x_{i,n}) \tag{3.8}$$

$$\vdots$$

$$(\neg x_{i,n-1} \vee \neg x_{i,n}) \tag{3.9}$$

The *constraint* clauses are the ones which actually model the problem. In the *direct* encoding the constraints are modeled by specifying the pairs of inconsistent assignments through the use of *conflict* clauses. Listing 4 represents two inconsistent value assignments, indicating that when value v is assigned to the CSP variable x_i, value w cannot be assigned to the CSP variable x_j and vice-versa. Such *conflict* clauses must be specified for all inconsistent value assignments of all variables of the CSP.

Listing 4 *conflict* clauses

$$\neg x_{i,v} \vee \neg x_{j,w} \tag{3.10}$$

Support Encoding

The *support* encoding [84, 85] is very similar to the *direct* encoding: variable domains are represented in the same way, through the use the *at-least-one* and *at-most-one* clauses. As for the constraints, these are represented using *support* clauses instead of *conflict* clauses.

Support clauses are achieved by specifying which assignments are compatible with a specific assignment, thus indicating which values may be assigned to which CSP variables when a specific value is assigned to a given CSP variable. These *support clauses* can be represented in CNF as in Listing 5, which states that if value v is assigned to CSP variable x_i, values $w_1, w_2, \ldots, w_k$ can be assigned to variable x_j.

Listing 5 *support* clauses

$$\neg x_{i,v} \vee x_{j,w_1} \vee x_{j,w_2} \vee \ldots \vee x_{j,w_k} \tag{3.11}$$

Log Encoding

The *log* encoding [86, 80] is very different from the *direct* and *support* encodings. The major difference is the way the domain of each CSP variable is specified.

In this case, for each CSP variable, there are $m = [log_2\ d]$ boolean variables, where d is the size of the variable domain, and each of the 2^m combinations represent a value assignment. In a *log* encoding, for each CSP variable i, there are x_i^b boolean variables, with $x_i^b = 1$, if and only if the bit b of the value assigned to a CSP variable i is set to *1*.

Log encoding does not use *at-most-one* and *at-least-one* clauses, meanwhile, if the cardinality of the variable domains is not a power of two, there is the need to prohibit the values which do not belong to the variable domains, achieved though the use of *prohibited-value* clauses.

The *constraint* clauses are usually specified in terms of *conflict* causes in the same as in the *direct* encoding, but using the variable representation of the *log* encoding.

Considering two CSP variables, x_i and x_j, with domains $\{0, 1, 2\}$, 2 bits are necessary to represent the variable domain values, but since the cardinality of the domain is not a power of 2, there is the need to prohibit the value 3, which does not belong to the variable domains. To do so, we must create *prohibited-value* clauses, as in Listing 6 which represents *i=3* and *j=3*.

Listing 6 *log-encoding prohibited-value clauses*

$$\neg x_i^0 \vee \neg x_i^1 \tag{3.12}$$

$$\neg x_j^0 \vee \neg x_j^1 \tag{3.13}$$

Using the same example, but considering the combination of assignments $[i = 2, j = 1]$ is prohibited in a solution to a given problem, we can model the constraints as in Listing 7.

Listing 7 *log-encoding - conflict clauses*

$$x_i^0 \vee \neg x_i^1 \vee \neg x_j^0 \vee x_j^1 \tag{3.14}$$

3.4.2 SAT Solver Implementations

Several SAT solvers participate in regularly staged competitions which rank their performance. MiniSat [87] is a widely used SAT solver which has been getting very good results in these events [88].

MiniSat is implemented so as to be a small, complete and efficient SAT solver. The major concerns of the MiniSat authors were to provide a tool that could be easily changed to match the needs of the user and also one that could be easily interfaced with other tools.

MiniSat is a conflict-driven SAT solver, inspired in both Satzoo [89] and Satnik [90], rethinking and simplifying these two solvers without sacrificing the performance. More specifically, MiniSat is based in the widely known DPLL algorithm, relying on conflict-driven backtracking [91], and boolean constraint propagation using *watched literals* [82].

As mentioned earlier, MiniSat is a very extensible tool, allowing its easy integration with other applications. MiniSat provides a simple C++ API which allows other applications to model a SAT problem directly into MiniSat, without the need to generate intermediate CNF files which would have to be parsed.

Thanks to this API, modeling a SAT problem in MiniSat is quite simple. This is achieved by calling API functions which allows to add new variables, new clauses and run the solver. In particular, the function *newVar()* creates a new variable, the function *addClause()* is used to add clauses which model the problem, and finally, the function *solve()* is used to obtain a solution to the problem, if it exists. The *solve()* function returns `FALSE` if there is no solution to the problem, `TRUE` otherwise. If a solution is found, it can be accessed though the public vector "model".

3.5 Conclusion

In this Chapter we introduced the Constraint Programming paradigm, a major approach to Declarative Programming with a focus on complex combinatorial problems. We covered the basic concepts and presented a brief history of CP, including the most important evolutions since it was introduced.

We also presented different approaches to Constraint Programming, including Constraint Propagation, Constraint-Based Local Search and Boolean Satisfiability Solvers: the approaches used in the work presented in this thesis.

We have also briefly introduced the specific solvers used in our work: Gecode, Adaptive Search and MiniSat.

Chapter 4

Modeling Intrusion Detection with Constraints

This Chapter describes how to model and perform Network Intrusion Detection, using a signature based approach to Network Intrusion Detection, resorting to Constraint Programming methodologies. We describe the architecture of the system as well as details of each detection mechanism available in NeMODe.

4.1 Introduction

Network Intrusion Detection Systems are one of the most important tools in computer network management to maintain the security, integrity and quality of computer networks and keep data safe. To maintain the quality and integrity of the services provided by a computer network, some aspects must be verified in order to maintain the security of data.

The description of those conditions, together with a verification that they are met can be seen as an Intrusion Detection task. These conditions, specified in terms of properties of parts of the (observed) network traffic, will amount to a specification of a desired or an unwanted state of the network, such as that brought about by a system intrusion or another form of malicious access.

Those conditions can naturally be described using a declarative programming approach, such as Constraint Programming [46], using Constraint Propagation methods [46], Constraint Based Local Search (CBLS) [76], or systems and mechanisms based on Boolean Satisfiability Problems(SAT) [79], enabling the description of these situations in a declarative and expressive way.

Using Constraint Programming to perform Network Intrusion detection allows to specify network intrusion signatures as relations between several network entities, enabling an easy way to describe and perform the detection of network attacks that span several network attacks.

4.2 Overall Architecture

The NeMODe system [2] is built to be a parallel intrusion detection system based on the Constraint Programming paradigm, by being able to run in parallel several detection mechanisms, based on different constraint programming methodologies in order to take advantage of the best solution produced, so, its architecture must reflect this parallelism as well as the various detection mechanisms available in the system.

NeMODe is composed of three major interconnected components; 1) a compiler, 2) a set of detection mechanisms based on Constraint Programming, and 3) a best solution selector.

The system is composed of two inputs and one output. As inputs, the system receives the description of a specific network situation and the network traffic. As output, the systems outputs the best match found to the problem, i.e. a set of network packets that prove the existence of the attack, if it exists on the network traffic.

The intrusion to be detected is described in a custom built language for the NeMODe system, with terminology related to computer network, *talking* about network entities and relations between those entities.

The description of the intrusion is then fed to the compiler, which parses it into a semantic model and generates code for each of the intrusion detection mechanisms available on the system, according to the described network situation. At present, the compiler generates code for Gecode, Adaptive Search and MiniSat.

After all recognizers have been generated, each back-end receives as input the network traffic and produces a valid solution, if the intrusion described as a NeMODe program exists on the network traffic that was given as input to each back-end detection mechanism.

In a final stage, when all back-end detection mechanisms have been generated, each one will provide a solution that identifies the specific network situation. From those solutions, there is the need to choose one. Since every solution provided by any of the solvers is a valid solution, the system may simply choose the first solution as the best solution to the problem.

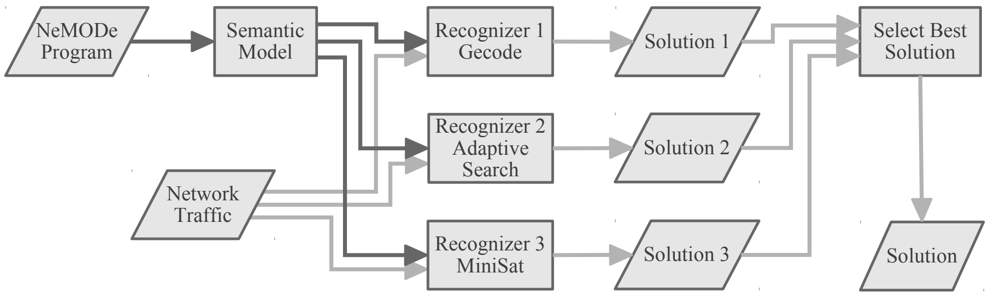


Figure 4.1: NeMODe system architecture

Fig. 4.1 represents the architecture of the system, including the compiler, the detection mechanisms and the solution selector. It also represents the data flow between each component.

4.3 Modeling intrusions as a CSP

Our approach to intrusion detection relies on being able to describe the desired network attack signatures as a CSP and then identify the set of packets that match the target network situation in the network traffic, through the use of several detection mechanisms based on constraint programming, such as: Gecode, a Propagation based solver; Adaptive Search, a Constraint Based Local search algorithm and MiniSat, a solver for Boolean Satisfiability Problems.

In order to be able to use Constraint Programming on Network Intrusion Detection, the intrusion signature to be detected needs to be modeled as a Constraint Satisfaction Problem (CSP), by stating relations between a set of variables with a specific domain. This CSP is composed by:

- a set of variables V, representing the network packets;
- the domains D for the variables V;
- a set of constraints C, which relates the variables to describe the network situation.

On this CSP, each network packet variable is a tuple of integer variables, representing the significant fields of a network packet necessary to model the intrusion signatures that we are interested in.

The domains of the network packet variables, D, are the values actually seen on the network traffic window, a set of tuples of integer values, each tuple representing a network packet actually observed on the network traffic window and each integer value representing the fields that are relevant to network monitoring.

The payload of each packet is stored separately in an array containing the payload of all packets seen on the traffic window.

A solution to such a CSP, if it exists, is the set of packets that correspond to the network intrusion described by the CSP.

Listing 8 Representation of a network CSP

$$CSP = (V, D, C) \tag{4.1}$$

$$V = \{V_1, V_2, \ldots, V_n\}, \quad \forall V_i \in V : V_i = (X_1, X_2, \ldots, X_z) \tag{4.2}$$

$$D = \{D_1, D_2, \ldots, D_m\}, \quad \forall D_i \in D : D_i = (Y_1, Y_2, \ldots, Y_z) \tag{4.3}$$

$$\begin{aligned} &Data = \{Data_1, \ldots, Data_m\}, \\ &\qquad \forall\ Data_i \in Data, \forall\ D_i \in D : payload(D_i) = Data_i \end{aligned} \tag{4.4}$$

$$\begin{aligned} &C = \{C_1(V_i, \ldots, V_j), \ldots, C_k(V_i, \ldots, V_j)\} \\ &\qquad \forall C_k \in C : C_k = \{CF_1(V_i, \ldots, V_j), \ldots, CF_l(V_i, \ldots, V_j)\} \end{aligned} \tag{4.5}$$

Listing 8 shows a network CSP, represented by the triple $CSP = (V, D, C)$, where V is a set of network packet variables, and each $V_i = (X_1, X_2, \ldots, X_z)$ is a network packet variable, composed by a set of integer variables; D is the set of domains which are the packets found in the network traffic, where each packet $D_i = (Y_1, Y_2, \ldots, Y_z)$ is a set of integer values, representing a network packet found in the network traffic. $Data$ are the payloads of all packets found in the network traffic, where the payload of packet D_i is $Data_i$; and C is the set of constraints which describes the attack to be detected, where each constraint C_k is mapped into a composition of *library* and network specific constraints.

4.3.1 Problem Variables

To model a Network Intrusion Detection problem as a CSP, the variables of the problems represent fields in network packets, which, when assigned some value, will identify the network packets that make up the network signature used, the *trail* left by the attacker.

A very important step in describing a Network Intrusion Detection problem as a CSP is to decide of how many variables the problem will be composed of. This will be tied to the number of packets necessary to identify the desired intrusion: those used in the signature of the attack, which will constitute the proof of the attack.

4.3.2 Variable domain

A very important step in modeling the Network Intrusion Detection problem as a CSP is to ensure that a network packet variable has the correct domain, otherwise the solver can produce valid solutions according to the problem description, but which do not occur in the network traffic being analyzed.

The domain of the network packet variables must be the network packets actually seen on the network traffic, so that the variables can only take values that make sense, the ones that correspond to real network traffic, this way, a solution that is valid according to the problem specification is also valid according to the given network traffic, since the solution will be a subset of the network traffic.

Listing 9 shows a formal representation of a network packet domain, where V_i is a network packet; D is the domain of the packets, the packets found in the network traffic; and expression 4.8 states that packet V_i must belong to the current network traffic.

Listing 9 Domain of variable V_i

$$V_i = (V_1, V_2, \dots, V_z) \tag{4.6}$$

$$D = \{D_1, D_2, \dots, D_m\}, \quad \forall D_i \in d : D_i = (Y_1, Y_2, \dots, Y_z) \tag{4.7}$$

$$\forall V_i \in V, \ \forall D = \text{network\_traffic} \ : \ V_i \in D \tag{4.8}$$

4.3.3 Constraints

One of the major steps in modeling a Network Intrusion Detection problem as a CSP is to state constraints over the network packet variables in order to describe relations between the network packet variables, so we can model the desired network situation.

Most solvers provide built-in constraints which can be combined in order to model most of the constraints and relations necessary in a Network Intrusion Detection problem, such as the ones that force a packet to verify some property or relation with some other packets. Other constraints, such as the ones that require a packet to contain a specific datum in the payload or state some relations between the payload of two network packets can't be implemented using the built-in constraints available in the solvers, since they usually don't have the capability of working with strings, thus not being able to state constraints over strings. In these specific situations, we have to create custom network constraints to describe these relations.

4.3.4 Network Entities

To model Network Intrusion Detection problems, NeMODe provides, as network entities to model network attacks, several network packet types, as well as some individual fields, essential to model Network Intrusion Detection problems. At this time, NeMODe provides 3 types of network packets, although this can easily be extended to cope with other packet types. The ones that were needed in our experiments are:

1. Transmission Control Protocol (TCP) packets
2. User Datagram Protocol (UDP) packets
3. Address Resolution Protocol (ARP) packets

Each of these is represented as an `Array` of 25 `Integer` values, the size of the larger network type, ARP packets. While the ARP packets use all 25 fields, other network packets don't, as they have less fields, so, these *extra* fields are ignored. The network payload of both UDP and TCP packets are represented separately as a `String`. Besides the fields found on the network packet, we decided to add an extra *identification* field which identifies the packet and helps relate the packet with its payload.

The TCP packets use 23 `Integer` fields of the 25 available and the UDP use 14. Although NeMODe is prepared to use these network packet types with these number of fields, it can easily be extended to cope with more fields on each network packet type.

TCP and UDP network packet share the first 14 fields, the fields related with packet time-stamp and source/destination. These fields are listed in Table 4.1.

UDP packets use only the fields described in Table 4.1, but TCP packets need some extra fields to represent the TCP specific fields: the TCP flags and the acknowledgment number. Table 4.2 presents the extra TCP fields as well as its position in the array which represents the network packet.

As for the ARP packets, they are represented as an `Array` of 23 `Integers`, representing the fields necessary to model the problems we experimented. Table 4.3 presents the fields used to represent the ARP packets as well as their position in the `Array` representing the network packets.

Array position	Packet Field
0	Packet ID
1	Packet type - TCP, UDP or ARP
2	Time-stamp(seconds)
3	Time-stamp(microseconds)
4	1^{st} octet of the Source Address
5	2^{nd} octet of the Source Address
6	3^{rd} octet of the Source Address
7	4^{th} octet of the Source Address
8	Source Port
9	1^{st} octet of the Destination Address
10	2^{nd} octet of the Destination Address
11	3^{rd} octet of the Destination Address
12	4^{th} octet of the Destination Address
13	Destination Port

Table 4.1: TCP and UDP *Shared* Fields

This representation is used for both the network packet variables and packets in the traffic, although, depending on the back-end detection mechanism used of the solver, the internal representation of the network packet variables may differ due to specific limitations of the solvers used by these detection mechanisms. Further on, we explain the details of the variable representation on each of available detection mechanism.

Network Traffic

NeMODe needs to represent the network traffic internally so it can detect the desired network signatures. We decided to do this as a set of packets, each one represented as in Sect. 4.3.4, using `Arrays` to represent such set. So, the network traffic is represented by an `Array` of `Arrays` of 25 `Integer`, the number of fields of the larger network packet type.

Besides the network packet fields described above, there is the need to represent the TCP and UDP payload, which is essentially treated as a `String`. So, we decided to store the payload of the network packets in a separate `Array` of `Strings`, each `String` representing the payload of one packet. The connection between the network packet and its payload is achieved by the internal network packet identification number, which

Array position	**Packet Field**
14	CWR
15	ECE
16	URG
17	ACK
18	PSH
19	RST
20	SYN
21	FIN
22	Acknowledgment Number

Table 4.2: TCP *Extra* Fields

is its position in the set representing the network traffic and will be used to reference the packet payload. So, the payload of packet `traffic[i]` is `payload[i]`.

Listing 10 represents the network traffic together with the packet payload, where `traffic` represent the traffic, `payload` the payloads found in the network traffic, and `n` the number of packets in the traffic.

Listing 10 Reified Constraints

$$\text{traffic} = [\ [V_{0,0},\ \ldots,\ V_{0,22}],\ \ldots,\ [V_{n,0},\ \ldots,\ V_{n,22}]\ ] \tag{4.9}$$

$$\text{payloads} = [\ P_0,\ \ldots,\ P_n] \tag{4.10}$$

4.4 Modeling with Propagation Based Solvers

Propagation Based systems relies on functions which reduce variable domains, which in turn reduces the search space, until no more violations to the constraints used to model the problem are found, and all variables are reduced to a single-valued domain, thus reaching a solution to the problem, if one exists.

In order to model a problem in Propagation Based Solvers, we need to model the problem as a CSP, and to do so, three things need to be defined: 1) the variables of the problem, 2) the domain of the variables, and 3) the constraints which describe the problem. In the case of Network Intrusion Detection problems, the same modeling will be used independently of the solver being used.

Array position	**Packet Field**
0	Packet ID
1	Packet type
2	Time-stamp(seconds)
3	Time-stamp(microseconds)
4	Operation
5	1^{st} octet of Sender Hardware Address (SHA)
6	2^{st} octet of Sender Hardware Address (SHA)
7	3^{st} octet of Sender Hardware Address (SHA)
8	4^{st} octet of Sender Hardware Address (SHA)
9	5^{st} octet of Sender Hardware Address (SHA)
10	6^{st} octet of Sender Hardware Address (SHA)
11	1^{st} octet of Sender Protocol Adress (SPA) (IPV4)
12	2^{st} octet of Sender Protocol Adress (SPA) (IPV4)
13	3^{st} octet of Sender Protocol Adress (SPA) (IPV4)
14	4^{st} octet of Sender Protocol Adress (SPA) (IPV4)
15	1^{st} octet of Target Hardware Adress (THA)
16	2^{st} octet of Target Hardware Adress (THA)
17	3^{st} octet of Target Hardware Adress (THA)
18	4^{st} octet of Target Hardware Adress (THA)
19	5^{st} octet of Target Hardware Adress (THA)
20	6^{st} octet of Target Hardware Adress (THA)
21	1^{st} octet of Target Protocol Adress (TPA) (IPV4)
22	2^{st} octet of Target Protocol Adress (TPA) (IPV4)
23	3^{st} octet of Target Protocol Adress (TPA) (IPV4)
24	4^{st} octet of Target Protocol Adress (TPA) (IPV4)

Table 4.3: ARP Packet Fields

4.4.1 Modeling in Gecode

Modeling Network Intrusion Detection problems in Gecode is basically done by asserting relations between the variables of the problem in order to describe the desired Network Intrusion Signatures.

3 main steps need to be done to model a Network Intrusion Detection in Gecode:

1. model the variables
2. specify the domains of each variable
3. specify the constraints in order to model the problem.

Variable representation

In Gecode, the primary type of the variables is `Integer`. There are helpers which ease the description of the problem in terms of variables, such as the type `Integer Array`, an array of variables of type `Integer`. This is a perfect data type to represent the network packet variables in a Constraint Satisfaction Problem, since it can easily be represented as a tuple of `Integer` variables, as provided by the `Integer Array`.

To represent the problem variables, which we call *network packet variables*, we decided to use the `IntVarArray` which can be used as a single variable, instead of using the individual variables found in the array. This data type allows to easily represent the network packet variables, since each network packet variable is a tuple of `Integer` variables, which can be directly mapped into a `IntVarArray`.

The first step to model a problem is to declare the necessary packets to model the problem. Gecode provides an easy way to declare `IntVarArray`, by simply stating the number of `Integer` variables, and the upper and lower bounds of the domain of each `Integer` variable. This procedure is done to each network packet variable of the problem, so, the variables of the problem will be a set of `IntVarArray`.

Listing 11 presents the initialization of the network packet variables in Gecode, where `vars` is the array with all network packet variables, In this case, a signature with 5 packets is used, so, `5 IntVarArray` variables are created, and then initialized with `19 Integer` variables with a domain ranging from `0` to `1000000`[1], which is the range of values that each `Integer` variable can take in Intrusion Detection problems. In this example, we use an `IntVarArray` composed of `19 Integer`, since the example uses TCP packets. For other types of packets, different sizes are used.

[1]This is the largest number that can occur in the header of a packet, the timestamp.

Listing 11 Gecode Variable Initialization

```
    ...
    IntVarArray vars[5];
    ...
    for(i=0; i<5; i++)
        vars[i]=IntVarArray(*this,19,0,1000000);
    ...

```

Using only `Integer` variables poses a problem, since some network packet fields need to be represented as a set of characters, such as the network payload. The payload on a network packet is a completely different field from other fields, since it is a set of opaque data, which can be *understood* as a `String`. Such fields cannot be represented together with other fields of the network packets variables, since `IntVarArray` is composed only by `Integer` variables, and Gecode is not prepared to work with `String` data-structures.

To deal with this situation, we decided to add an extra `Integer` variable in the `IntVarArray` representing the network packet variables, which, when a solution is found, the value assigned to that variable will be a *pointer* to the payload of a specific network packet of the network traffic. This simplifies the representation, since the payloads will be treated as normal `Integer` variables.

Variable domain

To model the variables in Gecode, we decided to use `IntVarArray`, a set of `Integer` variables. The initial domain of such variables is defined by an interval of values, not allowing to specify initial domains as sets of non contiguous values. This situation is not the most desirable to represent the domain of the variables of a Intrusion Detection problem, which is composed of a set of sparse values which fit in a very large interval, ranging from low values to very high values.

So, the domain of the problem variables will have to be a contiguous `Integer` interval, including all possible values seen on the network traffic. This type of domain is not suitable for Network Intrusion Detection problems, since this way, the solver would generate solutions that don't exist on the network traffic, which makes no sense since we are looking for a set of network packets with specific properties which are actually found in the network traffic source. In order to solve this, the domain has to be narrowed down to the values which occur in the network traffic by using constraints to limit the possible values that each variable can take.

In a Network Intrusion Detection problem, the domain of the network packet variables must be the packets actually seen in the traffic, since the only solutions to the problem that make sense are the ones composed by packets found on the network traffic. Narrowing the variable domain is one of the major problems in modeling this type of problems in Gecode, since it affects the performance of the system significantly.

The packets in the network traffic are composed by individual values of two types; integer values to represent the network packet fields and strings or set of character values to represent the payload.

The integer values found on a network packet range from small values to very high values, but only a few of those are found in the network packets, thus, most of the values found on the interval which contains all the values found on the network traffic are not used. Using this set of values to describe the domain of the network packet variables leads to a problem: producing solutions that are valid according to the description but not according to the network traffic, since they can not be found on the traffic.

Gecode only allows the specification of variable domains as interval of integers, which, is not the most suited to these types of problems, so, there is the need use some mechanisms provided by Gecode to constraint the values that each variable can take to the ones actually seen on the network traffic, despite the initial variable domain being much larger.

To limit the domain of the values that can be assigned to each variable we used two main approaches [7, 6]; 1) using reified constraints to impose restrictions over each `Integer` variable of each `IntVarArray`, limiting the values of each variable, and 2) using `Extensional` constraints, which force a tuple of constraint variables of a problem to take values from a set of tuple of values.

Reified [46] constraints allows to state constraints in a form of logical connectives, such as conjunctions and disjunctions. This allows to state constraints such as in Listing 12, forcing variable A to take the values 5 or 10, if those values pertain to the initial domain of variable A.

Listing 12 Reified Constraints

$$A = 5 \lor A = 10$$

The reified constraints can be applied to the individual `Integer` variables of each `IntVarArray` variable in order to reduce the domain of each `Integer` variable to the values found the network traffic. Each `Integer` variable represent a field of a network packet, so, the domain of such variable should be reduced to the values actually found for that specific field in all packets of the network traffic. This is not enough to ensure that the variables will be assigned with valid values, since each the individual `Integer` variables of the each `IntVarArray` should be assigned with values from a single network packet, since the individual values of a network packet don't make sense when used alone.

Listing 13 represents how the domain of the network packet variables can be reduced to the packets seen on the network traffic using reified constraints in Gecode, where D represents the set of network packets seen on the network traffic and $(D_{i,1}, \dots, D_{i,22})$, a network packet, in a total of m packets. V represents the set of network variables used to model the problem, in a total of n packets and $(V_{i,1}, \dots, V_{i,22})$ represents a network packet variable. Expressions 4.13 and 4.14 represents the reified constraints applied to the network packet variables V_1 and V_n.

Listing 13 Reified Constraints

$$D = \{(D_{1,1}, \dots, D_{1,22}), \dots, (D_{m,1}, \dots, D_{m,22})\} \tag{4.11}$$

$$V = \{(V_{1,1}, \dots, V_{1,22}), \dots, (V_{n,1}, \dots, V_{n,22})\} \tag{4.12}$$

$$\begin{aligned}((V_{1,1} = D_{1,1} \lor V_{1,1} = D_{2,1} \lor \dots \lor V_{1,1} = D_{m,1}) \land \\ (V_{1,2} = D_{1,2} \lor V_{1,2} = D_{2,2} \lor \dots \lor V_{1,2} = D_{m,2}) \ \land \dots \\ \land \dots \ \land (V_{1,22} = D_{1,2} \lor V_{1,22} = D_{1,22} \lor \dots \lor V_{1,22} = D_{m,22})) \lor \dots\end{aligned} \tag{4.13}$$

$$\vdots$$

$$\begin{aligned}\dots \lor ((V_{n,1} = D_{1,1} \lor V_{n,1} = D_{2,1} \lor \dots \lor V_{n,1} = D_{m,1}) \land \\ (V_{n,2} = D_{1,2} \lor V_{n,2} = D_{2,2} \lor \dots \lor V_{n,2} = D_{m,2}) \ \land \dots \\ \land \dots \ \land (V_{n,22} = D_{1,2} \lor V_{n,22} = D_{1,22} \lor \dots \lor V_{n,22} = D_{m,22}))\end{aligned} \tag{4.14}$$

Another approach to limit the domain of the network packet variables is to use *extensional* constraints, which force a tuple of variables (`IntVarArray`) to take values from set of tuples of values, like a matrix. Reducing the variable domain this way simplifies the process, since the network packet variables are tuples of variables, and the network traffic a set of tuples of values.

Listing 14 represents the use of extensional constraint, where V represents the set of network packet variables; $(V_{i,1}, \dots, V_{i,22})$ a network packet variable; D a matrix representing the network traffic; and $(D_{i,1}, \dots, D_{i,1})$ a network packet found in the network traffic. Expression 4.17 of Listing 14 represents the extensional constraint applied to any network packet variable V_i, over the matrix D, forcing V_i to belong to D, once a solution is found.

Listing 14 Extensional constraints

$$V = \{(V_{1,1}, \ldots, V_{1,22}), \ldots, (V_{n,1}, \ldots, V_{n,22})\} \tag{4.15}$$

$$D = \{(D_{(1,1)}, \ldots, D_{(1,22)}), \ldots\ , (D_{(m,1)}, \ldots, D_{(m,22)})\}, \tag{4.16}$$

$$\forall\ V_i = (V_{i,1}, \ldots, V_{i,22}) \in V,\ \forall D,\ extensional(V_i, D) \Rightarrow V_i \in D \tag{4.17}$$

The use of extensional constraints greatly simplifies the reduction of the variable domains but most importantly, it improves the performance of Gecode when compared to using reified constraints.

Although the use of extensional constraints turned out to be quite successful, Gecode presents performance issues when the domain of the variables range from very low values, such as 0, to very high values, such as 1000000, the range of values found on the network traffic. To deal with this situation, Gecode provides the *element* constraint which allows to use an un-instantiated variable as an index into an array of values or variables. This allows to translate the matrix representing the network traffic into a matrix of indexes into an array with all distinct values that occur in the original network traffic, thus reducing the range of the domains, but still respecting the network traffic as the variable domain.

Combining the element with the extensional constraint allows the use of the translated network traffic, containing only indexes to values, instead of the original network traffic matrix with the original values.

The use of the extensional constraint combined with the element constraint is represented in Listing 15, where V represents the set of network packet variables to model the problem; $V_j = (V_{j,1}, \ldots, V_{j,22})$ a network packet variable; D the network traffic; $D_i = (D_{(1,1)}, \ldots, D_{(1,22)})$ a network packet found on the traffic; DV all the individual values found in the network traffic D; and DT the network traffic D represented as indexes.

The network packet payload is treated differently from the other fields, but due to its representation, the approach used to ensure a valid network packets domain can be applied without any concern about the payload, since this is treated as an integer variable, just like the remaining network packet fields.

Listing 15 Extensional and element constraints combined

$$V = \{(V_{1,1}, \ldots, V_{1,22}), \ldots, (V_{n,1}, \ldots, V_{n,22})\} \tag{4.18}$$

$$D = \{(D_{(1,1)}, \ldots, D_{(1,22)}), \ldots, \ldots, (D_{(k,1)}, \ldots, D_{(k,22)})\}, \tag{4.19}$$

$$DV = \{DV_{(1,1)} \cup \ldots \cup\ DV_{(1,22)} \cup, \ldots, \cup\ DV_{(k,1)} \cup \ldots \cup\ DV_{(k,22)}\}, \tag{4.20}$$

$$DT = \{(DT_{(1,1)}, \ldots, DT_{(1,22)}), \ldots, (DT_{(k,1)}, \ldots, DT_{(k,22)})\}, \tag{4.21}$$

$$\begin{aligned} &\forall\ V_j = (V_{j,1}, \ldots, V_{j,22}), DT_{(k,i)}, \in DT, \\ &\qquad extensional\_element(V_j, D, DT) \Rightarrow\ V_i \in DT,\ DV_{DT_{(k,i)}} = D_{(k,i)} \end{aligned} \tag{4.22}$$

Constraint specification

The most important part in describing a Network Intrusion Detection problem in Gecode is the specification of the constraints. Stating the constraint over the network variables is what models and describes a specific attack, which exists in the present network traffic iff the constraint problem has a solution.

The constraints are stated over one or more network packet variables, and are responsible to specify and ensure which properties are verified by each packet and which relations should hold between multiple packets, as used to model the problem.

Gecode provides built-in constraints over `Integer` variables which allows the modeling of many types of problems. On Network Intrusion Detection problems, many of these constraints can be applied to the network packet variables, which are `Integer`.

These built-in constraints can be combined to model most of the necessary constraints in Network Intrusion Detection problems: either the simple constraints which states that a network packet should verify some property, or more complex constraints that relate two or more packets, such as stating that the source address of packet A should be equal to the destination address of packet B.

Other constraints, such as the ones that force a packet to contain a specific data in the payload or state some relations between the payload of two network packets can't be done through the use of the built-in constraints available in the solvers, since they can't work with strings, so, we had to create custom constraints to describe these relations.

Although the built-in constraints allows the description of most constraints necessary to model Network Intrusion Detection problems, they are not very user friendly, since they are prepared to work with `Integer` variables and there is the need to combine them in order to describe most of the constraints.

So, in order to ease the description of the problems using the built-in constraints, specific network *meta* constraints combining several Gecode built-in constraints, which can be directly applied to the network packet variables instead of the individual `Integer` variables of the network packet components.

Listing 16 represents a simple *meta* constraint which forces a TCP packet to have its *SYN* field set, forcing it to be a *SYN* packet, where `var` represents the network packet variable to which the constraint will be applied, and `var[20]` is the `Integer` variable that represents the *SYN* field. In this simple case, the built-in Gecode constraint is specified in Line 2, which states that a specific `Integer` variable must have a given value, in this simple case, the variable representing the SYN flag of the TCP packet should have the value of `1`, meaning it has the SYN flag set.

Listing 16 `must_be_syn` *meta* constraint

```
1     void must_be_syn(IntVarArray var){
2       post(*this, var[20] == 1);
3     }
```

The use of Gecode built-in constraints is not enough to model all properties and relations that need to be verified in Network Intrusion Detection problems, such as, constraints over the packet payload, which can require a network packet variable to contain a specific expression on its payload or even relate two or more network packets according to some relations over their payloads. These have to be implemented as custom constraint propagators in Gecode.

Gecode provides generic propagator classes which can be extended to create new constraints. In case of the payload related constraints, two main factors are considered when creating the custom propagators, 1) the field of the network packet variable that correspond the payload, and 2) the network traffic being analyzed.

The propagator is implemented in a way to reduce the domain of the `Integer` variable that *points* to the payload of the packet, by successively checking which network packets verify the desired expression or relation in their payload. The ones that do not match, or do not respect the desired relation are removed from the domain of the `Integer` variable representing the packet payload.

The propagator *collects* all indexes of the packets that match the desired constraint, computes an intersection of the set of valid values with the set of values in the domain of the variable, thereby, removing the invalid values from the domain.

Due to the constraints that ensure that a packet variable only takes values from a single network packet at a time, removing the values that represent the packet that do not respect the desired expression also leads to the removal of the values of the other fields of the same packet from the domain of the respective fields.

Listing 17 presents a snippet of pseudo-code that represents a custom payload propagator. Lines 2-6 traverse all network packets of the domain, Line 3 verifies if the packet matches the desired expression, and, if so, in Line 4, the index of that packet is added to `M`. After all packets in the domain have been analyzed, in Line 6, the domain of the `Integer` variable is intersected with set `M`, thus removing from the domain the indexes to the packets that violates the desired expression.

Listing 17 payload custom propagator - pseudo-code

```
1   ...
2   FOR all packets in domain
3      IF packet matches expression THEN
4         collect packet in M
5      ENDIF
6   ENDFOR
7   ...
8   variable domain ← INTERSECTION(variable domain, set M)
9   ...
```

Modeling

After the network packet variables have been specified, the variable domain has been set, and all necessary constraints have been implemented, the desired network case can be modeled. In order to do so, the necessary constraints are applied to the variables of the problem.

Listing 18 show a snippet of Gecode to model a simple illustrative case. Two TCP network packet variables are created in Line 2, the network traffic is declared in Line 3, in this case allowing for `10000` TCP packets. From Line 5 to Line 7 the problem is described, forcing the first network packet to by a *SYN* packet and have as its destination port, the port 80. Then, a constraints is applied to both packets so that the source port of the second packet has be the same as the destination port of the first packet.

Listing 18 problem modeling - code snippet

```
    ...
    IntVarArray vars[2];
    int traffic[10000][22];
    ...
    must_be_syn(vars[0]);
    dst_port(vars[0], 80);
    equal_src_dst_port(vars[1],vars[0]);
    ...
```

4.5 Modeling with CBLS Solvers

When using Constraint-Based Local Search to perform Network Intrusion Detection, two major decisions need to be done: 1) decide the number of network packets that needs to be found in order to identify the network situation being sought, and 2) decide the constraints which model the problem.

In Constraint-Based Local Search, the constraints are built in order to drive a heuristic search, by providing the number of violations of each variable used in the constraints, guiding the search algorithm to a solution to the problem.

Constraint-Based Local Search starts by creating a first tentative solution by assigning values, usually randomly chosen, to the variables of the problem. It then performs small changes to that solution in order to converge on a solution, using heuristics to decide which changes will be done. This step is repeated until an objective function is reached, meaning that a solution has been found. The initial tentative solution is very important in the way the solution is reached, so, picking an initial solution that suits Intrusion Detection problems may be important.

4.5.1 Modeling with Adaptive Search

Adaptive Search (AS) is a Constraint-Based Local Search (CBLS) algorithm, taking into account the structure of the problem and using variable-based information to design general heuristics which help solve the problem. The iterative repairs to the candidate solution are based on variable and constraint error information which seeks to reduce errors on the variables used to model the problem.

In Adaptive Search, the constraints are used to drive an heuristic search, guiding the search algorithm to reach a solution to the problem. They are modeled as a set of heuristic functions which have the purpose of indicating the error of the variables to which the constraint have been applied, considering the purpose of the constraint.

Modeling a Network Intrusion Detection problem as an Adaptive Search problem relies on 3 main definitions:

1. `V`, the set of network packet variables necessary to describe the sought-after signature attack.
2. `D`, the observed network traffic where the attacks will be looked for, containing all the network packets seen on a network traffic window.
3. `C`, a set of constraints which describe the network situation.

Variables

Adaptive Search presents a particularity on the problems that it can solve, they have to be modeled as a permutation, i.e. in a problem using `N` variables, the domain of each variable will be $D = \{I, \ldots, I + N - 1\}$, where `I` is the lower value that each variable can take, and a solution to such problem will be a permutation of `D`.

Considering the set of network packet variables `V`, necessary to represent the desired Network Intrusion Detection problem, and the network traffic, `D`, the domain of the network packet variables, `D` being larger than `V` makes a Network Intrusion Detection problem incompatible with Adaptive Search, since a valid solution to the problem would be a subset of `D` instead of a permutation of `D`, which could compromise the modeling of a Network Intrusion Detection with Adaptive Search.

To work around this problem, we model the network situation using as many variables as the number of network packets in the network traffic. Most of these variables are ignored by the heuristic functions, which only use the ones needed to describe the signature.

So, if the network situation being modeled uses `N` network packet variables and the network traffic is composed of `M` packets, the problem will be modeled using `M` variables, of which only the first `N` will be used to reach a solution, ignoring the last $M - N$ variables. This way, a solution to the problem will be a set of `M` variables, but only the first `N` are the ones with the packets that identify the attack.

In order to use only the N variables that describe the network situation, several approaches have been used: first, all the constraints used to describe the network situation are applied only to the first N variables, which are the variables really necessary to model the problem. The second approach is to give a *null* error value to all variables which are not used to model the problem, the variables $\{P_{n+1}, \ldots, P_m\}$. Assigning a *null* error value to these variables will prevent Adaptive Search from choosing them as the candidate for a value swap.

Also due to that characteristic of Adaptive Search, in order to model a network signature in AS, the packets of the network traffic were indexed: each variable represents a packet, and when assigned a value, indexes a packet in the network traffic, not a set of variables representing all fields of the network packets. This way, the variables of the problem will be a set of integer variables, each one representing a network packet.

Using a set of indexing variables to represent the network packet variables facilitates the problem of dealing with the network packet payloads, which cannot be treated equally to the other packet fields. Since a network packet variable is only the index to a packet in the network traffic, any verification that has to be done over that variable is made on the packet to where the index points to.

Variable Domain

Because Adaptive Search requires the problems be stated as a permutation, the modeling of Network Intrusion Detection problems in Adaptive Search gets more complex, but due to this fact, and the way that the network packet variables are represented, restricting the domain of each network packet variable to the packets found in the network traffic is actually simple, since a solution to the problem will be subset of a permutation of the indexes to network packets found on the network traffic.

Listing 19 represents the network packet variables in Adaptive Search as well as its domain; `D` representing the network traffic, $D_i = (D_{(1,1)}, \ldots, D_{(1,22)})$ a network packet window found on the network traffic, `I` represents the indexes of each network packet found on the network traffic, where I_i is the index to the packet D_i, `V` the set of network packet variables used to model the problem, where $\{V_1, \ldots, V_n\}$ are the network packets actually used to model the problem, $\{V_{n+1}, \ldots, V_m\}$ the network packet variables ignored by the solver.

Expression 4.26 represents the relation between the indexes `I` and the network traffic `D`, expression 4.27 represents the connection between the variables `V` and the indexes `I`, showing that the variables should only take values from `I`, as well as the solution to the problem, the first n^{th} elements of `V`.

Constraints

With Adaptive Search, the constraints have the purpose of indicating an error value, the amount of error that the variables to which the constraint have been applied are violating considering the purpose of the constraint, and are used to drive a heuristic search to for the solution. The implementation of the constraints consists in creating

Listing 19 Variables in Adaptive Search

$$D \ = \ \{(D_{(1,1)},\ldots,D_{(1,22)}),\ldots,(D_{(n,1)},\ldots,D_{(n,22)}),\ldots,(D_{(m,1)},\ldots,D_{(m,22)})\} \quad (4.23)$$

$$I = \{I_1,\ldots,I_n,\ldots,I_m\} \quad (4.24)$$

$$V = \{V_1,\ldots,V_n,\ldots,V_m\} \quad (4.25)$$

$$\forall I_i \in I,\ \ \forall D_i \in D,\ \ D_i = (D_{(i,1)},\ldots,D_{(i,22)}) \ \ \Rightarrow I_i \Leftrightarrow D_i \quad (4.26)$$

$$\forall V_i \in V,\ \ \Rightarrow V_i \in I,\ \ solution = \{V_1,\ldots,V_n\} \quad (4.27)$$

functions which quantifies the error or the number or violations, guiding the search algorithm towards a solution.

Adaptive Search relies on several heuristic functions, two of the most important being:

1. `Cost_Of_Solution` which computes the amount of error of a candidate solution, informing the algorithm how far away it is from a valid solution;
2. `Cost_On_Variable` which informs the algorithm of the cost of changing a variable in a candidate solution.

These heuristics rely on the error of the constraints applied to the variables of the problem in order to model the desired network situation.

The constraints used to model the problem need to access the variables used to model the CSP and the network traffic data, since they are applied to the variables, but rely in the network traffic to compute the associated errors.

Each constraint applied to a set of variables specifies a set of rules that must be verified in order for the constraint to be satisfied. These rules are then checked against the packets corresponding to the indexes assigned to the variables of the tentative solution and which are involved in the constraint, thus computing the error of each variable and constraint.

The network packet payload constraints are treated exactly like the others, the only difference is that rules which makes the constraint valid are checked against the *payload* of the packet indexed by variables of the current candidate solution.

Calculating the error of a constraint

Modeling a problem with Adaptive Search entails specifying a set of variables and a set of constraints. Associated to each constraint, there is an error function which calculates the number of rules being violated by the constraint. Each constraint is composed of a set of combined rules that must be checked. Each of these is specific to each constraint, applied to a set of network packet variables and the values found on a tentative solution are checked against the existing packets on the network traffic.

To calculate the constraint error, each of the rules that compose the constraint is individually checked and accounted for, with a different *weight*, depending on the relevance of the rule in the constraint. The total error of a constraint is given by the weighted sum of the number of rules violated by the current tentative solution.

As an example, a constraint that forces a network packet to be a *SYN* packet, can be defined as in Listing 20. In this simple constraint there is only one rule, stating that the network packet associated to the variable `var` must be a network packet with the `SYN` flag set, meaning that a particular field must have the value of `1`.

Line 2, checks if the corresponding packet in the network traffic window has the correct value on that specific field. If the value is correct, then the variable `var` doesn't violate this constraint and the function returns the value `0`(zero), otherwise, the function returns an error that reflects the violation of this constraint.

The return value depends on how relevant the constraint is to the problem, higher values being for less important constraints, lower for higher. The most important being the ones which state more restrictions over the variables.

This approach is used to calculate the error of a constraint, the cost of a solution and the associated error of a variable.

Listing 20 SYN Packet constraint

```
1 int syn (int window [][22], int var) {
2   if (window[var][14] == 1)
3     return 0;
4   else
5     return 1000;
6 }
```

Calculating the `Cost_Of_Solution`

A critical Adaptive Search heuristics is the `cost_of_solution` function, which computes the cost of a tentative solution, indicating how far it is from a valid solution, so it can decide the value that will be assigned to the *bad* variable of the current tentative solution, leading to the next best candidate solution.

The `cost_of_solution` is implemented by applying the necessary constraints over specific network packet variables in order to model the desired Network Intrusion Detection problem. Each constraint will return an error value which indicates the amount of error of the current tentative solution for the given constraint. The computation of the `cost_of_solution` is then achieved by adding the error values of all constraints applied to the set of variables used to model the problem, thus reaching the total error of the given tentative solution.

Listing 21 demonstrates how to compute the `cost_of_solution` for a simple problem of identifying 10 `SYN` packets on the network traffic. In this example, the variable `sol` is an array with all variables of the model, representing the current tentative solution, and `syn`, the constraint listed in Listing 20, applied to all 10 variables in order to model this simple problem. When applying the constraint to all variables, `err` will accumulate all the violations of all variables, providing the total cost of the current tentative solution.

Listing 21 `Cost_Of_Solution - Set of SYN packets`

```
int Cost_Of_Solution (void) {
  int i;
  int err=0;
  for (i=0; i<10; i++)
    err += syn (window, sol[i]);
  return err;
}
```

Calculating the `Cost_On_Variable`

Adaptive Search needs to compute the error of each variable on a tentative solution, in order to choose a candidate for a swap, the one with higher cost value.

The error of each variable is calculated by using the associated error value of all constraints in which it occurs. To compute the cost of a single variable, every constraint applied to that particular variable is evaluated, and the sum is the variable cost.

Continuing our example of a simple situation composed of 10 *SYN* packets, the cost of a variable in the current tentative solution is in Listing 22, where `i` represents the variable with respect to which the cost is being calculated, and variable `window` represents the network traffic that will be used to check the rules of the constraint `syn`, defined in Listing 20.

Listing 22 `Cost_On_Variable` - `SYN` Packet

```
int Cost_On_Variable (int i) {
  return syn (window, sol[i]);
}
```

Modeling

The actual modeling of a Network Intrusion Detection problem in Adaptive Search is achieved by using the necessary heuristic functions representing the constraints to create the `Cost_Of_Solution` and `Cost_On_Variable`.

Listing 23 show how a simple problem can be modeled. This example is about finding two network packets, the first one should have its SYN flag set and with destination port 80. The second packet should have as source address, the destination address of the first network packet. Line 2 defines the `Cost_Of_Solution`, on Line 5 it is checked if the first network packet has its SYN flag set, Line 6 checks if its destination port is 80, and Line 7 checks the source address of the second network packet is equal to the destination address of the first packet. Line 23 returns the accumulated error of all constraints.

Line 12 of Listing 23 defines the `Cost_On_Variable`, from lines 15 to 19 are checked the errors of the first network packet, in case of the argument `i` is 0(zero). Line 12 checks the errors associated to the second network packet. In the end, the accumulated error for this variable, is returned in Line 23.

4.5.2 Improving the Adaptive Search performance

Adaptive Search is a very efficient algorithm to solve combinatorial problems, but, if not properly tuned it can perform poorly. In order to get the best performance, we need to tune the heuristic functions to help Adaptive Search reach the desired solution as fast as possible, otherwise, the solver can get *lost* in the process of searching for a valid solution to the problem, degrading its performance.

Besides the heuristic functions which affect the performance of Adaptive-Search, there are a number of internal configuration parameters that influence how the AS algorithm behaves.

Fine-tuning the heuristic functions

The main heuristic functions used by Adaptive Search are `Cost_Of_Solution` and `Cost_On_Variable`. The `Cost_Of_Solution` inform the algorithm how far away the

Listing 23 Problem modeling in Adaptive Search - code snippet

```
...
int Cost_Of_Solution (void) {
   int err=0;

   err+=syn(window, sol[0]);
   err+=dst_port(window, sol[0], 80);
   err+=equal_src_dst_port(sol[1], sol[0]);

   return(err);
}

int Cost_On_Variable (int i) {
   int err=0;

   if(i==0){
      err+=syn(window, sol[0]);
      err+=dst_port(window, sol[0], 80);
      err+=equal_src_dst_port(sol[1], sol[0]);
   } else
   if(i==1)
      err+=equal_src_dst_port(sol[1], sol[0]);

   return(err);
}
...
```

current candidate solution is from a valid solution, as for the `Cost_On_Variable`, it informs the algorithm the cost of a variable with a given value assigned. These two heuristics are critical for the performance of Adaptive Search and require fine tuning.

These heuristics are built by using the constraints which model the specific Network Intrusion Detection problem, so, the way the constraints are modeled, more specifically the error value return by each constraint, has a great impact on Adaptive Search performance. One way to improve performance is to give more importance to some constraints than others, assigning a different *weight* to the constraints, depending on how important they are.

We experimented different error values for each constraint, to understand the behavior of Adaptive Search while using different weights on each constraint. As expected, these experiments revealed that the algorithm is quite sensitive to these changes.

These experiments also revealed that to improve the performance of Adaptive Search, the constraints which impose more restrictions on the solution, thus the more complex ones, also the most important constraints while modeling the problem, should have a lower error value. As for the auxiliary constraints, the ones which impose less restrictions on the solution, it was verified that they should have a higher error value in order to improve the Adaptive Search performance.

With these results, we decided to assign different error values according to the constraints, depending on the importance and complexity of the constraints: higher error values to the least important constraints, and lower error values to the most important ones.

By doing this, we are forcing the simplest constraints to be *solved* first, leaving the more complex ones for last, since Adaptive Search selects the variable with the higher error value in the current solution to compute the next candidate solution.

This approach to fine-tuning Adaptive Search turned out very good heuristic functions, making Adaptive Search faster to reach a valid solution, thus, making the AS detection mechanism of NeMODe very efficient.

We now list the error value for the most important constraints of NeMODe if the constraint is being violated, with a small description of each constraint:

- `syn(packet)` - `return 1000;`
  Force a network packet to have its `SYN` flag set.
- `reset(packet)` - `return 1000;`
  Force a network packet to have its `RESET` flag set.
- `ack(packet)` - `return 1000;`
  Force a network packet not to have its `ACK` flag set.
- `not_ack(packet)` - `return 1000;`
  Force a network packet not to have its `ACK` flag set.
- `dst_port(packet, port)` - `return 1000;`
  Force a the destination port of a network packet to be equal to `port`.
- `src_port_different(packet1, packet2)` - `return 500;`
  Force two network packets to have different source ports.
- `src_different(packet1, packet2)` - `return 500;`
  Force two network packets to have different source addresses.
- `different(packet1, packet2)` - `return 100;`
  Force two network packets to be different.

- `proceed(packet1, packet2)` - `return 2;`
  Force `packet1` to appear after `packet2`.
- `useconds_max(packet1, packet2, usecs)` - `return 1;`
  Force `packet1` to appear at-most `usecs` micro-seconds after `packet2`.

To obtain these return values, we tried several values for each constraint while experimenting the network situations presented in Sec. 5.5(page 109). The values which produce the best results were selected.

Fine-tuning the Adaptive Search parameters

Adaptive Search provides configuration parameters which allow to change its behavior, adapting the way it the algorithm searches for a solution, allowing to fine-tune specifically for each problem.

The following list presents the main configuration parameters of Adaptive Search as well as a brief description of each one, taken from the Adaptive-Search manual [92]:

- `ad_exhaustive`: if true, the solver always evaluates (the cost of) all possible swaps to chose the best swap. If false a projection of the error on each variable is used to first select the worst variable.
- `ad_prob_select_loc_min`: this is a percentage to force a local minimum (i.e. when the 2 selected variables to swap are the same) instead of staying on a plateau (a swap involves 2 different variables but the overall cost will remain the same). If a value > 100 is given, this option is not used.
- `ad_freeze_loc_min`: number of swaps a variable is frozen when a local minimum is encountered (i.e. the 2 variables to swap are the same).
- `ad_freeze_swap`: number of swaps the 2 variables that have been selected (and thus swapped) to improve the solution are frozen.
- `ad_reset_limit`: number of frozen variables before a reset is triggered.
- `ad_nb_var_to_reset`: number of variables to randomly reset.
- `ad_restart_limit`: maximum number of iterations before restarting from scratch (give a big number to avoid a restart).
- `ad_restart_max`: maximal number of restart to perform before giving up.
- `reset_percent`: percentage of variables to reset.

After some experiments these configuration parameters while solving some problems, we got to the conclusion that they affect the performance of Adaptive-Search in a great scale, as it would be expected. These experiments allowed us to find the values for each of these parameters which improve the performance of Adaptive Search in a great scale. The following list presents those values:

- `ad_freeze_loc_min = 8`: If a local minimum is found, freeze the variable during 8 swaps.
- `ad_freeze_swap = 5`: If two variables are swapped, they are frozen during 5 swaps.
- `ad_reset_limit = (NB_VAR / 4)+ 1`: If there are `(NB_VAR / 4)+ 1` frozen variables, where `NB_VAR` represents the number of network variables in the problem, a reset is triggered.
- `int reset_percent = 1`: When a reset is triggered, only `1%` of the variables are reset.
- `int ad_restart_max = 0`: Do not allow restarts.

By analyzing the values that we found to be better, we see that Adaptive Search should be configured in a way so it doesn't restart the solving process of a problem from scratch, it should minimize the number of variables to reset and should not should not allow the freeze of variables for a long time, forcing Adaptive Search to swap variables more often.

4.6 Modeling with SAT Solvers

SAT [79] problem consists on determining if there exists a valid assignment to all variables of a Boolean function so that the function is satisfiable, or, to determine that there is no valid assignment that can make such Boolean function **True**(a SAT problem), implying that the Boolean function is **False**(an UNSAT problem).

In order to solve a SAT problem, it is necessary to make a description of the problem as a Boolean function, composed by Boolean variables, which can only take **True** or **False** values. Usually this function is specified in the Conjunctive Normal Form(CNF) [79], a conjunction of clauses, where each clause is a disjunction of literals, and each literal is a Boolean variable or it's negation.

While modeling a Network Intrusion Detection problem as a SAT problem, the purpose of the Boolean function which describes the problem is a set of Boolean clauses representing the valid values for each variable, considering the constraints used to model the problem and the network traffic, so, the final purpose is to state which packets are a combination that satisfies the desired network situation.

4.6.1 Encoding a CSP with SAT

As previously mentioned in Sect. 3.4(see page 37), to model a problem with SAT, there is the need to create a number of CNF rules. This process is called *encoding* [80, 86, 79]. There are several ways to encode a CSP, the most commonly used are:

- Direct Encoding
- Support Encoding
- Log Encoding

In this work, we use both *direct* and *support* encodings, depending on the constraint being encoded. These can be used together, as they use the same variable representation and the same types of clauses.

In both encodings, the variables are represented in the same way: for each CSP variable i, and for each value d, there is a variable $V_{i,d}$ which represents the assignment of value d to CSP variable i.

These encodings are very similar, both composed by 3 types of clauses:

1. *at_least_one* clauses
2. *at_most_one* clauses
3. *constraint* clauses

Both encodings use the same type of *at_least_one* and *at_most_one* clauses. The *at_least_one* clauses state that at least one value must be assigned to each CSP variable, while the *at_most_one* clauses states that only one value can be assigned to a CSP variable.

The difference between these encodings is the way the constraints are specified. The *direct* encoding [79] uses *conflict* clauses, stating pairs of two inconsistent value assignments, indicating that when value d_j has been assigned to CSP variable i, value d_k cannot be assigned to CSP variable l.

As for the *support* encoding [79], it uses *support* clauses to represent the constraints. These are achieved by stating which values are compatible with a CSP variable, when a specific value has been assigned to a given CSP variable.

4.6.2 Modeling with MiniSat

To solve a SAT problem, MiniSat goes through two major steps; 1) read and parse the problem description, represented the Conjunctive Normal Form, in order to build the internal representation of the problem in MiniSat; and 2) the actual solving of the problem, already internally represented in MiniSat.

The step of reading and parsing the problem, which is normally read from text file with all CNF clauses, is very time consuming, having a great impact in the performance of MiniSat.

One of the main concerns of the MiniSat authors was to provide a tool that can easily be adapted and interfaced with other tools, making it appropriate as a back-end to NeMODe.

We used the MiniSat capability of programmatically specifying the CNF rules, thereby bypassing the text file parsing step.

Variables

Modeling a problem in SAT is quite different from doing so in any other approach to constraint programming, as we may only use `Boolean` variables. Encoding a CSP as SAT involve two types of variables: 1) the CSP variables; and 2) the variables which represent the assignment of each possible value to each CSP variable [86].

In a Network Intrusion Detection problem, each CSP variable represents a network packet, and then, there is a variable for each possible network packet that can be assigned to each SAT variable. We decided that a CSP variable represents an entire packet by indexing all packets on the network traffic, instead of having a CSP variable for each network packet field, thereby reducing the number of variables.

When encoding a network signature as a CSP problem, the first thing to be done is to create the variables which represent the assignment of each possible value to a CSP variable. For each CSP variable, representing a packet, there is a set of variables, one for each value that can be assigned to it.

Listing 24 represent the variables used to model a Network Intrusion Detection problem as a SAT problem, where D represents the set of network packets actually seen on the network traffic; each $D_j = \{F_{(j,1)}, F_{(j,2)}, \ldots, F_{(j,m)}\}$, a network packet actually seen in the network traffic, x the total number of network packets found in the network traffic; $F_{(j,i)}$ the field i of packet j, and m the total number of fields of a packet.

`P` is the set of all CSP variables, representing the network packet variables used to describe the desired network signature; and n the number of CSP variables.

`V` is the set of variables which represent the assignment of all possible values to each CSP variable P_i, where $V_i = \{V_{P_i,D_1},\ \ldots,\ V_{P_i,D_x}\}$ is the set of variables that represent all possible assignments that CSP variable P_i can take, i.e. V_{P_1,D_1} means that value D_1 was assigned to the SAT variable P_1, if set to true.

Listing 24 SAT variables

$$D = \{D_1, D_2, \ldots, D_x\} \tag{4.28}$$

$$\forall D_j \in D \ :\ D_j = \{F_{(j,1)}, F_{(j,2)}, \ldots, F_{(j,m)}\}, \tag{4.29}$$

$$P = \{P_1, P_2, \ldots, P_n\} \tag{4.30}$$

$$\begin{split} V = \{V_{P_1,D_1},\ \ldots,\ V_{P_1,D_x}, V_{P_2,D_1},\ \ldots,\ V_{P_2,D_x}, \ldots, \\ , \ldots,\ V_{P_n,D_1},\ \ldots,\ V_{P_n,D_x}\} \end{split} \tag{4.31}$$

A solution to such SAT problem will be a subset of V, containing only the variables that have been assigned with `True` values, the ones that are part of the solution, identifying the network packets that belong to the desired signature. Such solution is represented in Listing 25, where `S` is the set of all variables of V which has a `True` value, representing the solution of the problem.

Listing 25 SAT solution

$$\begin{split} \forall\ S = \{S_{P_1,D_{y1}}, S_{P_2,D_{y2}}\ \ldots,\ S_{P_n,D_{yn}}\} \Rightarrow S \subset V, \\ \forall\ \ S_j \in S,\ \ S_j = S_{P_j,D_{yj}},\ \ S_j = True \end{split} \tag{4.32}$$

Variable Domain

In a SAT problem, there is no concept of variable domain as in other constraint solving approaches, since the variables used to model the problem are Boolean variables.

In the encoding of a Network Intrusion Detection problem as a SAT problem, the constraints used to model the problem are also the ones responsible for ensuring that a valid solution makes sense on the given network traffic, and that the variables can only take values from the effective network traffic.

Constraints

Constraints in SAT encode the problem in CNF. The purpose of which is to model the valid values, according to the rules that should be verified by the constraint.

Encoding an intrusion signature as a SAT problem in CNF is quite complex and can grow very rapidly, due to the number of variables involved in a SAT problem. So, we created functions which model the necessary constraints for Network Intrusion Detection problems, and create the necessary CNF clauses, representing the rules of the constraint.

The modeling of a problem in SAT is then achieved by using such constraints in the necessary arrangement in order to achieve the desired description.

Two major types of encoding are necessary to model a problem in SAT:

1. encoding the set of variables, which ensures the integrity of the solution;
2. encodings which actually model the problem, which in turn model the desired intrusion signature, and the ones that are actually encoded by the constraints.

The first step of the encoding is almost independent of the network traffic and of the intrusion detection to be modeled, as it depends only on packets in the network traffic window, and the number of network packets used to model the desired signature. This step is almost static, and can be reused when those parameters are shared among problems, thus avoiding re-computation, resulting in a performance gain.

Variable encoding

The variable encoding, although almost independent of the problem being modeled, is very important. Although we decided to use a set of *indexing* variables to represent the network packets, the variable encoding specifies the domain of the variables, by specifying the set of rules which states that each variable has to take at least one value, an index to a network packet, and should at most one index to a network packet, thus, stating the domain of each variable.

Encoding the set of variables is made in two steps:

1. ensuring that at least one value is assigned to each SAT variable by using the *at_least_one* clauses;

2. ensuring that at most one value is assigned to each SAT variable, through the use of *at_ most_ one* clauses.

The *at_ least_ one* clauses are a set of clauses represented in Conjunctive Normal Form (CNF) stating that each SAT variable should take at least one value from any network packet on the network traffic log.

Listing 26 represents a formal representation of such clauses, where x represents the number of network packets in the network traffic and n the number of network packets used to model the intrusion signature.

If V_{P_i,D_j} is assigned with value `True`, means that packet j from the network traffic has been assigned to CSP variable i.

Listing 26 *at_ least_ one* clauses - formal description

$$\bigwedge_{i=1}^{n} \left(\bigvee_{j=1}^{x} (\; V_{P_i,D_j} \;) \right)$$

Listing 27 presents the *at_ least_ one* clauses to model a small network CSP made of 2 variables, while considering a network traffic of 2 network packets. D is the network traffic, where each D_i represent a packet; P the set of CSP variables; V the set of variables which represent the possible value assignments to each CSP variable, where each V_{P_i,D_j} represents the assignment of packet j to CSP variable i.

Listing 27 *at_ least_ one clauses* - example

$$D = \{D_1, D_2\} \tag{4.33}$$

$$P = \{P_1, P_2\} \tag{4.34}$$

$$V = \{V_{P_1,D_1}, V_{P_1,D_2}, V_{P_2,D_1}, V_{P_2,D_2}\} \tag{4.35}$$

$$\text{at\_least\_one\_clauses} \quad = \quad \big(V_{P_1,D_1} \;\vee\; V_{P_1,D_2}\big) \bigwedge \big(V_{P_2,D_1} \;\vee\; V_{P_2,D_2}\big) \tag{4.36}$$

The *at_ most_ one* clauses ensures that only one value is assigned to a SAT variable, which is accomplished by creating *conflict causes* between all combinations of possible values that can be assigned to a single variable, e.g. $\neg V_{P_1,D_1} \vee \neg V_{P_1,D_2}$, means that when value D_1 is assigned to variable P_1, the network packet represented by D_2 can not be assigned to the same variable P_1.

Listing 28 presents a formal representation of the *at_ most_ one clauses*, where x represents the number of network packets in the network traffic and n the number of network packets used to model intrusion signature.

Listing 28 *at_most_one clauses* - formal description

$$\bigwedge_{i=1}^{n} \Bigg(\bigwedge_{j=1}^{x} \Bigg(\bigwedge_{k=j+1}^{x} \big(\neg V_{P_i,D_j} \vee \neg V_{P_i,D_k} \big) \Bigg) \Bigg)$$

Listing 29 presents the *at_ most_ one_ clauses* for the example presented above, which consists of a CSP with 2 variables, and a network traffic made of 2 network packets.

Listing 29 *at_most_one clauses* - example

$$D = \{D_1, D_2\} \tag{4.37}$$

$$P = \{P_1, P_2\} \tag{4.38}$$

$$V = \{V_{P_1,D_1}, V_{P_1,D_2}, V_{P_2,D_1}, V_{P_2,D_2}\} \tag{4.39}$$

$$\text{at\_most\_one\_clauses} \quad = \quad \big(\neg V_{P_1,D_1} \ \vee \ \neg V_{P_1,D_2}\big) \ \wedge \ \big(\neg V_{P_2,D_1} \ \vee \ \neg V_{P_2,D_2}\big) \tag{4.40}$$

Due to the high complexity of the CNF rules and size these rules can reach, we created functions which, depending on parameters such as the number of network packets in the network traffic and the number of variables necessary to model the problem, generate the necessary rules to encode the problem. These functions are executed by the MiniSat solver, which generates the necessary rules and updates the MiniSat database with the newly created rules.

Constraint encoding

The second part of encoding a network signature as a SAT problem is the encoding of the problem itself: the constraints that compose the network signature and which actually model the problem. This encoding follows the same approach used to encode the variables, relying on *conflict causes* and *support clauses* to describe each constraint.

Each constraint is also modeled in a function, which creates the necessary CNF rules equivalent to the constraint being true. Using such functions, we can encode the desired Network Intrusion Detection problem in SAT, hiding the complexity of the CNF rules.

Constraints are encoded by analyzing the network traffic and, based on that traffic and the desired constraints, CNF rules are created by stating which packets are compatible with each other, according to the constraint being encoded.

Next, we present the modeling of the most important constraints as a SAT problem:

`packet_field_equal_to(packet, field, value)`

This constraint allows one to require a given field of a specific network packet to be equal to some value. This type of constraint is useful in many situations, e.g. requiring that the `destination port` of a network packet be equal to some port number.

The encoding of this constraint is accomplished by analyzing the network traffic and testing if the desired field on each network packet is equal to the desired value, and then, create a *disjunctive clause* with the variables representing such values.

Listing 30 presents a formal representation of the necessary clauses to encode this constraint, where `p` is the packet number to which the constraint is applied; `f` is the field of the packet to which the constraint is applied; and `v` the value that should be found in the field `f` of the packet number `p`.

Listing 30 `packet_field_equal_to(p, f, v)`

$$\forall\ packet\_field\_equal\_to(p,f,v),\quad 1 \leq j \leq x,\quad F_{(j,f)} = v :$$
$$clauses = \bigvee(\ V_{p,D_j}\ )$$

`field_equal_field(p1,f1,p2,f2)`

The constraint `field_equal_field(p1, f1, p2, f2)` allows one to force a field of a packet to be equal to another field of another packet, allowing the statement of relations between two network packets, e.g. the source port of one packet to be equal to the destination port of another network packet.

The encoding of this constraint is accomplished by analyzing the network traffic log and creating a set of *support clauses* describing which variables are compatible between them in order to satisfy this constraint. This set of variables is computed by verifying which network packets are compatible with each other according to the specifications of the constraint, by comparing the desired fields of each network packet.

Listing 31 presents the formal description of the *support clauses* used to encode this constraint, where `f1` represents the field of packet `p1` which should be equal to field `f2` of packet number `p2`.

Listing 31 `field_equal_field(p1, f1, p2, f2)`

$$\forall\ field\_equal\_field(p1, f1, p2, f2),\ F_{(p1,f1)} = F_{(p2,f2)} :$$
$$clauses = \bigwedge \left(\left(\neg V_{p1,D_1} \bigvee \left(V_{p2,D_2}\right)\right) \bigwedge \left(\neg V_{p2,D_2} \bigvee \left(V_{p1,D_1}\right)\right) \right)$$

`packet_different_packet(p1,p2)`

This constraint assures that the network packet variables `p1` and `p2` are assigned with different network packets, assuring that `p1` and `p2` are not the same.

This constraint is encoded by using *support clauses*, stating that when a specific network packet is assigned to a given variable, all other network packets, except the one that was assigned, are compatible with it. We use the network packet identification number to distinguish the different packets.

Listing 32 presents a formal description of the *support clauses* used to encode this constraint, where `p1` and `p2` are the network packets that must be different. $F_{j,12}$ represents the 12th field of packet j containing a unique identification number of the network packet.

Listing 32 `packet_different_packet(p1,p2)`

$$\forall\ packet\_different\_packet(p1, p2),\ F_{j,12} \neq F_{k,12} :$$
$$clauses = \bigwedge_{j=0,k=0,j\neq k}^{x} \left(\left(\neg V_{p1,D_j} \bigvee \left(V_{p2,D_k}\right)\right) \bigwedge \left(\neg V_{p2,D_j} \bigvee \left(V_{p1,D_k}\right)\right) \right)$$

`time_packet_greater_time_packet(p1, p2)`

This constraint is used to force the order between two network packets, ensuring that the network packet variable `p2` appears after `p1`, in a chronological order.

To encode this constraint, we used *support clauses* which states the compatibilities between each network packet of the network traffic window, regarding the temporal order between the two network packets, looking at the timestamps of network packets found on the network traffic.

More specifically, to encode the temporal constraints, for each packet found in the network traffic, we analyse the entire network traffic, looking for compatible packets which satisfy the temporal constraint, using the arithmetic expression found in Listing 33, taking into account the seconds and microseconds of the packets timestamp. This results in a set of packets which are compatible with each other, regarding the temporal restrictions. Then, based on these packets, a set of *support clauses* is created to encode the constraint.

Listing 33 presents a formal description of the *support clauses* used to encode this constraint, where `p1` and `p2` are the packets to which the constraint is to be applied. $F_{j,0}$ represents the 1^{st} field of network packet j, the time stamp of the packet j in seconds; and $F_{j,1}$ represents the 2^{nd} field of network packet j, the micro-seconds of the network packet time stamp. Expression $F_{j,0} * 10^6 + F_{j,1}$ represents the time stamp of network packet j, converted into micro-seconds.

Listing 33 `time_packet_greater_time_packet(p1, p2)`

$$\forall\, time\_packet\_greater\_time\_packet(p1, p2),$$

$$F_{j,0} * 10^6 + F_{j,1} < F_{k,0} * 10^6 + F_{k,1} :$$

$$clauses = \bigwedge_{j=0,k=0,j\neq k}^{x} \left(\left(\neg V_{p_1,D_j} \bigvee \left(V_{p_2,D_k} \right) \right) \bigwedge \left(\neg V_{p_2,D_k} \bigvee \left(V_{p_1,D_j} \right) \right) \right)$$

`usecs_packet_greater_packet(p1,p2,usecs)`

This constraint is used to make sure the network packet `p2` appears `usecs` microseconds after packet `p1`. To encode this constraint we use *support clauses* to state which packets are compatibles with each other by analyzing the timestamps of each network packet and calculating their temporal distance.

The approach to encode the `time_packet_greater_time_packet(p1, p2)` was also used in this temporal constraint. The only difference is the formula which calculates the compatible packets, which computes the packets within a range of *usecs* microseconds. The resulting packets are then used to create the *support clauses*.

Listing 34 `usecs_packet_greater_packet(p1,p2,usecs)`

$$\forall\ usecs\_packet\_greater\_packet(p1, p2, usecs),$$
$$F_{j,0} * 10^6 + F_{j,1} < F_{k,0} * 10^6 + F_{k,1} + usecs :$$
$$clauses = \bigwedge_{j=0,k=0,j\neq k}^{x} \left(\left(\neg V_{P_1,D_j} \bigvee \left(V_{P_2,D_k}\right)\right) \bigwedge \left(\neg V_{P_2,D_k} \bigvee \left(V_{P_1,D_j}\right)\right) \right)$$

Listing 34 presents a formal description of the *support clauses* used to encode this constraint, where `p2` is the network packet that should appear at most `usecs` microseconds after packet `p1`. $F_{j,0}$ represents the 1^{st} field of network packet j, the time stamp of the packet j in seconds; and $F_{j,1}$ represents the 2^{nd} field of network packet j, the micro-seconds of the network packet time stamp. Expression $F_{j,0} * 10^6 + F_{j,1}$ represents the time stamp of network packet j, converted into micro-seconds.

Modeling a problem

To model an IDS signature with SAT, we first apply the previously defined functions which encode and define the all necessary variables. Then we state the constraints which actually describe the given network signature, using the functions which implement them.

Listing 35 exemplifies the modeling of a simple situation in MiniSat: First, we start by encoding the variables in line 6 by using the function `setup(S, n_vars, n_packets)`, which takes as parameters the MiniSat solver `S`, the number of variables necessary to model the problem `n_vars`, and the number of packets in the network traffic `n_packets`. In line 9 we state that `packet 0` should have its SYN flag set, by using the constraint represented in the function `pkt_field_equal`, which states that field `14` of packet `0` should have a value of `1`, which means that it has the SYN flag set. Line 12 states that `packet 0` should have the destination port `80`, by stating that field `11`, representing the destination port, should have the value `80`. From line 15 to line 18 it is stated that the source address of `packet 1` should be equal to the destination address of `packet 2`, which is achieved by using the constraint represented in the function `pkt_field_equal_pkt_field`, which states that a particular field of one packet must be equal to other field of other packet, which in this case are the fields representing the source address of `packet 1` and destination address of packet `0`.

Listing 35 problem modeling - code snippet

```
...
int n_vars=2;
int n_packets=400;

//variable encoding
setup(S, n_vars, n_packets);

//packet 0 should have a SYN flag
pkt_field_equal(S, n_vars, n_packets, 0, 20, 1)

//packet 0 should have the destination port = 80
pkt_field_equal(S, n_vars, n_packets, 0, 13, 80)

//packet 1 source address = packet 0 destination address
pkt_field_equal_pkt_field(S, n_vars, n_packets, 1, 4, 0, 9);
pkt_field_equal_pkt_field(S, n_vars, n_packets, 1, 5, 0, 10);
pkt_field_equal_pkt_field(S, n_vars, n_packets, 1, 6, 0, 11);
pkt_field_equal_pkt_field(S, n_vars, n_packets, 1, 7, 0, 12);
...
```

4.7 Conclusion

In this Chapter, we described how we used the Constraint Programming paradigm, in some of its forms, to perform Network Intrusion Detection. We described the use of Propagation Based, Constraint-Based Local Search and Boolean Satisfiability solvers. We have demonstrated that we can model Network Intrusion Detection signatures using these methodologies, in order to perform the detection of signature-based attacks.

Each approach to Constraint Programming poses specific problems in modeling a Network Intrusion Detection problem. Constraint Based solvers allow a relatively easy modeling of the problem. On the other hand, it is quite difficult to efficiently restrict the domain of the network packet variables, so that a solution to the problem be composed exclusively of packets found in the actual traffic, without sacrificing performance.

Modeling in Constraint-Based Local Search is easy in the sense that the constraint specification is straightforward, but since AS is quite sensitive to the heuristic used to model the problem, it turns out difficult to figure out the best heuristics, thus, making the modeling of Network Intrusion Detection problem in Adaptive Search quite complex. In AS, ensuring that the domain of the network packet variables is correct comes for free, due to the way we modeled the network packet variables.

As for SAT, the modeling of the problem is quite linear after all constraint functions have been implemented in order to encode the necessary CNF rules. The major problem in SAT is the modeling of such constraint functions, which generate quite large and complex sets of Boolean rules. In MiniSat, due to the specificity of the SAT problems, we don't need to worry about the domain of the network packet variables, since it is taken care of by the encoding the of problem.

Chapter 5

A Domain-Specific Language for IDS

This Chapter starts with a brief introduction of Domain-Specific Languages and their main characteristics. Then, we describe the Domain-Specific Language provided by NeMODe which eases the description of the signatures of specific attacks, leading to the generation of executable code for each of the detection mechanism available in NeMODe, from a single source. We also present the specification of the DSL, the code generation process for each detection back-end and some examples.

5.1 Domain-Specific Languages

Domain Specific Languages (DSLs) are programming languages specially adapted to a specific problem domain, as opposed to GPL, which are created with no specific domain in mind, trying to be as general as possible so they can be used in a wide variety of settings.

5.1.1 DSLs and GPLs

General-Purpose Languages can be large and complex in order to cope with a vast diversity of problem domains. Due to this, GPLs are indeed suitable to solve problems of virtually any domain problem.

The *generality* of these programming languages brings about two major concerns:

1. The problems have to be *programmed* using a *generic language*, identical for all problems, which, in some cases, may make the modeling of the problem awkward.

2. The applications written with such languages may be not as efficient as they could be, since they are implemented with generic tools.

On the other hand, Domain-Specific Languages, also known as micro languages or little languages, are small, focused on a particular domain, using very restricted notions and abstractions thus being much more expressive for modeling specific problems, many times being called definitions, specifications or descriptions [93]. Due to their high expressive power, DSLs are usually declarative, allowing to "program" the problem in terms of *what* should be achieved rather than *how* it should be achieved.

Since Domain-Specific Languages are custom built for a specific-domain, they can be very simple to use by users *fluent* on the application domain or by domain experts, allowing for a non-programmer user to be able to use, validate, modify or write DSL programs without difficulties [94].

Although Domain Specific Languages are mostly used by application-domain experts, scripting or macro languages found in spreadsheet-like applications, enabling simple programming tasks, are designed to be used by *normal* users.

DSL Development

Although very easy to use, DSLs are hard to develop, because, due to their specific application domain, it requires a multifaceted development team with a comprehensive knowledge of the domain and, at the same time, knowledge about programming language development, two capabilities which can be hard to combine.

Executability of DSLs

In General-Purpose Languages the programs are built with the intention of producing executable applications, this is not necessary true with DSLs, as these are often used to generate *inputs* to some other tools, such as *application generators* or *parser generators*.

Many times they rely on a well-defined execution semantics to produce some output, such as spreadsheets or HTML [93].

Applications such as YACC [95], a parser generator; $\TeX$ [96] a language to typeset text documents; PIC [95], a language for describing pictures are examples of DSLs which don't generate an executable application.

5.1.2 The origins of DSLs

DSLs have been used for programming applications for a long while, tools like APT [97], developed between 1957 and 1958, a language to program numerically controlled machine tools; BNF [98], introduced in 1959, the well-known syntax specification formalism can be thought of as early formalism of Domain-Specific Languages [93].

Among the oldest DSLs, we have other well known programming languages, such as Cobol, Fortran and Lisp. Although these languages are General Purpose Languages, they were originally designed for specific domains: Cobol for business computing, Fortran for numeric computation and Lisp for symbolic processing [99]. Over the years, these languages have evolved, slowly becoming larger, more complex and more amenable General Purpose programming.

Tools such as the UNIX tools `awk` [100] and `sed` [101], which have been around for years may also be considered as Domain Specific Languages. Modern widely used tools, such as spreadsheets, HTML, SQL, CSS, among others, are also DSLs [94].

Although Domain-Specific Languages have been used over years in a variety of application domains, the scientific and academic interest in these only came much later. The first studies regarding this subject are dated from 1985 when Martin [102] did an exhaustive account of Fourth-Generation Languages (4GLs); and from 1989 where Biggerstaff and Perlis [103] include a number of articles on software reuse including DSL development and program generation [93].

From those and other more recent studies, several definitions have been proposed, such as the one from Van Deursen et al. [99]:

> "A *domain-specific language* (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular domain."

5.1.3 DSL design

Domain-Specific Languages can be developed in a vast number of ways, but depending on the approach used to develop a DSL, two stages are mandatory and usually present in the process of implementing a Language:

1. Analysis
2. Implementation

Analysis

The analysis stage is the prime stage in the process of designing a DSL, considered by many authors a pre-requisite to start developing a DSL [104]. This stage may comprise several sub-stages:

1. The first of these sub-steps is identify the problem domain, which, while looking like an obvious step, is sometimes overlooked by developers.
2. After the problem domain has been identified, there is the need to collect relevant knowledge about the application domain. This knowledge is very important and should be mature enough before the development of any DSL in order to catch the correct *essence* of the application domain.

When the domain knowledge has been successfully collected, it needs to be processed to extract the semantic notions and operations, which will allow for a powerful and expressive way to write programs which in turn generate applications for the given application domain. With the semantic notions and operations defined, the DSL can be designed, without losing focus on the application domain and its semantics.

Implementation

The implementation phase is essential, resulting in a tool which receives as input a DSL program and transforms it into a domain specific output or even into executable applications. Several approaches can be taken to implement a DSL, but there are two important phases which are always present:

1. Domain-specific library design and implementation
2. Compiler design and implementation

As a first step, one should design and implemented a library which covers for all the semantic notions found during the initial analysis step.

After this library has been implemented, a compiler should be designed and implemented according to the DSL design specifications resulting from the analysis step. This compiler must be able to translate a DSL program into a sequence of calls to library functions, as previously defined.

5.1.4 Approaches to DSL implementation

Several approaches have been used over the years to satisfy the growing needs for specialized programming languages over specific application domains. The most widely used approaches since the first DSLs, and still valid in present days fit these two categories:

1. Internal languages
2. External languages

Internal languages

Internal languages, also known as embedded languages or Domain-specific embedded languagess (DSELs) [93], are *add-ons* to General-Purpose Languages, providing supplementary expressive power over some specific application domain. These extensions to GPLs include all the semantic notions which have been acquired in the design step of the language, usually implemented by means of a set of subroutine libraries, which implement the domain notions and operations.

This approach to DSLs does not provide the same freedom as an external languages, since the underlying constructs of the base GPL must be respected. On the other hand, this approach allows the use of the features found in the base GPL, at no extra cost.

When using this method to develop a Domain-Specific Language, several approaches can be adopted to extend a GPL, of which the most commons are:

1. domain-specific libraries
2. pre-processing or macro processing

When using **domain-specific libraries** to extend a General-Purpose Language, the developers make use of functions, procedures and other related tools available in the base GPL to implement the domain-specific operations. Using the methods available in the base GPL, the final DSL is very limited in terms of expressiveness, since it has to respect every single aspect of the base GPL, including its syntax. Since the domain-specific *addition* to the GPL is achieved by the creation of custom domain-specific functions, there is no need to make any changes in the compiler/interpreter, as those those additions are correct GPL programs, which turns out to be the main advantage of this approach.

The **pre-processing or macro processing** approach is very flexible, since it uses a set of macros which convert the domain-specific language constructs into GPL statements, thus not being restricted to the syntax and other characteristics of the base GPL. Although more flexible, this approach is not capable of error checking at the domain-specific level, making the identification of errors quite complex. Also, any optimization process has to be done at the GPL level.

External languages

External languages, also known as *micro-languages* or *little-languages* are a widely used method to implement Domain-Specific Languages. These external languages are small programming languages created from scratch, custom-built for an application domain, usually declarative and very expressive, and providing a compiler which generates applications or other form of output from programs written in the custom domain-specific language.

This approach to DSL building is most common, and follows the same methods which used to build General-Purpose Languages, using standard compiler construction tools. Sometimes one uses specifically tailored tools, to help design and implement Domain-Specific Languages. Section 5.1.5 gives a brief introductions to such systems.

This approach presents some significant advantages, allowing to design every single aspects of the language to the needs of the specific application domain, having no restrictions in terms of notions, primitives, variables and other language constructs. Among other advantages, there is also the possibility to include error detection and other language helpers at the domain level.

External languages also have some issues, mostly related to the cost of building a programming language from scratch together with the need of a strong knowledge in the development team about the specific domain. Another problem is the inability to re-use the language in other application domains.

5.1.5 DSL Development Systems

As previously noted, the development of a DSL is a difficult task, demanding a development team with a large knowledge and expertise in both the specific application domain and in programming language development.

To ease the development of DSLs, systems have been developed over the years with a wide variety of capabilities, in order to help developers to create DSLs: Draco [105], ASF+DSF [106], Kephera [107], Kodyack [108] or InfoWiz [109], just to mention a few.

While these *helper* systems vary in many aspects, they tend to use the same type of input to help building a DSL, specific information about the desired DSL, usually described by means a of specialized meta-language, usually rule-based. These aspects are normally related to syntax, pretty-printing, consistency checking, execution, translation and debugging [93].

These tools help with some aspects of DSL development, usually in the implementation step. In other aspects of the development of a DSL these systems are not adequate.

5.2 Requirements Overview

Using Constraint Programming to perform Network Intrusion Detection allows one to be very expressive and model intrusion signatures, which, sometimes are quite hard or even impossible to model in *standard* Network Intrusion Detection systems.

Despite of the expressiveness brought by using the Constraint Programming paradigm, the modeling of the desired network signature can be quite complex for someone not familiar with Constraint Programming. In order to ease the description of the network situations, we decided to create a declarative, intuitive Domain-Specific Language for Network Intrusion Detection [5], with network specific jargon expressions and terms to model the intrusion signatures, which talks about network entities, their properties and relations among them, allowing to describe network intrusion signatures, and, with based on those descriptions, generate intrusion detectors.

The key goal of this DSL is to ease the way in which network attack signatures are described, using constraint programming, hiding from the user all the constraint programing aspects and complexity of modeling network signatures as a Constraint Satisfaction Problem (CSP), but still, using the methodologies of Constraint Programming to describe the problem at a much higher level of abstraction, describing how the network entities should relate among themselves and what properties should hold.

Maintaining the declarativeness and expressiveness of the Constraint Programming (CP), allows an easy and intuitive way of describing the network attack signatures, by describing the properties that *must* or *must not* be seen on the individual network packets, as well as the relationships that should or should not exist between each of the network packets.

The primary goal of the Domain Specific Language provided by NeMODe (NEtwork MOnitoring DEclarative approach) is to ease the description of network intrusion signatures, but, equally important, to generate source code for each of the detection back-end mechanisms of NeMODe, from a single specification.

The DSL is a front-end compiler to several back-ends, one for each intrusion detection mechanism. This organization allows it to generate several recognizers based on different solver methods, from a single, unified description.

5.2.1 Hello world

As a simple example to introduce the DSL, we present a small network signature which looks for 10 TCP packets with the SYN flag set.

Listing 36 shows how this simple problem can be modeled in NeMODe. We start by naming the program in Line 1, then, in Line 2 we define the network traffic, as well the solvers which will be used.

In Lines 3-6 we create a variable `A`, which should be a TCP packet and should have its SYN flag set. These statements are then assigned to variable `P`.

Then, in Line 8, we clone the statements found in variable `P` 10 times, assigning the result to variable `C`. This creates 10 sets of statements like the ones found in `P`, but using distinct variables.

The statements assigned to `P`, and the *clone* assigned to `C` have no effect if not *used*. To do so, in Line 9 we *use* the *clone* `C`, which applies the statements previously assigned to it.

Finally, in Line 11 we specify the alert message to be shown when the signature is found.

Listing 36 Helloworld example

```
1   helloworld {
2     RES = solve('traffic.pcap', [as,gecode,minisat]) {
3        P = {
4           tcp_packet(A),
5           syn(A)
6        },
7
8        C := clone(10,P),
9        C
10  } => {
11     alert(Hello world')
12  };
```

5.2.2 Other approaches

Most widely used approaches to Intrusion Detection use specific rule-based languages to describe the network intrusions. Usually these systems do not allow the relation between several network packets, and when they do, they do it in a very limited way, usually resorting to plugins to achieve the desired relation.

As a simple example to compare the DSL of NeMODe with other approaches, in Listing 37 we present the description of a SS brute force attack in Snort which looks for 5 SSH connection attempts in a 60 seconds time interval.

In Sec. 7.4.5(see page 159) we further evaluate the DSL of NeMODe with Snort.

Listing 37 SSH password bruteforce Snort rule

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 22 \
(msg:"Possible SSH brute force attempt"; \
flow:to_server,established; \
threshold:type threshold, track by_src, count 5, seconds 60; \
content:"SSH-"; offset: 0; depth: 18;)
```

5.3 The NeMODe Specification

An input program written in the NeMODe DSL is composed of two major parts; 1) the description of the desired network attack signature; and 2) the actions to take when such a signature is detected on the network.

The description of the desired signature is composed by two main parts, as found on Line 1 of Listing 38:

1. An optional set of initial declarations.
2. The actual description of the desired signature.

The initial declarations are optional (line 3 of Listing 38), since they their purpose is to ease the description of the network signatures, by defining a set of variables representing IP addresses, host names, port numbers or services, which can later be used on the description of the problem, making the program more *readable*, by referring to hostnames and ports or services, instead of IP addresses of port numbers. These initial declarations are represented by a comma separated list of declarations (Line 5 of Listing 38).

Listing 38 NeMODe DSL simplified grammar - Program main structure

```
1    program → init case | case
2
3    init → decl_list
4
5    decl_list → declaration
6              | decl_list, declaration
7
8    declaration → ID = NUMBER
9                | ID = IP_ADDRESS
```

The main part of a NeMODe program is a network ***case***(line 1 of Listing 41), where the network situation to be modeled is actually modeled. A ***case*** is composed by two parts:

1. ***solver_ list***
2. ***action***

5.3.1 Signature description

The ***solver_ list*** (Line 3 of Listing 41) is the most important part of a NeMODe program, since this is where the desired intrusion signature is modeled. It also contains the identification of the tool which will be used to solve the problem. The ***action*** (Line 1 of Listing 57, page 102) is the action to take when the desired network situation is detected.

Although the current implementation of NeMODe accepts a description of what to do when the desired network situation is detected, it still does not produce any code to perform the specified action, since our primary goal is to describe and detect the signatures. In future implementations of NeMODe, it will be produced some code which that will make sure that the desired actions will be taken.

The ***solver_ list*** (Line 3 of Listing 41) is a simple comma separated of ***solver***, each one modeling different parts of the desired network signature. There are two types of ***solver*** (Line 8 of Listing 41) in order to distinguish two completely different filtering methods, the two types are:

1. ***filter***
2. ***solve***

The ***solve*** type is used to describe and solve complex network intrusion signatures and is what really uses a constraint programming approach to perform the detection of the network signature.

In a ***solve***, we can specify any type of statement in order to model the desired network situation (Line 8 of Listing 41).

Listing 39 presents a simple example of a *solver* of type *solve*, where we state the traffic to be analyzed and the list of solvers to be used. Then, we specify a set of statements which model a given signature. In this case we are looking for two network packets, `A` and `B`. `A` should have destination port 80, and `B` should have as destination port the source port of `A`.

Listing 39 *Solve* example

```
...
S = solve('traffic.pcap', [as,gecode,minisat]) {
  tcp_packet(A), dst_port(A) == 80,
  tcp_packet(B), dst_port(B) == src_port(A)
}
...
```

As for the ***filter***, is only used to perform some simple initial filtering tasks, accomplished by the use of simple packet analyzer tools, such as `tcpdump` [110], only allowing the specification of simple properties over the network packets, that can be specified with tools like `tcpdump`(Line 7 of Listing 41). The statements which can be used in a filter, are represented in (Line 17 of Listing 41).

Listing 40 presents a simple example of a *solver* of type *filter*. We start by stating the traffic to be analysed as well as the tool which will be used to filter the network traffic. In this example, we are filtering TCP packets which have the source or destination port 80.

Listing 40 *Filter* example

```
...
F = filter('traffic.pcap', [tcpdump]) {
  tcp_packet(A), port(A)=80
}
...
```

A ***solver***(Line 8 of Listing 41), independently of its type, is composed by 3 parts:

1. network traffic source
2. a list of filtering tools
3. network signature description

The network traffic source is the identification of the network traffic which will be used analyzed to look for the desired network signature. The filtering tools indicate which solver/filter will be used to filter the network traffic source and detect the desired network signature. Finally, the network signature description models the desired network attack signature.

A *solver* is *stored* into a variable, *ID*, which can later be used as an input to another filtering stage.

Listing 41 NeMODe DSL simplified grammar - a ***case***

```
case → ID { solver_list } => { action };

solver_list → solver
            | solver_list , solver

solver →
    ID = filter ( STRING , [ id_list ] ) { stmt_filter_list }
  | ID = solve ( STRING , [ id_list ] ) { stmt_list }

stmt_list → stmt
          | stmt_list , stmt

stmt → primitive | connective
     | ID  = { stmt_list } | ID
     | macro_stmt | logic_stmt

stmt_filter_list → stmt_filter
                 | stmt_filter_list , stmt_filter

stmt_filter → primitive_type ( var  )
            | data ( var , NUMBER ) == STRING
            | address eq_op ID | address eq_op ip_address
            | port eq_op NUMBER |  port eq_op ID
            | src_dst_port eq_op NUMBER | src_dst_port eq_op ID

id_list → ID
        | id_list , ID
```

Listing 42 presents an example on how to use the result of a *solver* as input to another *solver*. In this example we use a *filter* which uses `tcpdump` to filter the network traffic, looking for packets with source or destination ports 80. The *filter* is assigned to `F`, which is then used as input for the *solver* of type *solve*.

Then, using `F` as network traffic, the *solve* looks for 2 TCP packets `A` and `B` with specific properties, the destination port of `B` should be the same as the source port of `A`.

Listing 42 Using a *solver* as input to another *solver* - example

```
...
F = filter('traffic.pcap', [tcpdump]) {
  tcp_packet(A), port(A)=80
}

S = solve(F, [as,gecode,minisat]) {
  tcp_packet(A), tcp_packet(B),
  dst_port(B) == src_port(A)
}
...
```

The most important part of a NeMODe program is the list of statements, *stmt_ list* (line 1 of Listing 41), where the network signature is actually described.

NeMODe DSL provides 6 types of statements(line 13 of Listing 41):

1. ***primitive*** statements
2. ***connective*** statements
3. ***definition*** statements
4. ***use*** statements
5. ***macro*** statements
6. ***logical*** statements

Primitive **statements**

The *primitive* statements (Listing 43) are the simplest statements available in NeMODe, the ones which state simple properties that must be verified by the network packets, forcing some specific properties of a network packets to hold true.

More specifically, the primitive statements available in NeMODe allows to require a network packet to:

1. be a tcp, udp or arp packet;
2. have any of its tcp flags set;
3. not to acknowledge another tcp packet;
4. have a specific data on its payload;
5. have a specific source or destination address;
6. have a specific source or destination port;

Listing 43 NeMODe simplified grammar - primitive statements

```
primitive → primitive_type ( var  )
          | data ( var  ) ~= STRING
          | data ( var , NUMBER ) == STRING
          | address eq_op ID | address eq_op ip_address
          | port eq_op NUMBER |  port eq_op ID
          | src_dst_port eq_op NUMBER | src_dst_port eq_op ID

primitive_type → tcp_packet | udp_packet | arp_packet
               | arp_reply  | arp_query
               | urg | ack | psh
               | rst | syn | fin | nak
```

Listing 44 presents a simple example using 2 *primitive* statements, which states that we are looking for a TCP packet `A`, which should have the destination port 80.

Listing 44 *Primitive* statements - example

```
 ...
 tcp_packet(A), dst_port(A)==80
 ...
```

Connective statements

The *connective* statements (Listing 45) are important because they describe network situations that spread across several network packets. The *connective* statements have the purpose of enforcing relations between two network packets, more specifically, they allow to force:

1. a tcp packet to acknowledge another tcp packet;
2. a destination/source port of a packet to be equal/different to another destination/source port of other packet;

3. a destination/source address to be equal/different to a destination/source address of other packet;
4. the payload of two network packets to be equal/different at specific positions;
5. two network packets to have a temporal relation, such as their temporal distance to be inferior to a given amount of time.

Listing 45 NeMODe simplified grammar - connective statement

```
connective → ack ( var ) eq_op var
   | src_dst_port eq_op src_dst_port
   | address eq_op address
   | time rel_op time
   | data(var, NUMBER, NUMBER) eq_op data(var, NUMBER, NUMBER)
```

Listing 46 presents a small example which use *connective* statements, more specifically it states that source port of TCP packet `A` should be equal to destination port of packet `B`.

Listing 46 *Connective* statements - example

```
  ...
  tcp_packet(A), tcp_packet(B),
  src_port(A) == dst_port(B)
  ...
```

The *primitive* and *connective* can be used to describe most of the network intrusion signatures we used , but the NeMODe DSL provides some more statements types to help the description of such signatures, the *definition* statements, the *use* statements and the *macro* statements.

Definition **statements**

The *definition* statements (Line 2 of Listing 47) allows to define a variable as a group of statements, which can later be used in the description of a network situation. This type of statements have no effect on the program unless they are used latter, since they are only the definition of a variable.

This can also be understood as a method to define macros, since we are assigning a set of statements to a given name, which can later be used.

Listing 48 shows an example where two statements are used two define variable `C`.

Listing 47 NeMODe simplified grammar - definition statement

```
1 stmt → primitive | connective
2        | ID  = { stmt_list } | ID
3        | macro_stmt | logic_stmt
```

Listing 48 ***definition*** example

```
...
C = {
      tcp_packet(A),  syn(A)
}
...
```

Use statements

The *definition* statements have no effect, unless they are actually used, to do so, there is the *use* statement (Line 2 of Listing 47), allowing to use a previous *definition*. This activates the *definition* which produces the desired effect.

Listing 49 presents a simple example demonstrating the *use* of a previous *definition*. Considering `C` a *definition*, created in Line 2, we *use* `C` in Line 6. This will apply the statements assigned to `C`.

Listing 49 ***use*** example

```
1 ...
2 C = {
3      tcp_packet(A), dst_port(A)==80
4 },
5
6 C
7 ...
```

Macro statements

To ease the description of some properties that should be verified, we created the *macro* statements (Listing 50), designed to avoid the unnecessary repetition of code.

The *clone* statement (Line 7 of Listing 50) is one of the *macro* statements, which allows to *clone* a previously defined variable a given number of times. These *clones* are *stored* under a variable, e.g. `C := clone(3,P)`, which allows to later state constraints over a specific variable of a specific *iteration* of the *clone*. Also, when we the *clone* statement is used, it implicitly states that the variables of each instance of the *cloning* should be different.

Listing 50 NeMODe simplified grammar - The *macro* statements

```
1  macro_stmt → ID := clone
2             | same_different
3             | interval  ( var ) eq_op time
4             | duration ( var ) eq_op time
5             | connection ( var , var )
6
7  clone → clone ( NUMBER , var )
8
9  same_different →
10              same_src ( var )
11            | same_dst ( var )
12            | same_src_port ( var )
13            | same_dst_port ( var )
14            | different_src ( var )
15            | different_dst ( var )
16            | different_src_port ( var )
17            | different_dst_port ( var )
18
```

The *same_ different* statements include a set of statements which are used to state the same properties over all a specific variable in all instances of a *cloning*, e.g. `same_src(C:A)`, which forces all variables `A` of all instances of cloning `C` to have the same source address.

The *macro* statement *duration* (Line 4 of Listing 50) forces the overall duration of a *cloning* to a be higher or lower than a certain amount of time specified in seconds or micro-seconds, e.g. `duration(R) < secs(60)`.

The *macro* statement *interval* (Line 4 of Listing 50) forces the time between each *iteration* of a *cloning* to be higher or lower than a given amount of time, specified in seconds or micro-seconds, e.g. `interval(R) < secs(60)`.

As for the *macro* statement, *connection* (Line 5 of Listing 50), it forces two packets to belong to the same connection, forcing the source or destination of a specific packet be the same as the destination or source of another network packet.

Listing 51 shows an example usage of the *duration* statement; where Lines 1-7 defines a variable `C` as a block of statements, Line 6 defines a *cloning*, where `C` is cloned 3 times and stored in variable `R`. Line 7 states that the overall duration of `R` should be less than 60 seconds.

Listing 51 Example of a `macro` statement

```
C = {
      tcp_packet(A),
      syn(A)
    },

R:=clone(3,C),
duration(R) < secs(60)
```

Logical statements

The last type of statements, but also as important, are the *logical* statements (Listing 52). These statements have the purpose of allowing the specification of some logic operations over *primitive* and *connective* statements, such as in Listing 53. In this example, we stated that the source address of packet `A` should be equal to the destination of packet `B` *or* equal to the source address of packet `B`.

Listing 52 NeMODe simplified grammar - Logic statements

```
logic_stmt → logic_stmt logic_op logic_stmt
           | ( logic_stmt )
           | primitive
           | connective
```

Listing 53 Example: Logical statements

```
src(A)==dst(B) | src(A)==src(B)
```

Basic Network Entities

Essential to NeMODe are the basic network-related entities that can be used to describe network intrusions. Listing 54 presents the basic entities which are available to the DSL and are essential to describe a network situation, such as ports, ip address and time, used in several types of statements.

Temporal expressions

Since NeMODe allows the relation of network packets regarding their timestamps, one of the basic network entities is *time* (Line 7-12 of Listing 54). One can use simple *time units*, `usecs` and `secs`; or arithmetic expressions involving timestamps of several packets.

Listing 54 NeMODe simplified grammar - Basic entities

```
src_dst_port → dst_port ( var )
             | src_port ( var )

address → src ( var )
        | dst ( var )

time → usecs ( NUMBER )
     | secs ( NUMBER )
     | time_arith

time_arith → time ( var )
           | time ( var ) arith_op time_arith
```

Listing 55 presents a simple example which demonstrates the use of temporal expressions. We state that the time difference between TCP packets `B` and `A` should be less than 5 microseconds.

Listing 55 Temporal expressions - example

```
     tcp_packet(A), tcp_packet(B),
     time(B) - time(A) < usecs(5)
```

Basic Operators

Listing 56 NeMODe simplified grammar - Basic operators

```
eq_op → ==
      | !=

logic_op → ‘‘|’’
         | &

arith_op → +
         | -
         | *
```

Listing 56 presents the most relevant operators found in NeMODe: the *equality* and *disequality* operators `eq_op`; the *logical* operators `logic_op` and the *arithmetic* operators `arith_op`.

5.3.2 Actions

A *case* is composed by two parts, the *solver_ list*, which actually describes the desired network situation, and the *action*(Line 1 of Listing 57), which describes the action to take when the specified intrusion is detected.

In the current implementation of NeMODe, there is only one possible action, alert for an attack. The *alert* takes a list of strings and variables as arguments, which are concatenated to build the alert message. When using a variable as argument, the source address is used to generate the alert message.

Listing 57 NeMODe simplified grammar - Action statements

```
1 action → alert ( alert_arg_list )
2
3 alert_arg_list → alert_arg
4                | alert_arg_list , alert_arg
5
6 alert_arg → var | STRING
```

Listing 58 presents an example of an `alert` using 2 arguments, a string and variable `A`. This produces an alert message which is the concatenation of the string and the source address of TCP packet `A`.

Listing 58 Action - example

```
http_access {
   R = solve('traffic.pcap', [as]){
     ...
     tcp_packet(A), dst_port(A)==80
     ...
   }
} => {
   alert(``http access from '', R.A)
}
```

5.3.3 Variables

Variables in a NeMODe program (Listing 59) are designed to help the description of signatures, always upper case to be easily identified.

The variables can occur in several environments and situations, allowing us to categorize them in the following types:

1. initial *declaration* variables
2. *solver/filter* variables
3. *definition* variables
4. *cloning* variables
5. network packet variables

***Declaration* variables** The initial *declaration* variables are the ones used to declare host names, services or ports, as described earlier in this Chapter(Line 8 of Listing 38).

***Solver/Filter* variables** The *solver/filter* variables are the ones where we *store* the result of a solver or filter, which can then be used as the input to other solver or filter.

***Definition* variables** The *definition* variables are the ones used to *store* a set of statements, which will be used later, such as in a *cloning.*

***Cloning* variables** The *cloning* variables is where a *cloning* of a *definition* is *stored*, allowing it be accessible later, as well as its internal variables.

***Packet* variables** The network packet variables, as the name indicates, are the ones that represent the network packet, and to which the constraints are applied.

Although NeMODe provides these types of variables, they can be represented formally as in Listing 59. Most variables are represented by `IDs`(Line 1 of Listing 59), uppercase, alpha numeric, including the `'_'` character and always starting with a letter, as in Listing 60.

There are two exceptions to this representation, the variables of a clone (Line 5 of Listing 59), and the variables of a solver or filter (Line 7) of Listing 59), which can be accessed from an outer scope. These variables are variations of the *regular* `IDs`, which are arranged with other `IDs` using a special nomenclature, which respect and self-describe the scope of the variable.

To access a variable inside a clone, we first refer to the *clone*, then the *iteration* number and finally the variable, e.g. `C[2].A`, where `C` is the *clone*, `2` is the desired iteration of the clone, and `A` the variable.

Accessing a variable inside a *solver* is made in a similar way, but referring first to the *solver*, e.g. `S.C[2].A`, where `S` is the name of the desired solver. In Sec. 5.3.4 we further detail these types of variable.

Listing 59 NeMODe simplified grammar - Variables

```
var → ID
     | clone_var
     | filter_var

clone_var → ID [ NUMBER ] : ID

filter_var → ID . ID
           | ID . clone_var
```

Listing 60 `IDs` regular expression

```
identifier      [A-Z]([A-Z_]|[0-9])*
```

Except for network packets, variables are implicitly declared, being defined the first time they are referenced or used. As for the network packet variables, they are declared explicitly by the use of `tcp_packet(x)`, `udp_packet(x)` or `arp_packet(x)`, where `x` is the variable name.

5.3.4 Variable scope

A program in NeMODe contain several scopes, a first one which is the program itself, then, a second scope for the *solvers/filters*, and, inside each solver there might exist a third scope, a definition or even the clone of a definition. Figure 5.1 represents the variable scopes in a NeMODe program.

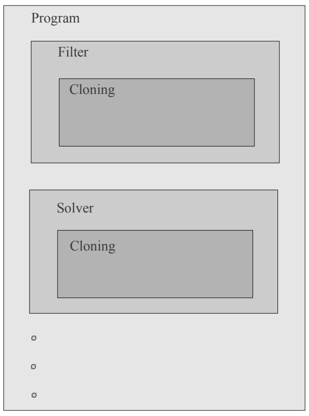


Figure 5.1: DSL scope diagram

At each scope level, it might be necessary to access a variable of a higher scope level, which is achieved in a transparent way, if there is no other variable with the same name on the current scope level, by simply referring to the desired variable. Otherwise there is the need to access that variable using a special syntax, described further ahead.

Accessing a variable inside a *clone*

To access a variable defined in a *definition* which was assigned to a variable, one starts to refer the *clone* variable, then the number of the *iteration* and finally the variable `ID`, e.g. `C[2].A`, where `C` is the *clone* variable; `2` the desired *iteration*; and `A` the variable which we desire to access.

Listing 61 shows such an example, where the statement `nak` is applied to variable `A` of the second *instance* of *clone* `C` (Line 8).

Listing 61 Referring a variable of a ***clone***

```
1 C = {
2       tcp_packet(A),
3       syn(A)
4     },
5
6 R := clone(3,C),
7
8 nak(R[1]:A)
```

Accessing a variable inside a *solver*

When there is the need to access a variable defined inside a *solver* or *filter* to state some constraint over it, one starts by referring the *filter* and then the desired variable, e.g. `gecode.A`, where `gecode` is the variable where the *solver* has been *stored*, and `A` the variable to we which we want to access.

There is also the possibility of referring a variable which is inside a *clone*, which in turn is inside a *solver*, e.g. `gecode.R[2].A`. In this case, `R` is the variable representing the desired *clone*; `2` the desired *iteration* of *clone* R; and `A` the variable that we want to access.

5.4 Implementation

The current implementation of NeMODe DSL is able to generate code for the several detection mechanisms: the Gecode solver, the Adaptive Search algorithm and the MiniSat solver. These approaches to constraint programming are different from each other, either in the way the problems are solved, and the way the problems are modeled and described, requiring distinct code generators, one for each of back-end.

Although we need to implement several code-generators, one for each solver, we were able to minimize the differences between the solvers by creating custom libraries for each constraint solver so that the code generation process may be shared between different back-ends.

Standard programming language implementation tools were used to implement parts of NeMODe: Flex and Bison, were employed to make the syntactic and semantic analysis of NeMODe programs.

Figure 5.2 represents the architecture of the code-generators; starting with a NeMODe program, which is then parsed into tokens with the help of Flex. Then, based on those tokens and the grammar which defines the NeMODe DSL, and with the help of Bison, a parser is created, which outputs a parse tree representation of the input program. Using the output of this parser we create a semantic model of the problem, and then, using the custom libraries for each of the solvers, we generate code for each detection mechanism.

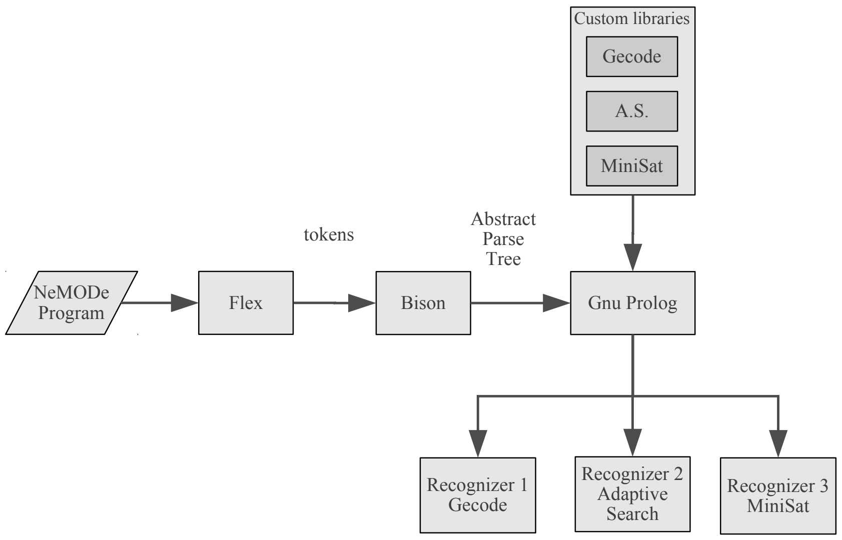


Figure 5.2: Code generators architecture

For the code generation, we choose GNU Prolog, since it presents excellent characteristics to rapidly implement prototypes as well as a great flexibility to work and manipulate tree data-structures like, such as the Abstract Parse Tree (APT), essential to generate code. So, Bison generates GNU Prolog code representing the Abstract Parse Tree, which is then analyzed in GNU Prolog, generating source code for Adaptive Search and Gecode.

The Bison and Flex modules of the code generation are shared among all back-end mechanisms, as their only task is to ensure that the input code is correct and generate an Abstract Parse Tree. After the APT has been created, the code generation is specific to each back-end. We now describe these back-end code generators.

5.4.1 Generating Adaptive Search code

The task of generating Adaptive Search code consists in creating the proper error functions which are needed for Adaptive Search to be able to solve the problem: `Cost_of_Solution` and `Cost_on_Variable`.

In order to ease the generation of this error functions, a library was created which implements small error functions, specific to the network intrusion detection domain, which are then used under a certain combination, according to the desired intrusion, which will be combined to generate the `cost_of_solution` and `cost_on_variable`.

Listing 62 shows an excerpt of the Adaptive Search `Cost_of_Solution` function generated for the port-scan attack, as described in Listing 69 (page 116).

Listing 62 Adaptive Search, `cost_of_solution` code excerpt

```
int Cost_Of_Solution(){
     ...
     err += tcp(MATRIX, sol[1]);
     ...
     err += src_dst(MATRIX, sol[1], sol[2]);
     ...
     return err;
}
```

Listing 63 presents an excerpt of the Adaptive Search `Cost_of_Variable` function generated for the port-scan attack, described in Listing 69 (see page 116).

Listing 63 Adaptive Search, `cost_of_variable` code excerpt

```
int Cost_On_Variable(int x){
     ...
     if (x == 1)
          err += tcp(MATRIX, sol[1]);
     if (x == 2)
          err += tcp(MATRIX, sol[2]);
     ...
     if (x == 1 || x == 2)
          err += src_dst(MATRIX, sol[1], sol[2]);
     ...
     return err;
}
```

5.4.2 Generating code for Gecode

Generating code for Gecode follows an approach similar to Adaptive Search, but, instead of generating error functions, it generates code based on Gecode constraint propagators and custom network propagators which describe the desired network signatures, so as to later solve the problem.

We created a custom library which defines functions that combine several stock Gecode constraints to define custom network related "macro" constraints. The same library includes definitions for a network-related constraint propagators, useful to implement some of the constraints needed to describe and solve Network Intrusion Detection problems.

With this library, the generation of code for Gecode is simplified, using the custom built, network related constraints instead of the general purpose constraints in Gecode.

Listing 64 presents an excerpt of generated Gecode code for a port-scan attack, defined in Listing 69.

Listing 64 Gecode code excerpt

```
...
must_be_tcp(vars[1]);
...
src_dst(vars[1], vars[2]);
...
branch(*this, vars[1], INT_VAR_SIZE_MIN, INT_VAL_MIN);
...
```

5.4.3 Generating code for MiniSat

The code generation for MiniSat is achieved with the help of a small library implementing the functions representing the necessary constraints to model the problem, which in turn encode each constraints as a set of clauses in Conjunctive Normal Form (CNF), necessary for MiniSat.

In particular, this library is composed of two types of functions: the functions which encode the constraints used to model the problem as CNF clauses, and the functions which are used to model the variables of the problem as a set of CNF clauses. We avoid the direct encoding of the CNF clauses, leaving the details to the library.

Code generation starts by using the functions which model the variables and their domain according to the number of variables of the problem being modeled and the size of the network traffic being used, only then, are the functions which model the constraints as CNF rules used, thus modeling the desired problem.

Listing 65 presents an excerpt of code generated for MiniSat for a port-scan attack, as defined in Listing 69 (see page 116).

Listing 65 MiniSat code excerpt

```
...
//variable encoding
setup(S, n_vars, n_packets);
...
//packet0 must be a TCP packet
pkt_field_equal(S, n_vars, n_packets, 0, 1, 1)
...
//src address of packet1 == dst address of packet0
pkt_field_equal_pkt_field(S, n_vars, n_packets, 1, 4, 0, 9);
pkt_field_equal_pkt_field(S, n_vars, n_packets, 1, 5, 0, 10);
pkt_field_equal_pkt_field(S, n_vars, n_packets, 1, 6, 0, 11);
pkt_field_equal_pkt_field(S, n_vars, n_packets, 1, 7, 0, 12);
...
```

5.5 Examples

During development of NeMODe we modeled some network attacks using the DSL in order to figure out the requirements that NeMODe should verify in terms of functionality and expressiveness, and also evaluate NeMODe. Each of the network attacks were successfully modeled and a recognizer code for each of the detection mechanisms available in NeMODe was implemented.

The network attacks we modeled were the following:

1. Portscan
2. SSH password brute-force
3. SYN flood
4. DNS spoof
5. DHCP spoof
6. ARP poisoning

We now proceed with a description of each of these.

5.5.1 Portscan

A Portscan can be the first step towards a complex network attack, having the as main purpose to figure out which services a specific host is running. This is achieved by successively connecting to the ports of the corresponding services, and, if a connection to a specific port is established, it means the host is running that particular service.

A Portscan attack can be detected by monitoring the number of network connections that are initiated and terminated a few moments later, from the same source address and to the same destination address and with different destination ports.

Should this occur, it means that either someone attempted to connect to a service not available in the host, which causes the connection to be immediately terminated, or someone connected to a service available on the host and immediately closed the connection, since the purpose of the connection was only to figure out if service was available on the host. If many of these connections are made to different destination ports on the same host in short interval of time, we are probably under a Portscan attack.

Modeling the attack

To model this attack, we create a *set* of two packets, one for the packet which initiates the connection, and other for the packet who closes it.

We then state constraints over these two packets so that the packet which closes the connection appears shortly after the packet which initiated the connection. This connection can be closed either by the attacker, if the service is available, or by the victim host, if the service is not available.

This *set* of packets is then *cloned* as many times as we find necessary to consider this an attack, and limit the maximum time interval allowed between each *set*. We also need to state that the incoming connections originate all in the same host[1], have the same destination and different destination ports.

Modeling in NeMODe

Listing 66 presents a simple example of NeMODe, describing a Portscan attack. The program starts by defining the name of the intrusion to be detected, Line 2; and then specify the network traffic source as well as the target CSP solvers to which it will be

[1]If considering a distributed attack, there is no need to check if the connections originate in the same host.

Listing 66 A Portscan attack using NeMODe

```
1   portscan {
2     RES = solve('portscan.pcap', [as,gecode,minisat]) {
3        P = {
4           tcp_packet(A), tcp_packet(B)
5           syn(A), nak(B), rst(B),
6           connection(A,B),
7           time(A) - time(B) < usecs(100),
8        },
9
10       C := clone(26,P),
11
12       same_src(C:A), same_dst(C:A),
13       different_dst_port(C:A),
14       max_interval(C) < usecs(500)
15    {
16  } => {
17    alert('Portscan attempt')
18  };
```

produced code, Line 2. In this case, the network traffic source will be a *tcpdump* log file entitled 'portscan.pcap' will generate code for all three detection mechanisms of NeMODe.

The network signature is described in Lines 2-14, and in Lines 16-18 is where the "semantic actions" to be taken if the attack pattern is found on the network are described, in this case, we only alert the network administrator.

Lines 3-8 constitute the definition of a set of rules, grouped under a *block* which has been named `P`. In this *block*, we create two `TCP` packets `A` and `B`, then state that packet `A` should have its `SYN` flag set, and that packet `B` should close the connection started by packet `A`. Then we state that packet `A` and `B` should belong to the same connection, using the statement `connection`. Finally we state and that the connection initiated by packet `A` and terminated by packet `B` should be very short, less than 100 microseconds.

Line 10 states that the packets which match the rules of block `P` are expected to occur 26 times. These occurrences are assigned to variable `C`, which is used later to state other constraints.

Line 12 states that all packets `A` of clone `C` should have the same source and destination address and line 13 states that the destination port of all packets `A` of clone `C` should have different source ports.

Finally, Line 14 states that the maximum distance between each instance of clone `C` should be less that 500 microseconds.

5.5.2 SSH password brute-force attack

A SSH password brute-force attack happens when the attacker tries to access the SSH service of a given host by *brute-forcing* SSH username/password combinations, i.e. trying a large amount of username/password combinations, based on some username/-password dictionary or some other approach, to gain access to the SSH server.

Modeling the attack

This type of attack is characterized by a large number of SSH connection attempts. To detect this attack, we can we can monitor the number of SSH connections that are initiated and terminated in a small amount of time, which means the connection was not successful. If there are a few connections like these, it means that the host is probably under a SSH password brute-force attack.

Modeling in NeMODe

Listing 67, shows how an SSH password brute-force attack can be described in NeMODe, we start by naming the network situation in Line 1, and then specify the network traffic source, the file 'ssh.pcap', and the target solvers to which we will generate code.

The network signature is actually described in Lines 3-13. Line 15, alerts the network administrator for an eventual SSH password brute-force attack using the statement `alert('SSH password brute attack')`, if the specific attack is found.

Lines 3-7 describe a `TCP` packet `A` which initiates an SSH connection. These statements are assigned to variable `P`, which later, in Line 9 we clone `10` times, meaning that we are looking for `10` packets, representing `10` SSH connection attempts.

In Line 11 we state that the packet `A` of each *instance* of clone `C` should all have the same source and destination address.

Then, in Line 12, we state that the overall time of all clones should be less than `60` seconds, the value we found reasonable to consider it an attack, using the statement `max_duration`.

Finally, in Line 15, we alert the network administrator for an eventual SSH password brute-force attack using the statement `alert('SSH password brute attack')`.

Listing 67 An SSH password brute-force attack using NeMODe

```
 ssh_brute_force {
   RES = solve('ssh.tcpdump', [as,gecode,minisat]) {
      P = {
         tcp_packet(A),
         dst_port(A)==22,
         syn(A), nak(A)
      },

      C := clone(10,P),

      same_src(C:A), same_dst(C:A),
      max_duration(C) < secs(60)
   }
} => {
   alert('SSH password brute attack')
};
```

5.5.3 SYN flood

A SYN flood attack happens when the attacker initiates more TCP/IP connections than the server can handle, and, at the same time, ignores the replies from the server, forcing the server to have a large number of half open connections in standby, leading to a Denial-Of-Service when this number reaches the limit of connections.

Modeling the Attack

This type of attacks can be detected if a large number of connections is made from a single host to a specific host in a very short time interval, meaning the host is probably under a SYN flood attack.

Modeling in NeMODe

Listing 68 shows how a SYN flood attack can be described using NeMODe, we start by defining the name by which this network situation will be known, Line 1, and specifying the network traffic source, the *tcpdump* network traffic log 'syn.tcpdump' and the solvers for which will be generated source code, Line 2.

From Line 3 to Line 13 is where the signature is actually described. Line 15 describes the action to take if the network situation is found, alerting for a SYN flood attack.

In Lines 3-13 we state that `A` should be a TCP packet, which must have its `SYN` flag set and a null acknowledgement field. These statements model a network packet which initiates TCP connections, which are assigned to variable `P`.

Then, in Line 9, we clone the statements defined in variable `P` `30` times, the number we found reasonable to be considered a SYN flood attack, representing `30` connection attempts. Then we store this *cloning* as variable `C`.

After the cloning, in Line 11 we specify that all packets of all instances of *cloning* `C` should all have the same source and destination address. Also, in Line 12, we define the maximum interval time between each instance of cloning `C` should be inferior to 500 microseconds.

Listing 68 A SYN flood attack in NeMODe

```
1   syn {
2     RES = solve('syn.tcpdump', [as,gecode,minisat]) {
3        P = {
4           tcp_packet(A),
5           syn(A),
6           nak(A)
7        },
8
9        C := clone(30,P),
10
11       same_src(C:A), same_dst(C:A),
12       max_interval(C) < usecs(500)
13    }
14  } => {
15       alert('SYN flood attack, packet')
16  };
```

A distributed SYN flood attack

This description of the problem can be trivially modified for a Distributed SYN flood attack, where the attackers are geographically distributed, allowing for a stronger attack [1]. To make this transformation, we only need to remove the statement `same_src(C:A)` from Line 11, allowing the packets to originate from several hosts.

5.5.4 DNS spoof

A DNS spoof is a Man in The Middle (MITM) attack, where the attacker tries to provide a false answer to a DNS query posted by the victim host. If the attack succeeds the victim could be accessing a host controlled by the attacker instead of the legitimate host. This allows the attacker to extract information from the victim.

Modeling the Attack

In order to perform this type of attacks, the attacker tries to respond with a false DNS answer faster than the legitimate DNS server, providing a false IP address for the name to which the victim was querying.

To detect this type of attacks, we want to look for several replies to the same DNS query, indicating the host might be under a DNS spoof attempt.

Modeling in NeMODe

Listing 69 shows how this attack can be modeled in NeMODe. We start by naming the intrusion in Line 1, followed by the network traffic source in Line 2. The actual description of the desired network situation is done in Lines 2-14. Line 16 states what actions to take if the situation is found, in this situation the network administrators are alerted for an eventual DNS spoof attack.

Line 3 describes the packet that makes the DNS request. Lines 5-6, models a first reply to the DNS request and lines 8-9 describes the second reply.

From Line 11 to Line 13 we state that the network packets `B` and `C` should be different and that the *DNS id* in replies should be the equal to the *DNS id* of the DNS request. The *DNS id* is represented in the first two bytes of the packet data.

5.5.5 DHCP spoof

A DHCP spoof is another Man in The Middle (MITM) attack, where the attacker tries to reply to a DHCP request faster than the legitimate DHCP server for the local network, allowing the attacker to provide false network configurations to the victim host, e.g. a fake default gateway, which forces all traffic from and to the victim host to pass through an attacker controlled host, allowing it to capture or modify sensitive data.

Listing 69 A DNS spoof attack programmed in NeMODe

```
dns_spoofing {
    RES = solve('dns.pcap', [as,gecode,minisat]) {
       udp_packet(A), dst_port(A) == 53

       udp_packet(B), src_port(B) == 53,
       dst(B) == src(A), dst_port(B) == src_port(A),

       udp_packet(C), src_port(C) == 53,
       dst(C) == src(A), dst_port(C) == src_port(A),

       B != C,
       data(B,0,2) == data(A,0,2),
       data(C,0,2) == data(A,0,2)
    }
} => {
    alert('DNS Spoofing attempt')
};
```

Modeling the Attack

This kind of intrusion can be detected by looking for several answers to a single DHCP request, originating in different hosts. If the attacker spoofs its IP addresses, this detection method needs to be tuned (e.g. use Media Access Control (MAC) addresses).

Modeling in NeMODe

A NeMODe program which models a DHCP spoof situation is presented in Listing 70. The signature is described in Lines 2-8. Line 10 states which actions should be taken if the specific network situation is found.

Line 3 describes the packet that initiates a DHCP request, Line 4 describes a first reply to such request and Line 5 describes a second reply the DHCP request.

Finally, in Line 7, states that packets `B` and `C`, the first and second replies, should have different source addresses.

Listing 70 A DHCP Spoofing attack programmed in NeMODe

```
dhcp_spoofing {
    RES = solve('dhcp.tcpdump', [as,gecode,minisat]) {
       udp_packet(A), dst_port(A)==67,
       udp_packet(B), dst_port(B)==68,
       udp_packet(C), dst_port(C)==68,

       src(B) != src(C)
    }
} => {
    alert('DHCP Spoofing attempt')
};
```

5.5.6 ARP poisoning

An ARP poisoning attack happens when someone tries to poison the ARP tables of a router or specific host with fake data, making an IP address point to a MAC address corresponding to some other host which is not the legitimate *owner* of the given IP address.

This kind of attacks allows the attacker to gain unauthorized access to information, destined to someone else.

Modeling the Attack

This type of attack is achieved by sending a series of ARP packets with fake information in order to poison the ARP tables of the desired hosts.

One way to detect ARP poisoning attacks is to monitor ARP packets, looking to see if there are different IP addresses assigned to the same MAC address in a short time. If this happens, the host is most likely under an ARP poisoning attack.

Modeling in NeMODe

Listing 71 presents a possible description of an ARP poisoning attack in NeMODe. It starts by naming the specific network situation in Line 1, then stating what is network traffic source, and then specifying which solvers are going to be used.

The description of network signature is actually done in Lines2-21. In Line 23 we alert the administrator if the specific attack is found.

Listing 71 An ARP poisoning attack programmed in NeMODe

```
1  arp_poisoning {
2      RES = solve('arp.tcpdump', [as,gecode,minisat]) {
3         arp_packet(A), arp_reply(A),
4         arp_packet(B), arp_reply(B),
5         arp_packet(C), arp_reply(C),
6         arp_packet(D), arp_reply(D),
7
8         time(A) < time(B),
9         time(B) < time(C),
10        time(C) < time(D),
11
12        src(A) == src(B),
13        src(A) == src(C),
14        src(A) == src(D),
15
16        src_mac(A) != src_mac(B),
17        src_mac(A) != src_mac(C),
18        src_mac(A) != src_mac(D),
19
20        time(D) - time(A) < secs(5)
21     }
22 } => {
23     alert('ARP poisoning attempt')
24 };
```

Lines 3-6 describes four packets which should be ARP replies, representing the ARP replies that we are looking for. Lines 8-10 states that these packets should be in that specific temporal order, so later we can specify a global time interval between the first and the last ARP reply, in Line 20.

In Lines 12-14 we state that packets `A,B,C` and `D` should all have the same Sender Protocol Address (SPA), also known as IP Address, and in Lines 16-18, we state that the Sender Hardware Address (SHA) of packet `A`, also known as MAC Address, must be different from the SHA of packets `B,C` and `D`. This means that we have packets `A,B,C` and `D` with the same IP address, but the MAC address of packet `A` is different from the MAC address of packets `B,C` and `D`, indicating that we are probably under an ARP poisoning attack.

To make this signature stronger, in Line 20 we state that the time interval between network packets `B` and `D` should be less than 5 seconds, since an ARP poisoning attack tends to produces several ARP replies in a short time interval.

5.6 Conclusion

In this chapter we have described the Domain Specific Language provided by NeMODe, a small, simple programming language about network entities related to Network Intrusion Detection, which allows easy description of the desired situations to be found in the network traffic, using a declarative approach underlain by the use of constraint programming.

We have described in detail the specification of the DSL to help understand its syntax and semantics. Besides the description of the DSL, we also described its implementation as well as some examples of how to use the language.

The NeMODe DSL presented allows for an easy description of specific network situations. Code generation for each of the back-end detection mechanisms is available in NeMODe, enabling a parallel detection process, using all the mechanisms concurrently, in search for the faster solution.

The examples presented, although simple, demonstrate the expressiveness of NeMODe in describing network signatures.

Chapter 6

Sliding Network Traffic Window

This Chapter describes a Sliding Network Traffic Window and its implementation in Adaptive Search, simulating the essence of live network traffic, where new network packets are constantly arriving. Adopting a scheme such as this allows us to make a first evaluation of NeMODe while analyzing simulated live network traffic.

6.1 Introduction

The Gecode, Adaptive Search and MiniSat back-end detection mechanisms of NeMODe work on a static network traffic log, which may be obtained with `tcpdump` [110], a network packet sniffer, while a computer is under an actual attack.

By using a static traffic sample, we are limiting the capabilities of the detection mechanism. This is necessary because watching live network traffic would be difficult to handle, performance-wise, and also because we need to establish benchmark results which require a fixed data set.

Introducing a network traffic window that changes over time, which slides across a larger set gives the solver new capabilities, allowing it to analyze a much larger data set than was previously possible. Besides, if we get the network traffic window to *slide* across live network traffic, it allows us to analyze live network traffic in real time, by updating the network traffic window with incoming network packets captured from the wire.

If, instead of simply slide the network window over a larger set, or updating it with fresh network packets, we *keep* in the network window past packets which seem interesting for the network situation that we are trying to detect, we get the capability of detecting attacks that spread across a window larger than that previously used, thereby including a range of packets that span a considerably larger time interval than was previously attainable.

In this Chapter, we present a sliding network traffic window scanner using the Adaptive Search detection mechanism of NeMODe, with the ability to *buffer* packets which are relevant to the situation being analyzed, allowing the detection of attacks over a wider time interval. Our goal is to create a strong platform on which to perform network intrusion detection, on live network traffic, in the near future.

Adaptive Search was chosen as the solver to implement the sliding network traffic window since, from the solvers we have experimented with, it is the one which is most easily modified and is less sensitive to changes, such as the changes on the network traffic window, due to the customizability of the Adaptive Search algorithm.

The introduction of a sliding network traffic window is depicted in Fig. 6.1, where the network traffic source can be seen. This will be used to update the network traffic window when a new network packet arrives, inserting it into the traffic window, which is then used as input to the detection mechanism, in this case, Adaptive Search.

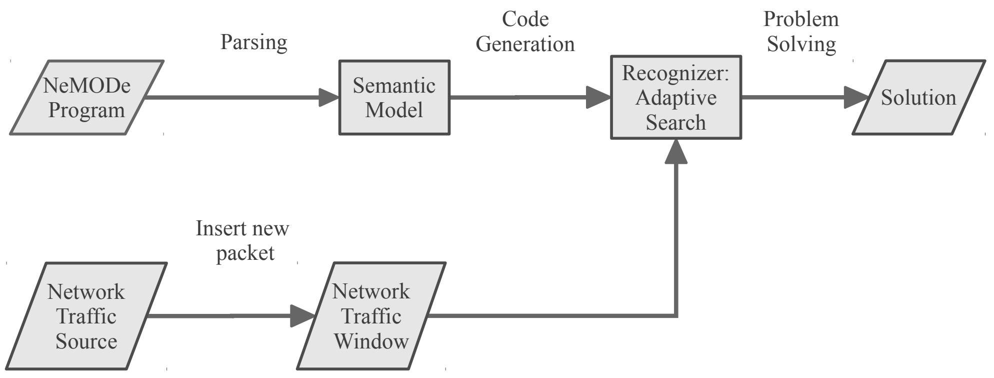


Figure 6.1: Network Traffic Sliding Window diagram in Adaptive Search

6.2 Sliding Network Traffic Window in AS

Adaptive Search relies on heuristics, reflected in the *error functions* in order to reach a solution to a combinatorial problem. In a Network Intrusion Detection problem, these heuristics directly pertain to the network traffic window, since the *error functions* are calculated by analyzing the packets actually found in the traffic window.

Due to this direct influence of the network traffic window over Adaptive Search heuristics, any changes made to the network traffic window will have an immediate effect in the heuristic functions used by Adaptive Search, changing the way it seeks for a solution and, most importantly, automatically adapting to any change made on the network traffic window.

Adaptive Search reaches a solution to a problem by starting with an initial state, and then iteratively performing minor changes to it, until an objective function is satisfied. At each step, every variable of each "tentative" solution is already assigned with a value, which is a reference to an actual network packet that belongs to some instance of the network packet window. So, when a network packet is removed from the packet window to make room for another packet, the "tentative" solution is no longer valid. Due to the high performance and insensitiveness to previous context of the Adaptive Search algorithm, it adapts very quickly to the new "instance" of the network packet window, without requiring any changes to the code.

6.2.1 Updating the Sliding Window

In order to update the sliding window with new packets, we decided to use a *first in first out* access discipline, where the oldest packet in the network traffic window is replaced by the newest packet arriving in the network. Two versions of this approach were implemented, and, depending on the network case being analyzed, the most suited version is used:

1. Remove oldest packet, insert new packet
2. Remove oldest not relevant packet, insert new packet

We now discuss both approaches:

Remove oldest packet, insert new packet

In a first version, when a new packet arrives, we simply remove the oldest network packet from the network traffic window and insert the new one in its position. At some point, while inserting new packets replacing the old ones, the packets in the network traffic window are no more ordered as in the original network traffic source, since we don't shift the packets when inserting a new one. This does not poses a problem to Adaptive Search, since it uses each packet time stamp when there is the need impose temporal order between network packets.

Listing 72 presents the pseudo code to insert a new packet in the network packet window, replacing the oldest one, where `i` represents the *index* of the oldest packet, or the position to insert the new packet, `window` is the network packet window, `new_packet` is the new packet and `window_size` is the network packet window size.

While updating the network traffic window, we keep track of the index of the last packet inserted, so, when a new packet arrives, we immediately know where to insert it, the next index. If the last packet was inserted in the last position of the network traffic window, the oldest packet is the first packet of the network traffic window, so, the new packet is inserted in that position.

Listing 72 Remove oldest packet, insert new packet

```
i = 0;                    //start at the beginning of the window
 do{
  wait_for_new_packet();    //wait for a new packet

  window[i]=new_packet;     //insert the new packet

  //prepare the index for the next packet
   if( i == last_position )
     i=0;                   //the next position is the first
   else
     i = i + 1;             //go to next position
}
```

This version of the sliding window is most suited to network situations in which the network packets which makes prove the existence of an intrusion are close together and fit in a small network traffic window, because this approach limits the number of network packets that compose the signature of an intrusion situation.

Remove oldest not relevant packet, insert new packet

The second version of the sliding window was implemented in order to keep specific packets in the network packet window: the ones which are understood to be important for the desired network situation, even if they are among the oldest packets and would otherwise be replaced by new ones. This approach allows us to detect intrusions in a wider range than the network traffic window being used, since the relevant packets to such situation are being “buffered” in the sliding network traffic window, allowing them to be related to newer packets which appear later in the network traffic.

The update of the network traffic window while using this approach is achieved in the following way: start by filling the network packet window with until the window is full, then, start on the oldest network packet, and check if it is relevant to the desired situation, if so, skip that network packet and test the next oldest packet. This procedure is repeated until a packet not counted as relevant is found, which became the one to be replaced by the new one.

This approach leads to one problem; the network window could get *clogged* with relevant packets, at risk of reaching a state where it is impossible to insert new packets, since every packet in the window is relevant. To deal with this situation, we introduce a new heuristic function, based on the age of the packet to decide when a previously buffered packet should not be anymore considered relevant and be replaced by a new one.

The pseudo-code presented in Listing 73(page 125) describes the process of inserting a new packet on the network window while preserving the relevant network packets. Variable `i` is the current *index* of the window where the new packet will be inserted, `window` an array representing the network packet window, and `new_packet` the newly arrived network packet. The function `wait_for_new_packet();` waits for a new packet in order to continue with its insertion on the network window, the `relevant(p)` function decides whether the network packet `p` is relevant to network situation being described, and the function `not_too_old(p)` decides if the packet has been *frozen* enough time to be removed from the window.

Listing 73 Remove oldest not relevant packet, insert new packet

```
i = 0;
 do {
  wait_for_new_packet();           //wait for a new packet

  //find the oldest packet not relevant or
  //the packet buffered for more time
   while( relevant(window[i]) AND not_too_old(window[i] ) ){
     if( i == window_size )
       i = 0;                     //go back to first position
     else
       i++;                       //test the next position
  }

  window[i] = new_packet;         //insert the new packet

  //prepare the index for the next packet
   if( i == last_position )
       i = 0;                //the next position is the first
   else
       i++;                  //go to next position
}
```

6.2.2 Deciding if a Packet is Relevant

The decision of checking if a network packet is relevant to the desired network situation is critical in keeping the *interesting* network packets in the traffic window, so as to relate them with packets that may appear in the future, beyond the limits of the network packet window size.

Deciding if a network packet is relevant is directly related to the intrusion being described as well as its signature, since that decision is achieved through the use of subset of the heuristics that have been used in the description of the network attack as an Adaptive Search problem. These heuristics are applied to the network packets being checked for relevance, and, depending on the result, the packet is considered relevant or not. These heuristics are usually very simple, checking specific packet fields such as ports, flags, addresses or time-stamp of a network packet.

For example, in a SSH password brute-force attempt, the network packets that might be relevant are the ones that are actually SSH packets. Also, only the ones that initiate a network connection are relevant, so, they should have its SYN flag set and not *acknowledge* another network packet. Listing 74 presents the heuristics to decide if a network packet is relevant to a SSH password brute-force attack. If the total heuristic functions value is null, the network packet is considered relevant and is kept in the window, otherwise, its replaced by a new packet.

Listing 74 Checking if packet is relevant

```
bool relevant(packet){
     int err=0;
     err+=syn(packet);
     err+=not_ack(packet);
     err+=dst_port_must_be(packet, 22);

    if( err == 0 )
       return true;
}
```

The relevance of a network packet for a specific intrusion is given by a boolean function, i.e. the packet is relevant or not. In the future we pretend to change this, allowing the packets to have different levels of relevance, which in turn could be used to best decided if a specific packet is kept in the traffic window or to help the heuristic search.

6.2.3 Window Update and Continuous Solutions

The arrival of new packets on the network traffic is completely independent of the Adaptive Search solving process, so, these two processes are handled by different threads. If one process were to depend on the other, it could lead to performance issues and miss the detection of some packets, which, in turn, could lead to not recognizing an attack, so, these two processes must run at the same time, updating the network packet window with new packets while the solver is continuously searching for solutions.

In order to combine these two processes, we decided to launch a thread (POSIX thread) from the main solver function, which will take care of everything involved in updating the network traffic window, listening for new packets, making the necessary maintenance to the current window instance, deciding if some of the network packets are relevant and should be *kept* or removed from the window, as well as inserting the newly arrived packet in its proper location.

Working with threads which access the same data at the same time could lead to inconsistencies and synchronization problems. To prevent this type of problems we decided to use the mutual exclusion mechanisms available in the POSIX threads (`Mutexes`) to block some procedures of the window update while important steps of the solver are being executed.

In particular, we have used the `Mutex` mechanism to block the procedure that updates the window while the solver is deciding if a current solution is in fact a solution to the problem. This is a critical step for the solver, which could lead to the loss of a valid solution if the window suffers any change at the time the solution is being checked for.

Listing 75 presents a snippet of code that demonstrates how the main Adaptive Search solver process is synchronized with the window update procedure by using `Mutexes`. In Listing 75, `main()` is the Adaptive Search `main` function, `Solve()` is the function which actually solves the problem and finds a solution, `Check_Solution()` verifies if a tentative solution is valid and needs to have exclusivity over the network packet window, not allowing any changes to its content while `Check_Solution()` is being executed.

The function `update_window()` is executed as a separate thread, launched from the Adaptive Search `main` function, and should not be executed while the solver is checking if a tentative solution is valid, so, immediately before the call of `update_window()` we wait until `Check_Solution()` has completed and unlocked the `mutex`. After that, we can update the window, and lock the `mutex` to make sure the solver doesn't try to check for a solution while the window is being updated. After the window has been updated, the `mutex` is unlocked, freeing the window update process and allowing both window update and solve process to work in *parallel*.

Listing 75 Combining Adaptive Search with the Window Update Thread

```
main(){
   ...
   pthread_create( &thread, NULL, update_window, null);
   ...
   for(;;){
       solution=Solve();
       ...
       pthread_mutex_lock(&mutex);
       Check_Solution(solution);
       pthread_mutex_unlock(&mutex);
   }
}

update_window(){
    ...
    for(;;){
       ...
       while( !new_packet() ) {}
       ...
       //wait for Check_Solution to terminate
       for(;;){
          if( pthread_mutex_trylock(&mutex)==0 )
             break;
       }
       update_window();
       pthread_mutex_unlock(&mutex);
       ...
    }
}
```

Figure 6.2 presents a flowchart representing both the Adaptive Search solving process, and the window update process. On the left side of the diagram is the *solve* process, on the right side the window update process. As it can be seen, and described above, each process must wait for each other at specific points through the use of the `Mutex` mechanism, using `pthread_mutex_trylock()` to gain the lock over the `mutex`, and `pthread_mutex_unlock()` to release it.

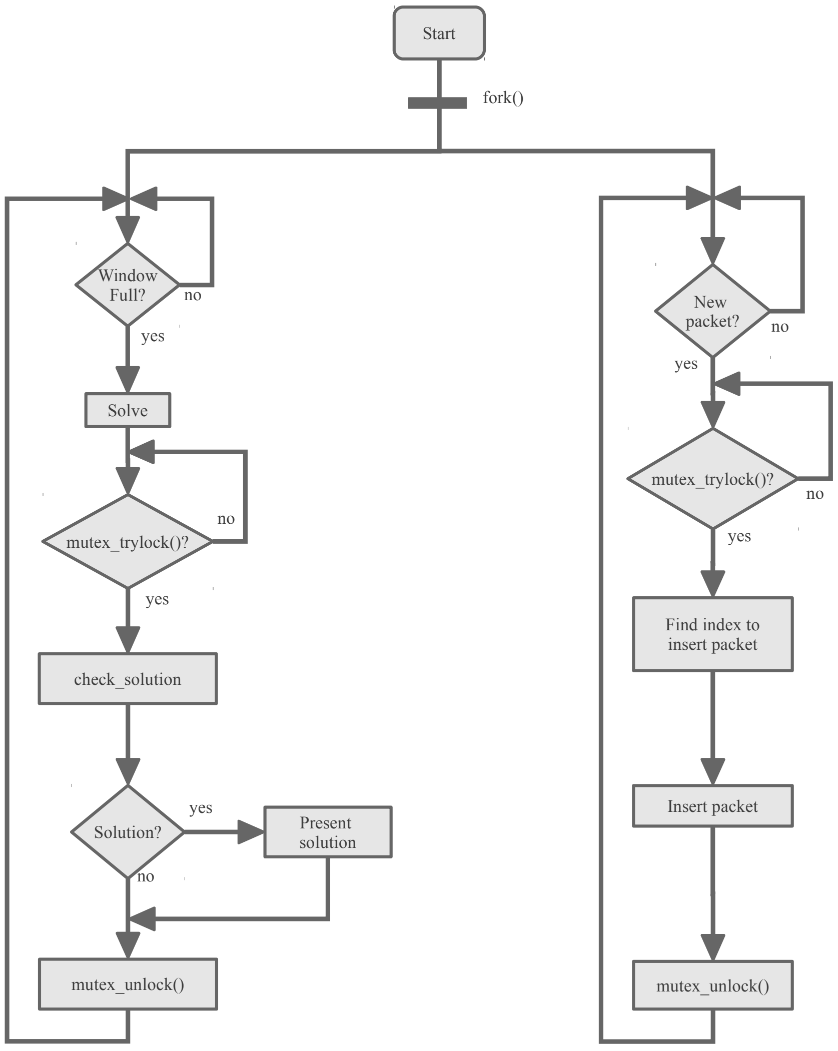


Figure 6.2: Window update while solving - Flowchart

6.3 Simulating live network traffic

The sliding network traffic window is implemented to use a `tcpdump` log file as network traffic source, but is designed to simulate live network traffic, up to a certain level, by simulating the network packet arrival at a given rate, which is accomplished by controlling when and which packet is considered a newly arrived packet, so it can be processed in order to be inserted in the network traffic window.

The network traffic source is represented internally in Adaptive Search as an `array` of network packets, so, each new packet to be inserted in the network traffic window will be copied from this `array` to the network traffic window, one at a time, following the order in which the packets were captured by `tcpdump`, starting from the oldest packet.

While updating the traffic window with new packets from the network traffic source, we keep track of which was the last packet from the source to be inserted in the network traffic window. This way, when its time to insert a new packet, we immediately know which is the next new packet to be inserted in the network traffic window.

To simulate live network traffic using a `tcpdump` log file, we introduce a *sleep* time between the update of the network traffic window with new packets, thus, simulating the arrival of new network packets at a given network bandwidth.

To implement this, we used the `nanosleep` function, which allows one to specify the time a process will be *frozen*, in seconds and nanoseconds, allowing to fine-tune the simulated network bandwidth.

The processes of simulating the network bandwidth and updating the network traffic window are completely integrated, as shown in Listing 76, which shows how these processes are combined in order to simulate a live network traffic, using a static network traffic source.

We start with the first packet of the network traffic source, and go up to the last packet of the network traffic source(Line 1), inserting them, one by one, at specific time intervals, in its correct position in the network traffic window. Next, we specify the time interval and use the `nanosleep` function to simulate the interval between two packet, Lines 3 to 8. The interval time is specified in variable `tim`, which is a data structure representing the time the process will be sleeping, and is changed according to the desired `sleep` time.

After the `sleep` time has elapsed, and the index to insert the new packet is calculated, Line 11, described in Sect. 6.2.1, the new network packet is copied from the network traffic source to the network traffic window, in the correct position, Lines 13-13, where `window` represents the network traffic window, `source`, the network traffic source, `index`, the index where the new packet will be inserted, `i`, the index of the new packet in the network traffic source, and `nfields`, the number of fields used to represent a network packet.

This approach to simulate real live network traffic allows the fine tuning of the packet arrival rate, thus simulating different network traffic speeds, allowing to test NeMODe in situations similar to real network traffic, at different network bandwidths.

6.4 Towards live network traffic

Although the network traffic sliding window is implemented to work only with static network traffic as network source, using live network traffic instead of network traffic logs can be done with few changes to the current implementation of the sliding network

Listing 76 Simulating live network traffic

```
   for(i=0; i<source_size; i++){

    //wait for next packet
    struct timespec tim, tim1;
    tim.tv_sec = 0;
    tim.tv_nsec = nsleep;

    nanosleep(&tim , &tim1);

    //calculate index to insert new packet
    ...

    //insert new packet
    window[i] = source[i];

  }
```

traffic in Adaptive Search. This section presents an approach on how to change the network traffic window in Adaptive Search so it can use live network traffic to look for the desired network case scenarios.

The network traffic log files used as network source are created with the help of `tcpdump`, which in turn relies on `libpcap` [30], an implementation of `pcap`, which is an API for capturing network traffic, to perform all the network traffic capture. `Libpcap` is used in many Intrusion Detection Systems, such as Snort [27] to perform network traffic capture and then perform the detection of the desired attacks and is almost considered a standard method to capture network traffic in Unix-like systems.

`Libpcap` allows to access each individual piece of data of each captured network packet, allowing to select only the relevant data of intrusion detection. This makes `libpcap` an ideal tool for NeMODe to perform network traffic capture in order to work with real time network traffic.

Shifting NeMODe towards live network traffic, more specifically, can easily be accomplished by integrating `libpcap` in Adaptive Search, allowing it to perform network traffic capture, and, at the same time, inserting the newly captured network packets in the network traffic window, thus working the same way as the network traffic window which we already implemented and described in Sec. 4.5, only this time, using live network traffic.

`Libpcap` allows some basic filtering while capturing network packets, which may dramatically reduce the size of the network traffic that needs to be analyzed. This characteristic can be used by NeMODe in order to reduce the number of network packets being analyzed. To take the advantage of this pre-filtering provided by `libpacp` we need to implement a first filtering level on the NeMODe DSL, which allows the specification of very simple filtering rules that specify which network packets are relevant to the network intrusion signature being analyzed, and are actually captured by `libpcap`.

The use of live network traffic as a network source was only implemented as a small prototype with a very simple, *hard coded* network situation to assess if using `libpacp` to capture network traffic and update the network traffic window to deal with live network traffic is valid. This prototype showed it is quite viable to use such an approach to live network traffic monitoring using Adaptive Search with a sliding network window.

Figure 6.3 presents a diagram of how we could integrate `lipbcap` in NeMODe, where it can be seen that the DSL of NeMODe, besides generating code for each for back-end available in NeMODe, also produces `libpcap` rules which will be used as input to `libpcap` to perform the network traffic capture, already pre-filtered according to the `libpcap` rules generated by the DSL, and specific to the network situation.

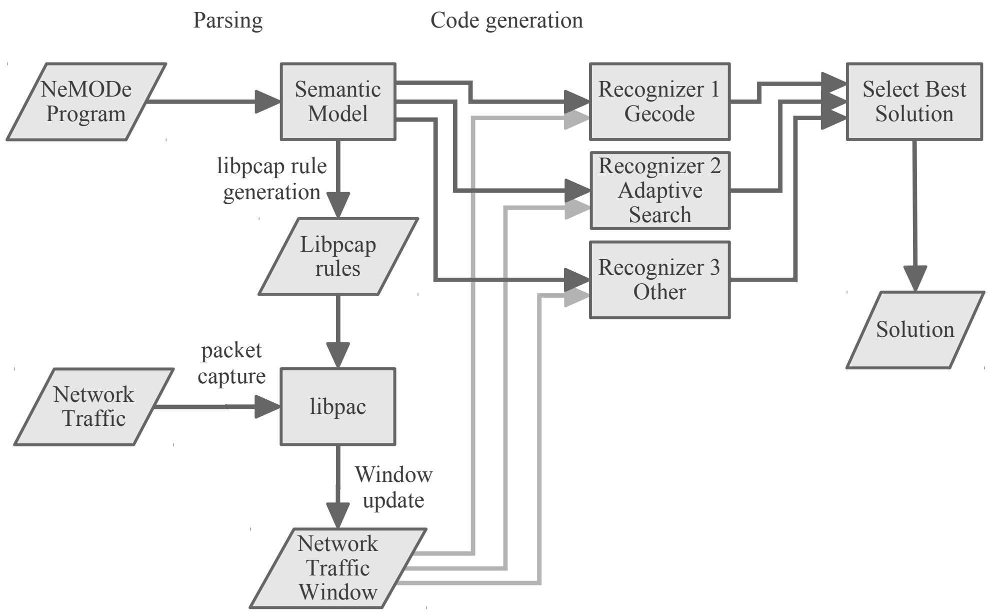


Figure 6.3: Live network traffic - Tentative diagram

6.5 Conclusion

In this Section, we presented an Adaptive Search detection mechanism for NeMODe, using a sliding network packet window, a significant step towards network intrusion detection on live network traffic, allowing us to work with network traffic logs much larger that we were able to in previous versions.

We demonstrated that with minor changes to Adaptive Search, it is possible to adapt it to work with a dynamic, sliding network window, which changes over time, and still detect the desired network situations.

Although the tests used network traffic logs, the results can be compared to using live network traffic, since the logs were created to simulate live network traffic and the solver adapted to simulate the arrival of network packets at a given rate.

Also important is the fact that, using this approach, we can now detect network situations that are *wider* than the network window used by the solver, allowing the detection of network attacks that span a considerable time interval.

We also formulate the use of live network traffic as network traffic source in order to update the network traffic window, thus, allowing the continuous monitoring of live network traffic, looking for the desired network situations.

Although the integration of `libpacp` in NeMODe, was only implemented as a rough prototype to verify the feasibility of the approach, and no benchmark results were produced, we are confident that we will be able to monitor live network traffic, at least with the Adaptive Search back-end detection mechanism of NeMODe.

Chapter 7

Experimental Evaluation

This Chapter reports and evaluates the experimental results performed while developing NeMODe in order to assess its performance, figure out its weaknesses, and enhance its capabilities, either in terms of performance, usability or expressiveness. We present the results obtained for each case in each detection mechanism as well as the evaluation of each case and solver. We also make a comparison of NeMODe with other approaches to Network Intrusion Detection.

7.1 Introduction

To evaluate NeMODe we experimented with several network situations in order to understand the behavior of the system, its performance, usability and expressiveness, under different conditions while using each of the detection mechanisms available in NeMODe.

Using these network intrusion signatures made us enhance NeMODe, forcing us to add more features to cope with all the examples that we tested. Also, these tests lead to more system tuning, due to the variety of tests and the differences between them.

The network intrusions that we decided to model are all signature-based, which can be modeled by stating relations between several network packets, thus, describing network intrusions which span several network packets. This is one of the most important features of NeMODe, which, either lack in most Network Intrusion Detection systems, or, when available, it only allows relations between different network packets in a very limited way.

We used a dedicated computer with the x86 architecture to run each test with each detection mechanism, and, since Adaptive Search had been recently ported to the Cell/B.E. architecture [111], we have also tested all the network situations using the Adaptive Search Cell/B.E. version.

The x86 computer architecture was an HP Proliant DL380 G4 with two Intel(R) Xeon(TM) CPU 3.40GHz and with 4 GB of memory, running Debian GNU/Linux 4.0 with Linux kernel version 2.6.18-5. As for the tests with the Cell/B.E. Adaptive Search version, they were run on a IBM BladeCenter H equipped with QS21 dual-Cell/BE blades, each with two 3.2 GHz processors, 2GB of RAM, running RHEL Server release 5.2.

During all experiments, we froze the versions of each constraint solver, to ensure reliable results without external influence. The versions used in our experiments are:

1. Gecode 3.1.0, compiled from source
2. Adaptive Search 0.9.0, compiled from source(x86)
3. Adaptive Search 0.9.0, compiled from source(Cell/B.E.)
4. MiniSat2 070721, compiled from source

To perform the experiments, we used log files representing network traffic containing the desired signatures to be detected. These log files were created with the help of *tcpdump* [110], a packet sniffer, while a computer was undergoing the desired attacks, thus, simulating a real-life scenario.

While developing NeMODe, we introduced the ability to use a sliding network traffic window over a network traffic log, thus, two major types of experiments were performed:

1. Experiments using a static network traffic window.
2. Experiments using a sliding network traffic window.

7.2 Analyzed Intrusion Signatures

In this section we present and describe the network situations that we have tests in NeMODe, for both static and sliding network packet windows, as well as the parameters used in each problem. The network situations that we decided to model in order evaluate NeMODe are the ones already presented and described in Section 5.5:

1. Portscan
2. SSH password brute-force

3. SYN flood
4. DNS spoof
5. DHCP spoof
6. ARP poisoning

All of these network situations were tested using the static network traffic window NeMODe version. For the sliding network traffic window NeMODe, only the following network situations were used, with a modeling identical to the previous ones:

1. SSH password brute-force
2. DNS spoof
3. DHCP spoof
4. ARP poisoning

The situations modeled in the static network traffic window NeMODe were all successfully described in NeMODe DSL, and code generated for each of the 3 back-end detection mechanisms, Gecode, Adaptive Search (both x86 and Cell/B.E.), and MiniSat.

For the sliding network traffic window version, we only generated code for x86 Adaptive Search.

The code generated for each network situation and for each solver was then run in the HP Proliant DL380 G4, and in the Cell/B.E. machine for the Cell/B.E. Adaptive Search version.

7.2.1 Portscan

For the Port-scan attack, we have created two network traffic log files, one composed of 100 network packets, and another of 400 packets. These log files were created with the help of `tcpdump` while a computer was being a victim of a Portscan, performed with the help of `nmap` [112], a security scanner used to discover computers and services on computer networks.

Nmap was used on its most simple form to perform the scan of the victim host services, as in Listing 77, where `10.10.10.59` is the IP address of the target victim.

Listing 77 Performing a Portscan

```
nmap 10.10.10.59
```

The `tcpdump` network log file was used the as the network traffic source for all 3 detection mechanisms running on the x86 as well as the Cell/B.E. with Adaptive Search.

The Portscan signature was modeled with 52 network packet variables, so, a solution to this problem is a set of 52 packets, taken from a set of 100, or 400 network packets, the *tcpdump* log file.

7.2.2 SSH password brute-force

To test the SSH password brute-force attack, we created network traffic log files while a computer was trying to gain access to the victim's SSH service using a brute-force approach, successively using username/password combinations to attempt to force access to the service. To perform the attack, we used `Medusa` [113], a fast, parallel, and modular login brute-forcer which supports many services, including SSH.

To perform the SSH password brute-force attack we used the `Medusa` tool as in Listing 78, where `10.10.10.59` is the IP address of the target host, `pds` the username used in the attack, and `password.list` the name of the file with the list of passwords used during the attack.

Listing 78 Performing an SSH password brute-force attack

```
medusa -h 10.10.10.59 -u pds -P password.list -M ssh
```

The `tcpdump` log file was used as the network traffic source for all tests with the SSH password brute-force attack, either in both x86 and Cell/B.E. architectures.

To perform the detection of the SSH password brute-force attack in NeMODe, the signature used to model the problem was composed by 10 network packets variables, so a solution to this problem would be a set of 10 packets taken from the network traffic source. The same modeling was used in the two versions of NeMODe, the static and sliding network traffic window versions.

Static network traffic window

While using the static network traffic window NeMODe version, two network traffic files were used, one with 182 and another with 400 network packets, each one containing 10 failed attempts to access the SSH service.

Sliding network traffic window

On the sliding network traffic window NeMODe version, the network traffic source was composed by 3000 network packets, containing 10 SSH brute-force attacks. We tested this attack while varying the size of the window, using the 10, 20, 30, 100 and 300 network packets as the size of the network traffic window, and also, the update time the network traffic window, so we could understand how the system reacts to these changes.

7.2.3 SYN flood

The network traffic source files used in NeMODe while detecting the SYN flood attack were two *tcpdump* network traffic logs files, one composed of 100 network packets and other of 400 network packets, both generated while the chosen victim host was under a SYN floodattack.

This attack was achieved with the help of `hping` [114] a packet generator and analyzer for several protocols, including TCP/IP. The attack was achieved using the options presented in Listing 79, where `10.10.10.59` is the IP address of the target host, and `80`, port of the target service.

Listing 79 Performing a SYN flood attack

```
hping -i u1 -S -p 80 10.10.10.59
```

The same `tcpdump` network traffic log file was used in all tests performed with the SYN flood attack.

The SYN flood attack was modeled in NeMODe using a network signature composed of 30 network packet variables, so that a solution to this problem was a set of 30 packets taken from the traffic source, composed of 400 packets.

7.2.4 DNS spoof

The DNS spoof attack detection was performed in NeMODe using, as a network traffic source, `tcpdump` network log files, generated while the host chosen as victim was under a DNS spoof attack, performed with `Ettercap` [115], using the DNS plugin, as in Listing 80, where `10.10.10.254` is the IP address of the legitimate DNS server of the host victim network, and `10.10.10.59`, the victim's host IP address.

Listing 80 Performing a DHCP spoof attack

```
  ettercap -i eth0 -T -M arp:remote -P dns_spoof \
                                   /10.10.10.254/ /10.10.10.59/
```

Besides these call parameters, we also changed the default settings of the DNS plugin so that the name `microsoft.com` be resolved to `209.92.24.80`. To do so, we added the lines presented in Listing 81 to file `/usr/share/ettercap/etter.dns`.

Listing 81 `/usr/share/ettercap/etter.dns`

```
microsoft.com      A   209.92.24.80
*.microsoft.com    A   209.92.24.80
```

The signature used to model the problem was described using 3 network packet variables, which, when the problem gets solved, if the attack is found on the network traffic source, those variables will be assigned to packets found in the network traffic source which respect the constraints used to model the problem. This signature was modeled in the same way for both static and sliding network traffic window versions of NeMODe.

Static network traffic window

When using a static network traffic window as the network traffic source for NeMODe, two traffic logs were used, composed by 400 and 100 packets, both containing 10 fake DNS replies, spread over the entire log.

Sliding network traffic window

Using the sliding network traffic window NeMODe, we used a network traffic source composed of 400 packets, containing 10 fake DNS replies. This network situation was tested with several window sizes, using sizes of 10, 20, 30, and 60 packets, and changing the network speed in order to see how the system reacts to such changes.

7.2.5 DHCP spoof

To perform the detection of the DHCP spoof attack in NeMODe we decided to create network traffic log files with the help of `tcpdump` while a computer designated as a target host was under a DHCP spoof attack, which was achieved with the help of `Ettercap` [115], a tool capable of sniffing live network traffics and performing some "man in the middle" attacks.

`Ettercap` was used as in Listing 82 to produce the DHCP spoof attack. `10.10.10.59` is the victim host IP address, and `10.10.10.254` the legitimate DHCP server of the network which the host victim is connected to.

Listing 82 Performing a DHCP spoof attack

```
ettercap -i eth0 -T -M \
              dhcp:10.10.10.59/255.255.255.0/10.10.10.254
```

Both `tcpdump` network traffic log files were used as network traffic source for all experiments, either with Gecode, Adaptive Search or MiniSat, in any of the available architectures, x86 for Gecode and Adaptive Searchand MiniSat, and Cell/B.E. for Adaptive Search.

The signature used to model this attack was composed by 3 network packet variables, so, a solution to a problem representing a DHCP spoof attack, if it exists, is a subset of the network traffic log: 3 network packets satisfying all constraints used to model the signature of the attack. This signature was used and modeled in the same way in both the static and the sliding window versions of NeMODe.

Static network traffic window

Performing the detection of the DHCP spoof attack with NeMODe using a static network traffic window, we used two log files, one composed of 400 network packets and another of 100 network packets, both containing 10 fake DHCP replies, simulating a DHCP spoof attack.

Sliding network traffic window

To detect this attack using NeMODe with a sliding network traffic window, the network traffic source was composed of 400 packets, containing 10 fake DHCP replies to DHCP queries made by the target victim. This network traffic source was used with different network traffic window sizes, using sizes of 10, 20, 30, and 60 network packets, and also, changing the update time of the network traffic window, simulating the speed of the network traffic, so that we could understand the behavior of NeMODe with the variation of these parameters.

7.2.6 ARP poisoning

The network traffic log files used as the network source to detect an ARP poisoning attack, were generated by `tcpdump`, while the victim host was being attacked with Ettercap, which was configured to perform ARP poisoning attacks.

Ettercap was used as in Listing 83 to perform the ARP poisoning attack, where `10.10.10.59` is the IP address of the host victim, and `10.10.10.254` is the IP address of the host victim's default gateway, which makes the victim host to see the attackers MAC address as the default gateway's MAC address, and the default gateway see the attackers MAC address as the victim host MAC address, forcing all communications between the two hosts to pass through the attacker.

Listing 83 Perform an ARP poisoning attack

```
ettercap -i eth0 -T -M ARP /10.10.10.254/ /10.10.10.59/
```

To model the signature of the ARP poisoning attack, we used 4 network packet variables. A solution to this problem is a set of 4 network packets, taken from the network traffic source, if the ARP poisoning attack exists in the network traffic. This signature was used and modeled with no modifications for both versions of NeMODe, the static and sliding network traffic window.

Static network traffic window

We used two traffic source log files as input for NeMODe while using a static traffic window to detect the ARP poisoning attack, one composed by 400 packets, and other composed by 100 packets, each one containing 10 distinct and fake ARP poisoning replies.

Sliding network traffic window

While using a sliding traffic window in NeMODe to perform the detection of an ARP poisoning attack, we used a traffic source composed of 500 packets, containing 10 ARP poisoning attacks, and used a sliding window over this traffic source with a size of 10, 20 and 50 packets, while varying the update time of the network traffic window, thus, simulating the network traffic speed, in order to understand how NeMODe behaves when changes affect these parameters.

7.3 Results

In this section we present the results in terms of performance obtained for the network situations presented in Section 7.2. The results are presented in two classes, the results of the static network traffic window NeMODe version, and the ones of the sliding network packet window NeMODe version. We also make a distinction between the tests run on the x86, and the ones run on Cell/B.E.

For the experiments using a static network traffic window, we present the performance in terms of time to detect the first occurrence of the attack being looked for. As for the experiments using a sliding network traffic window, we present the results in terms of detection rate for at a given network traffic window update rate, simulating different network traffic speeds.

7.3.1 Static network traffic window - x86

We present the results of the experiments using the static network traffic window NeMODe version to detect each of the network situations described above.

Table 7.1 presents the required time(user-time), in miliseconds(ms), to find the desired network situation for each of the attacks presented in this work, using Gecode and

Table 7.1: Average time(in ms) necessary to detect the intrusions using Gecode and Adaptive Search

Intrusion	**Log Size**	**Case Size**	**GC (ms)**	**AS(ms)**
Port	400	52	127.3	674.99
scan	100	52	73.59	407.52
SSH	400	10	18.75	4.37
password	182	10	12.6	1.49
SYN	400	30	50.54	25.39
flood	100	30	40.62	6.09
DNS	400	3	6.94	5.78
Spoof	100	3	4.37	2.26
DHCP	400	3	8.26	1.09
spoof	100	3	3.98	0.46
ARP	400	4	23.125	18.04
Spoof	100	4	5.46	2.34

Adaptive Search x86 version, as the detection mechanisms. The presented results are the average execution times of 128 runs. We also present the "Log Size", the size of the network traffic window, and the "Case Size", the number of network packet variables in the signature used to model the problem.

Table 7.2 presents the same results as Table 7.1, but using MiniSat as detection mechanism. We present the total time needed to setup the problem, i.e.: generate the CNF rules which encode the problem, the necessary time to solve the problem after all CNF rules have been generated, and the total time necessary to obtain the a valid solution to the problem, including the CNF rules generation and problem solving. All times are presented in mili-seconds(ms) and are the average of 128 runs.

Table 7.2: Average time(in ms) necessary to detect the intrusions using MiniSat

Signature	**Log size**	**Case size**	**Setup(ms)**[1]	**Solve(ms)**[2]	**Total(ms)**[3]
Port	400	52	1838.12	1384.40	3222.52
scan	100	52	124.37	66.88	191.25
SSH	400	10	332.5	243.12	575.62
password	182	10	51.25	23.12	74.37
SYN	400	30	965.41	1311.23	2276.64
flood	100	30	103.12	71.25	174.37
DNS	400	3	106.32	28.07	134.39
spoof	100	3	6.95	0.70	7.65
DHCP	400	3	105.93	11.17	117.10
spoof	100	3	6.79	0.48	7.27
ARP	400	4	195.93	34.53	230.46
spoof	100	4	11.71	2.89	14.60

1 - MiniSat setup time; 2 - MiniSat solve time; 3 - SAT total time (Setup + Solve)

For MiniSat, we also present the size of each problem when encoded as a SAT problem, represented in Table 7.3, while varying the size of the network traffic window. For each problem is presented the number of **Setup clauses**, which models the variables of the problem and ensures the consistency of a solution, the number of **Model Clauses**, which are the clauses which actually model the problem, the number of **Total clauses** as well as the number of **SAT variables** necessary to model the problem.

Table 7.3: Size of each SAT problem: number of rules and variables

Signature	**Log size**	**Case size**	**#Setup clauses**	**#Model clauses**	**#Total clauses**	**#Total vars**
Port	400	52	4149652	82478	4232130	20800
scan	100	52	257452	20678	278130	5200
SSH	400	10	798010	12020	810030	4000
password	182	10	164720	5480	170200	1820
SYN	400	30	2394030	57972	2452002	12000
flood	100	30	148530	14472	163002	3000
DNS	400	3	239403	4107	243510	1200
spoof	100	3	14853	1063	15916	300
DHCP	400	3	239403	3587	242990	1200
spoof	100	3	14853	897	15750	300
ARP	400	4	319204	8004	327208	1600
spoof	100	4	19804	2004	21808	400

7.3.2 Static network traffic window - Cell/B.E.

The Adaptive Search back-end detection mechanism has a special characteristic over the other solvers: it has implementations in both x86 and Cell/B.E. architectures. Due to the availability of the Cell/B.E. implementation of Adaptive Search, we also decided to make some experiments using the Cell/B.E. In this Section, we presents these results..

Each network situation analyzed in this work was run on the IBM QS21 dual-Cell/BE blades, several times using different number of cores to understand the behavior of Adaptive Search running on Cell/B.E. under different circumstances.

As we have different results depending on the network situation and the number of cores used to solve the problem, we present two types of results using the Adaptive Search detection mechanism of NeMODe on the Cell/B.E.:

1. Average time to detect the first occurrence of the attack.
2. Speedup (versus single-core) obtained by using different numbers of Cell/B.E. cores.

The average time needed to detect the first occurrence of the attack is presented in Tables 7.4 and 7.5 , where **Sig.** represents the network situation being detected, **Log size**, the number of packets in the network traffic log, and **Case size**, the number of packet variables that buildup the signature used to model the desired network situation. We present the results using 1, 2, 4, 6, 8, 10, 12 and 16 cores. The results are presented in mili-seconds(ms) and they are the average of 128 runs.

Table 7.4: Average time(in ms) necessary to detect the intrusions with Adaptive Search on Cell/B.E. using 1, 2, 4 and 6 cores

Sig.	**Log size**	**Case size**	**1 core**	**2 cores**	**4 cores**	**6 cores**
Port	400	52	118555.07	74420.23	43568.35	27854.92
scan	100	52	5279.37	2447.81	1327.96	816.4
SSH	400	10	103.05	6.72	7.85	10.92
password	182	10	98.12	4.01	6.42	9.62
SYN	400	30	1143.12	801.87	458.98	351.79
flood	100	30	493.67	257.03	188.98	142.81
DNS	400	3	571.32	309.45	158.59	105.39
spoof	100	3	121.64	28.2	16.71	15.39
DHCP	400	3	91.47	5.45	7.64	10.76
spoof	100	3	88.51	3.59	4.92	8.82
ARP	400	4	175.15	63.82	34.76	27.42
spoof	100	4	96.64	10.31	9.84	11.48

Tables 7.6 and 7.7 presents the speedup obtained by using 1, 2, 4, 6, 8, 10 and 12 Cell/B.E. cores to solve each problem, using the results presented in Tables 7.4 and 7.5, and having as reference for calculating the speedups, the results obtained while using 1 core. **Sig.** represents the network situation being detected, **Log size**, the number of packets in the network traffic log size, and **Case size**, the number of packets that buildup the signature used to model the desired network situation.

Table 7.5: Average time(in ms) necessary to detect the intrusions with Adaptive Search on Cell/B.E. using 8, 10, 12, 14 and 16 cores

Sig.	**Log size**	**Case size**	**8 cores**	**10 cores**	**12 cores**	**14 cores**	**16 cores**
Port	400	52	22630.46	17063.67	13729.37	12196.95	10894.68
scan	100	52	638.43	577.96	386.09	351.09	310.39
SSH	400	10	13.85	16.85	19.55	22.52	25.17
password	182	10	12.27	15.72	18.75	21.3	23.65
SYN	400	30	290.78	277.1	255.85	228.04	224.06
flood	100	30	133.2	132.18	122.81	121.4	120.62
DNS	400	3	90.14	69.68	66.71	61.56	59.53
spoof	100	3	17.10	18.35	21.09	23.43	25.31
DHCP	400	3	15.45	18.57	21.25	23.81	26.31
spoof	100	3	14.14	16.4	17.65	22.26	23.67
ARP	400	4	27.34	27.5	28.75	30.46	32.42
spoof	100	4	14.21	16.71	19.68	22.03	24.14

Table 7.6: Speedup on using Adaptive Search in Cell/B.E. with 1, 2, 4 and 6 cores. Average of 128 runs

Sig.	**Log size**	**Case size**	**1 core**	**2 cores**	**4 cores**	**6 cores**
Port	400	52	1.00	1.59	2.72	4.26
scan	100	52	1.00	2.16	3.98	6.47
SSH	400	10	1.00	15.33	13.13	9.44
password	182	10	1.00	24.47	15.28	10.20
SYN	400	30	1.00	1.43	2.49	3.2
flood	100	30	1.00	1.92	2.61	3.46
DNS	400	3	1.00	1.85	3.60	5.42
spoof	100	3	1.00	4.31	7.28	7.90
DHCP	400	3	1.00	16.78	11.97	8.50
spoof	100	3	1.00	24.65	17.99	10.04
ARP	400	4	1.00	2.74	5.04	6.39
spoof	100	4	1.00	9.37	9.82	8.42

Table 7.7: Speedup on using Adaptive Search in Cell/B.E. with 8, 10, 12, 14 and 16 cores. Average of 128 runs

Sig.	**Log size**	**Case size**	**8 cores**	**10 cores**	**12 cores**	**14 cores**	**16 cores**
Port	400	52	5.24	6.95	8.64	9.72	10.88
scan	100	52	8.27	9.13	13.67	15.04	17.01
SSH	400	10	7.44	6.12	5.27	4.58	4.09
password	182	10	8.00	6.24	5.23	4.61	4.15
SYN	400	30	3.93	4.13	4.47	5.01	5.10
flood	100	30	3.71	3.73	4.02	4.07	4.09
DNS	400	3	6.34	8.20	8.56	9.28	9.60
spoof	100	3	7.11	6.63	5.77	5.19	4.81
DHCP	400	3	1.00	4.93	4.30	3.84	3.48
spoof	100	3	6.26	5.40	5.01	3.98	3.74
ARP	400	4	6.41	6.37	6.09	5.75	5.40
spoof	100	4	6.80	5.78	4.91	4.39	4.00

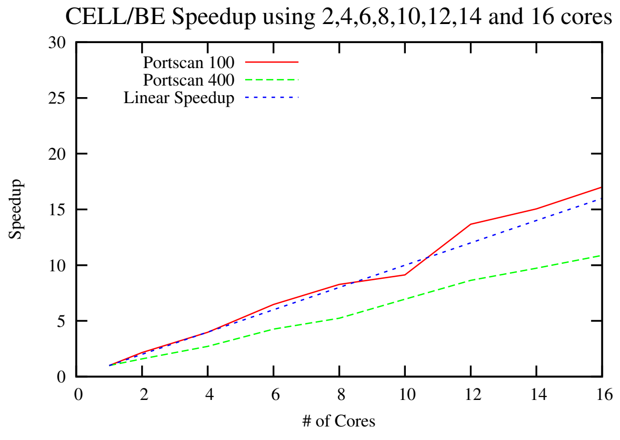


Figure 7.1: Portscan - speedup

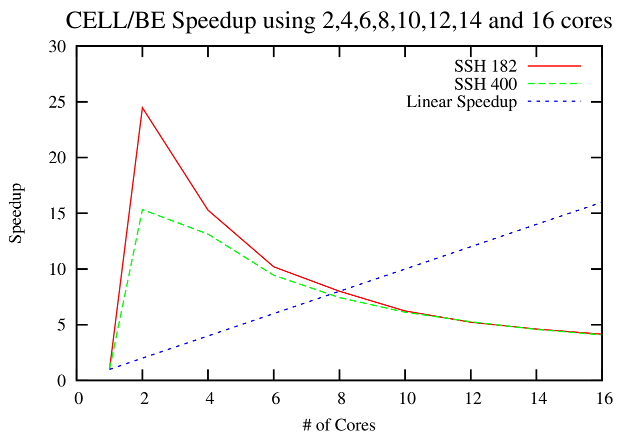


Figure 7.2: SSH password brute-force - speedup

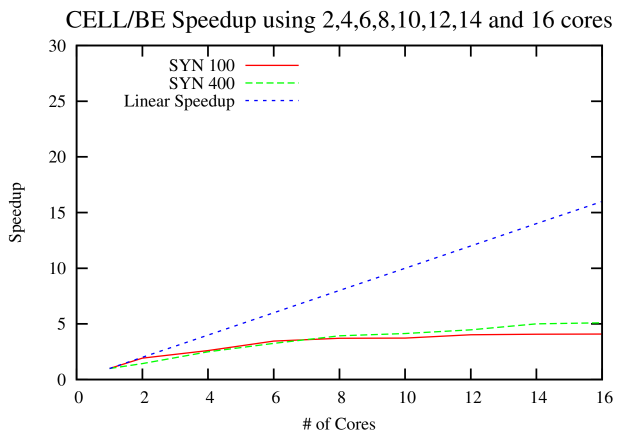


Figure 7.3: SYN flood - speedup

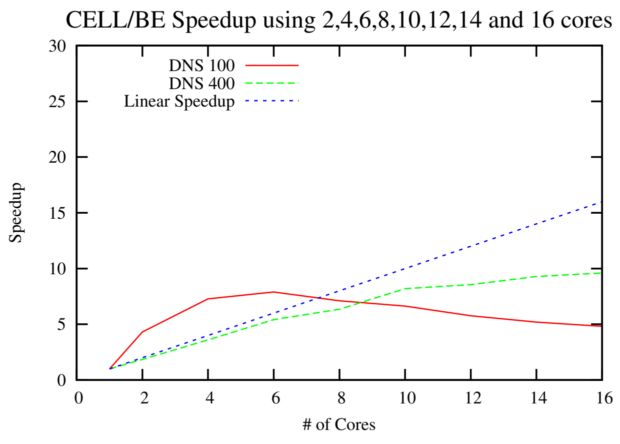


Figure 7.4: DNS spoof - speedup

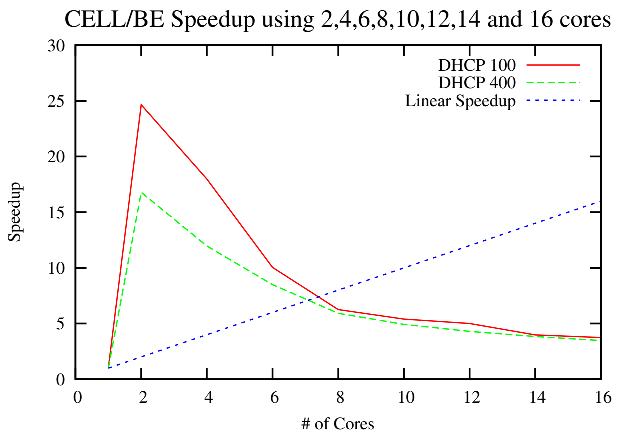


Figure 7.5: DHCP spoof - speedup

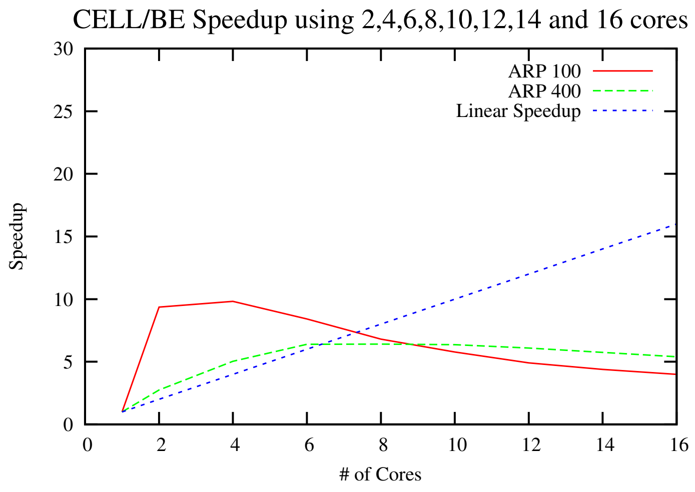


Figure 7.6: ARP poisoning - speedup

Figures 7.1-7.6 present the speedup obtained for all network situations while using different number of cores on the Cell/B.E., while using network packet windows of different sizes; Fig. 7.1 for the Portscan; Fig. 7.2 for the SSH password brute-force; Fig. 7.3 for the SYN flood; Fig. 7.5 for the DHCP spoof; Fig. 7.4 for the DNS spoof and Fig. 7.6 for the ARP poisoning. In each Fig. we also present a *linear speedup* line for reference.

7.3.3 Sliding network traffic window

The sliding network traffic window was only implemented in the x86 version of Adaptive Search. All results presented in this Section are related to using Adaptive Search with a sliding network traffic window.

The results of using a sliding network traffic window in NeMODe are presented in terms of detection rate, instead of the time necessary to detect the first occurrence of the desired network situation, as opposed to the time necessary to detect the first occurrence of the desired attack.

For a better understanding of the results, we represent them as charts, rather than to tables. Since we present the results for different network traffic speeds, it would make results hard to understand if represented in tables. The results are presented in two unit scales for a better reading:

1. Detection rate(%), while varying the network traffic window update time
2. Detection rate(%), while varying the network traffic speed

DNS spoof

Fig. 7.7 presents the detection rate of DNS spoofing attacks, while varying the time interval between the arrival of each network packet (in micro-seconds). The chart presents the results using a sliding window of 10, 20, 30, and 60 network packets, over a network traffic log of 400 network packets.

Fig. 7.8 presents the same results as Fig. 7.7, but considering the network traffic bandwidth, measured in Mbit/s, instead of the network packet interval. The results presented in both Figures are the average of a of 100 runs.

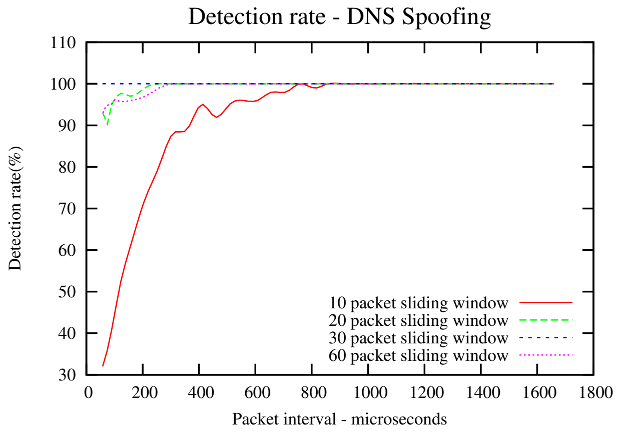


Figure 7.7: DNS spoof - Detection rate.

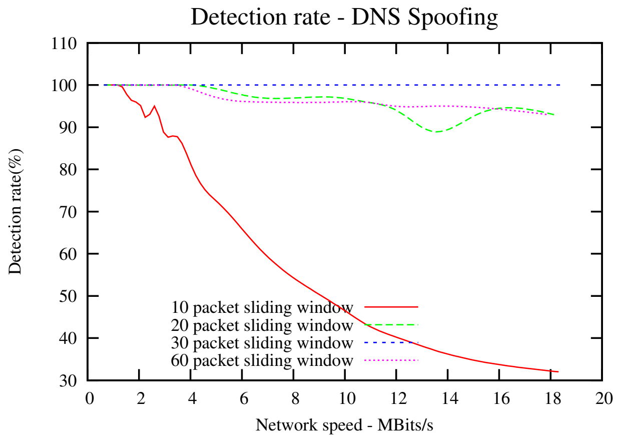


Figure 7.8: DNS spoof - Detection rate.

DHCP spoof

Fig. 7.9 represents the detection rate of DHCP spoofing attacks, while varying the time interval between the arrival of each network packet(in micro-seconds). We present the results for a sliding window of 10, 20, and 60 network packets, over a network traffic log of 400 network packets

Fig. 7.10 presents the same results as Fig. 7.9, but considering the network traffic bandwidth, measured in Mbit/s, instead of the network packet interval. The results presented are the average detection rate of a total of 100 runs.

SSH password brute-force

Figure 7.11 represents the detection rate of the SSH password brute-force attack, varying the time interval between the arrival of network packet(in micro-seconds). We

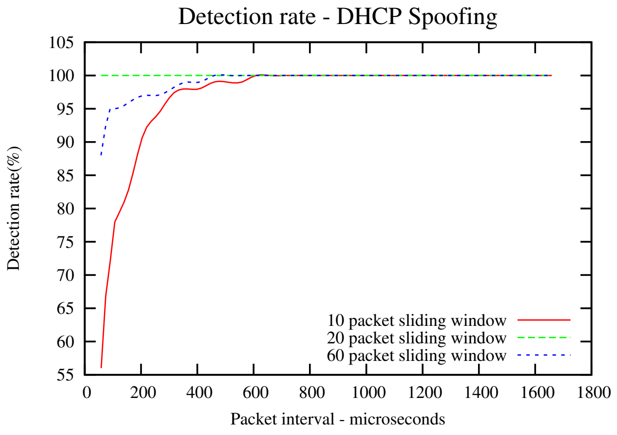


Figure 7.9: DHCP spoof - Detection rate

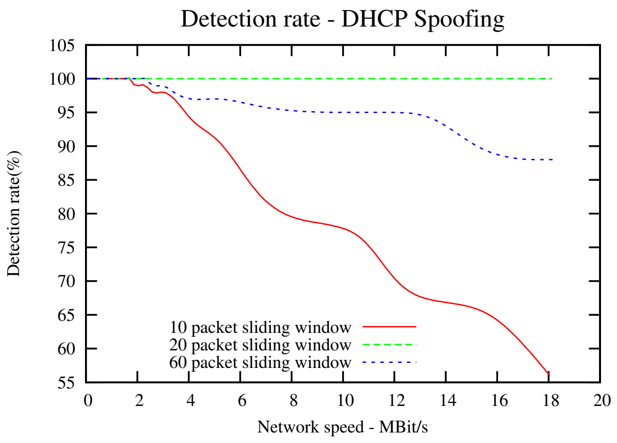


Figure 7.10: DHCP spoof - Detection rate

present the results of using a sliding window of 10, 20, 30, 100 and 300 network packets, over a network traffic log of 3000 network packets.

Figure 7.12 presents the same results as in Fig. 7.11 but considering the network traffic bandwidth, measured in Mbit/s, instead of the network packet window interval time. Both results presented are the average detection rate of 100 runs.

ARP poisoning

Figure 7.13 represents the detection rate of the ARP poisoning attack, varying the time interval between the arrival of network packet(in micro-seconds). We present the average results using a sliding window of 10, 20 and 50 network packets, over a network traffic window of 500 network packets.

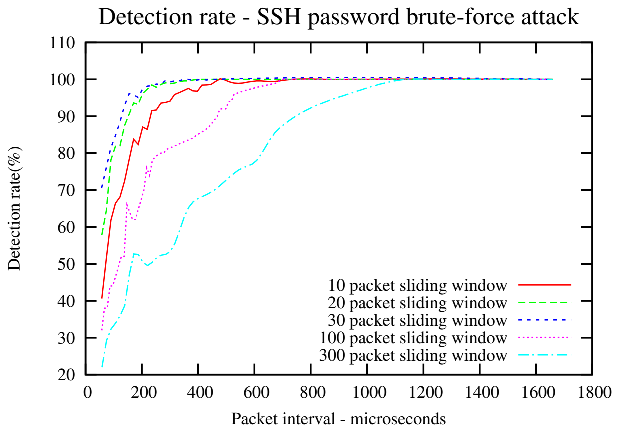


Figure 7.11: SSH password brute-force - Detection rate

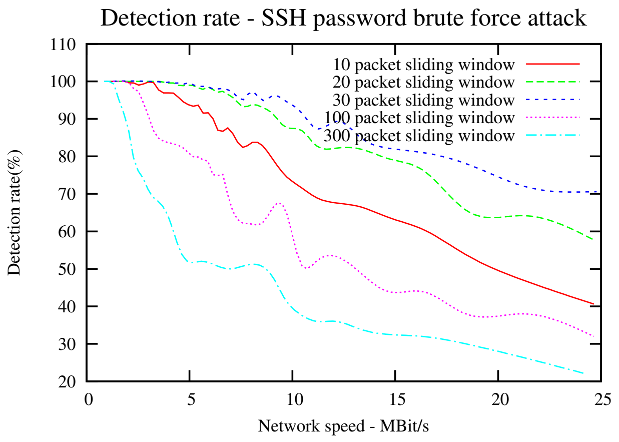


Figure 7.12: SSH password brute-force - Detection rate

Figure 7.14 presents the same results as in Fig. 7.13, but considering the network traffic bandwidth, measured in Mbit/s, instead of the network packet interval. The results for both Figures are the average of 100 runs.

7.4 Evaluation

In this section we evaluate all the approaches to the back-end detection mechanisms of NeMODe, while detecting all network situations described in this work. We do this using the static network traffic window with Gecode, Adaptive Search(x86 and Cell/B.E.) and MiniSat. We also evaluate the sliding network traffic window version of NeMODe running in the x86. We evaluate NeMODe not only regarding the experimental results, but also regarding the NeMODe specification DSL. Last, we compare our approach to other approaches to Network Intrusion Detection, in particular against Snort.

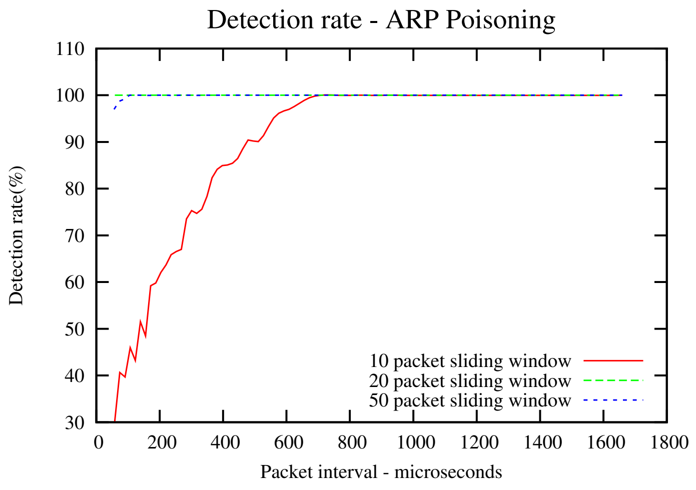


Figure 7.13: ARP poisoning - Detection rate

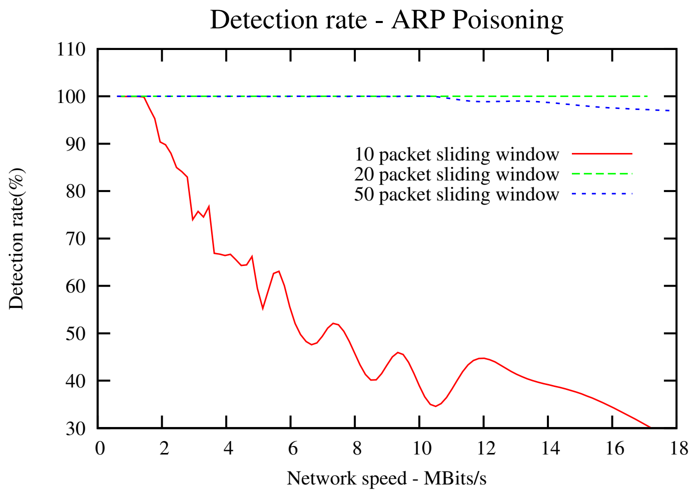


Figure 7.14: ARP poisoning - Detection rate

7.4.1 Static network traffic window - x86

The experimental results described in Sec. 7.3.1 reveal a significant performance variation, depending on the problem and the solver being used. Carefully analyzing Tables 7.1 and 7.2, we get to the conclusion that when the size of the problem grows, Gecode performs better than Adaptive Search and MiniSat, but when the size of the problem is smaller, Adaptive Search beats both Gecode and MiniSat in a large scale.

If we consider only the time taken by MiniSat to solve the problem, instead of the total time including the clause generation and solving process, MiniSat can be almost as fast as Gecode, and, in some cases, the most simple ones, it can be even faster than Adaptive Search. This consideration makes sense because the generation of the clauses which encode the problem as a SAT problem is not directly related to solving the problem, but to the description of the problem, and also, the rules used to setup the variables which represent the major number of rules, as can be seen in Table 7.3, can be shared among many network situations. These *setup* rules only depend on the size of the network traffic window and the number of network packets used to model the signature of the desired network situation, which can be pre-generated, thus allowing to discard most of the time consumed in generating the CNF clauses.

Although Adaptive Search performs better in most cases, there are some exceptions, such as the Portscan case. In this case, Gecode presents much better results, explained by the high number of variables used to model its signature, thus confirming that when the size of the problem is larger, mainly due to the number of packet variables used to model the desired situation, Gecode tends to perform better.

While making a broader analysis of Tables 7.1 and 7.2, we can conclude that all solvers are affected by both the size of the network traffic window and the size of the intrusion signature, some more so than others. Gecode is the one least affected by changes to the size of the problem, while MiniSat is the most affected, mostly due to the exponential increase of number of rules used to model the problem. The performance is most sensitive to the size of the signature, as can be seen in both Portscan and SYN flood, the network case scenarios with larger signatures, thus the ones that perform worst.

Considering the best results obtained, we achieved performance figures which allow us to perform live network intrusion detection in the future using this approach.

7.4.2 Static network traffic window - Cell/B.E.

The results presented in Sec. 7.3.2 shows that Adaptive Search Cell/B.E. version presents a behavior very similar to the Adaptive Search x86 version, but with some significant differences in performance, being overall slower that the x86 version.

The results presented in Tables 7.4 and 7.5 show that performance is affected in a greater scale than in the x86 version of Adaptive Search when the size of the problem increases, reaching very low figures with the Portscan, the case scenario with more network packet variables in its signature.

Besides Portscan, the performance of Adaptive Search in Cell/B.E. is affected by the size of the network traffic window in the same way as in the x86 version, being more affected by the size of the network signature than by the size of the network traffic window.

Adaptive Search was tested on Cell/B.E. using different number of cores, from 1 to 16 cores, on problems of different sizes and complexities, revealing interesting results in terms of possible parallel speedup to solve each problem.

Looking at Tables 7.4 and 7.5 and Figures 7.1-7.6 we can see there are two types of behaviors; on one hand, the speedup is proportional to the number of cores used to solve the problem; and on the other hand, it rises up to a certain number of cores, and then starts to decay when increasing beyond that.

The most complex and large problems are the ones which get closer to a linear speedup, always increasing the performance of Adaptive Search with the number of cores.

As for the smaller and simple problems, the speedups have a peculiar shape, with the increase of number of cores, there is a huge increase in performance when using only 2 cores, but then, it starts to decay up to the total of the 16 cores. This behavior may be explained with noise introduced by very small execution times, disguising the real speedup values.

Despite the difference of the results, we obtain on all network situations interesting speedup figures when comparing to the single core performance figures.

7.4.3 Sliding network packet window

The results presented in Sec. 7.3.3 are very promising, as we obtain a good detection rate at reasonably high network speeds while using Adaptive Search with a sliding network packet window.

Looking closer at the results, it is possible to see that the detection rate is low when the time interval between the arrival of two network packets is very small, i.e. with a high network speed. This characteristic is mostly noted while using a small network traffic window, which presents the lowest detection rate. This is explained by the fact that the packets are in the window for a very short time, and the solver is not capable of detecting all situations while the packets are still in the window.

With the increase of the window size, the detection rate at higher speeds also increases, but only up to a certain limit, starting to decline when the window becomes too large. The increase of the detection rate is explained by the fact that the network packets remain more time in the window, thus allowing the solver a better chance in looking for a solution. Still, when the window reaches a certain limit, the detection rate starts do decay when using higher bandwidth rates. This is due to the solver taking too long to analyze the larger window, thus missing some attacks.

The DNS spoofing attack gets a very low detection rate with high network speeds and using a window of 10 packets, reaching a detection rate of 100% only at about 1 MBit/s. Using a 30 packet window, we get a detection rate of 100% from the beginning, with a bandwidth 18MBit/s. The detection rate keeps constant up to a of 60 packets, when it starts to decay. With these results, we get to the conclusion that using a window with a size between 30 and 60 network packets, we get close to a detection rate of 100% on a 18MBit/s network.

The DHCP spoofing attack presents a behavior very similar to the DNS attacks, but manages to obtain a detection rate of 100% on higher speeds and smaller network windows, being capable of a detection rate of 100% at 18MBit/s, using a window of 20 network packets. Also, this rate gets constant up to a window of 60 packets, when it starts to decay. The conclusion that we can take from this results is similar one we learned from DNS spoofing attack: we are able to reach a detection rate of almost 100% using a window with a size between 20 and 60 packets with a bandwidth of 18MBit/s. The DHCP spoofing attack presents slightly better performance results due to the simpler signature.

The SSH password brute-force attack also presents a similar behavior as the other attacks, but it has more difficulty in reaching a 100% detection rate, which can be explained by the fact that the network signature of the attack is more complex, using more time-consuming constraints.

Although it doesn't present a performance as good as the other examples, still, it manages to get a detection rate of 100% up to about 5MBit/s while using a window of 30 network packets. This detection rate is constant up to a window of about a window of 100 network packets, when it starts to decay. Considering a bandwidth of about 10Mbit/s, we manage to get a detection rate of 80%, which is a very good result.

The results obtained by the SSH password brute-force attack are very encouraging, not only due to the detection rate at considerable network speeds, but mainly because we are able to detect intrusions that spread across a network window larger than the window being used by the solver, allowing us to detect network attacks which are *diluted* in time, thereby providing a good basis to perform intrusion detection on live network traffic.

The ARP poisoning attack presents a behavior very similar to the SSH password brute-force attack, but needs a slower network speed to detect reach a detection rate of 100% when using a smaller network traffic window. When the network traffic window is increased, it reaches a 100% detection rate faster than the SSH password brute-force attack.

It gets a detection rate of 100% with a network packet window of 20 packets at 17Mbit/s. This detection rate of 100% is constant up to a network packet window of 50 packets, when it starts to decay.

7.4.4 NeMODe Specification DSL

The DSL introduced for NeMODe, turns out to be expressive and powerful, allowing an easy description of all network intrusions analyzed in this work and generating recognizers able to detect the desired network situation.

Although other IDSs like Snort could detect some of the attacks presented in this work, they don't allow to describe the problems with the expressiveness used by NeMODe or even relate the several packets that make part of the attack. This topic is further discussed in Sect. 7.4.5.

7.4.5 NeMODe vs Snort

Most attacks presented in this work can be detected by tools which use different approaches to Network Intrusion Detection. These usually present limited ways to describe the desired network situation, if possible to describe them at all.

These tools, usually cannot describe or detect attacks that spread across several network packets, and when they do, the description of such attacks is very limited, resorting to preprocessors built with the single purpose of detecting a specific network situation.

Although most cases which we have experimented in NeMODe can be detected by systems such as Snort, they cannot be or modeled in a descriptive way as in NeMODe, making a direct comparison difficult, nevertheless, we decided to experiment on the same network situations in Snort using the same network traffic logs in order to obtain some experimental results.

We now describe how these attacks can be modeled in Snort and present the results obtained.

Portscan

A Portscan attack can be detected in Snort, allowing several ways to detect this type of attack, either by describing the situation in a limited way, or, by using of built-in pre-processors built with the single purpose of detecting Portscan attacks.

Usually specific preprocessors designed to detect portscans are required for Snort to detect this type of attacks, not being possible to *program* a portscan attack using the Snort rule language. Snort provides several of these preprocessors: `portscan`, `portscan2`, `flow-portscan` or `sfPortscan`.

In our experiments we selected the `sfPortscan` [28] preprocessor which allows Snort to detect various types of portscans and portsweeps. The `sfPortscan` was used as in Listing 84.

Listing 84 Using `sfPortscan` Snort module

```
preprocessor sfportscan: proto  { all } \
                         memcap { 10000000 } \
                         sense_level { high } \
                         logfile { portscan.log }
```

SSH password brute-force

It is possible to use Snort to describe and detect SSH password brute-force attacks by monitoring a large amount of SSH connections from the same source in a short period of time: this is achieved in a very limited way, resorting to built-in filters which impose a limit of network packets in given amount of time.

Listing 89 represents the rule which we used to detect this attack in Snort. It looks for packets going to port 22, where the SSH service is running, which have message "SSH-" in it's payload. If there are 5 of these network connection from the same source in the interval of 60 seconds or less, then we could be under an SSH password bruteforce.

Listing 85 SSH password bruteforce Snort rule

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 22 \
(msg:"Possible SSH brute force attempt"; \
flow:to_server,established; \
threshold:type threshold, track by_src, count 5, seconds 60; \
content:"SSH-"; offset: 0; depth: 18;)
```

Although this rule is effective in some cases, it doesn't make use of real relations between several network packets, making the description counter-intuitive and hard to express.

In our experiments we obtained a detection rate of 100%, mostly because the rule was specifically tailored to match the attacks in the network traffic logs being analyzed. If the attack is done in a slightly different way, the signature can fail to detect the attack.

SYN flood

The detection of a SYN flood attack in Snort is possible by creating a rule based in the `threshold` option which allows to count the number of packets with the `SYN` flag set over a small amount of time, which is basically the same approach we took in NeMODe, but in a more expressive way.

Listing 86 presents the rule which detects a SYN flood in Snort, where we look for 50 tcp packets in 5 seconds with the SYN flag set.

Listing 86 SYN flood Snort rule

```
alert tcp any any -> any any (msg:"Syn Flood"; \
flow: stateless;  flags:S,12; threshold: type threshold, \
track by_src, count 50, seconds 5; sid:10002;)
```

Although the use the `threshold` option allows to count the network packets over a certain period of time, it doesn't allows to create real relations between the packets involved in the attack.

DNS spoof

Snort provides some built-in rules which allows the detection of some DNS spoofing attacks, but they do so by analyzing only specific properties in the headers and payload of the network packets, not being able to relate several packets to model the problem. More specifically, the Snort ID 253 rule and Snort ID 254 rule, check for DNS replies with a `TTL` of 1 minute and no authority [116], which are usually characteristics of a DNS spoofing attempt.

Listing 87 presents both Snort ID 253 and Snort ID 254 rules we used to detect the DNS spoof attack.

This set of rules were used “as is” in the Snort rule data-base, having the specific purpose of detecting DNS spoof attempts. In fact, these rules can detect some DNS spoofing attempts, but in many cases, depending on the tools used to perform the attack, they will fail. In particular, this set of rules were not able to detect any of our DNS spoofing attempts made with the help of `ettercap`.

DHCP spoof

For the DHCP spoofing attacks, Snort does not provide a ready to use preprocessor or rule. One of the only ways to detect a DHCP spoofing in Snort is to monitor for

Listing 87 DNS spoof Snort rule

```
alert udp $EXTERNAL_NET 53 -> $HOME_NET any (msg:"DNS SPOOF \
query response PTR with TTL of 1 min. and no authority";\
content:"|85 80 00 01 00 01 00 00 00 00|"; \
content:"|C0 0C 00 0C 00 01 00 00 00|<|00 0F|"; \
classtype:bad-unknown; sid:253; rev:4;)

alert udp $EXTERNAL_NET 53 -> $HOME_NET any (msg:"DNS SPOOF \
query response with TTL of 1 min. and no authority"; \
content:"|81 80 00 01 00 01 00 00 00 00|"; \
content:"|C0 0C 00 01 00 01 00 00 00|<|00 04|"; \
classtype:bad-unknown; sid:254; rev:4;)
```

DHCP replies from hosts which are not a legitimate DHCP server [117]. This can actually detect some DHCP spoofing attacks, but can be easily evaded if the attacker also spoofs its IP address.

Listing 88 presents the rules necessary to detect this attack. First, we specify that we should accept all DHCP replies from the legitimate DHCP server with IP address `192.168.1.254`. Then we specify that any DHCP reply from other host will be considered a DHCP spoofing attempt.

To use this specific rule, Snort had to be run with the "–o" option, which reverses the order how the rules are read, considering "pass" statements before "alert" statements.

Listing 88 DHCP spoof Snort rule

```
pass udp 192.168.1.254 67 -> any 68
alert udp any 67 -> any 68 (msg: "Rogue DHCP server..."; sid:1)
```

This set of rules is effective to detect DHCP spoofing attempts, but they can fail if a new DHCP server is added to the network and the rule is not updated, leading to a large amount of false positives.

However, if an attacker decides to spoof its IP address, it can easily evade the detection.

ARP poisoning

Snort is also capable of detecting ARP poisoning attacks, but only if using the `arpsoof` preprocessor [28], which monitor for ARP packets against a user supplied ARP table containing valid `(MAC address, IP address)` pairs in the given network, which is hard to maintain when there are changes in the network.

Listing 89 presents the rules which were used to detect an ARP poisoning in Snort where it can be seen some known IP/MAC addresses combinations in the given network. If a different combination of the same IP/MAC addresses is found then we could be under an ARP poisoning attack. In this example we present only 3 IP/MAC addresses combination to simplify the example.

Listing 89 ARP poison Snort rule

```
preprocessor arpspoof
preprocessor arpspoof_detect_host: 192.168.1.70 \
                                            48:5d:60:72:d4:75
preprocessor arpspoof_detect_host: 192.168.1.90 \
                                            08:00:27:94:2c:46
preprocessor arpspoof_detect_host: 192.168.1.254 \
                                            00:24:17:70:26:EC
```

This Snort preprocessor is effective for detecting ARP spoofings, achieving a 100% detection rate (on our test runs). Still, it requires a large amount of maintenance if there are many hosts in the network to monitor, becoming unusable on large networks.

Snort results

To perform these experiments, we used Snort in *replay mode*, enabling the use of *tcpdump* or *pcap* files, allowing the use of the same network traffic log files used in NeMODe.

Snort was run as in Listing 90, where `file.pcap` is the tcpdump file with the network traffic, `file.conf` the configuration file of Snort. Parameter `-b` forces Snort to log packets in a binary tcpdump formatted file, making Snort faster.

Listing 90 Running Snort with a `pcap` file

```
snort -r file.pcap -c file.conf -b
```

Table 7.8 presents the results obtained by Snort. We present the time spent by the specific rule or preprocessor which detects the specific time; the time spent by other Snort preprocessors, also necessary for the detection of the specific rule, in microseconds; the total time needed to process the network traffic log and detect the specific attack. We also present the detection rate of Snort. We present the results for each network situation with log files of different sizes, the same used to evaluate NeMODe.

In all experiments with Snort we removed all unnecessary rules and preprocessors which were not relevant for the specific attack being detected.

Looking to the results obtained by Snort in Table 7.8, it is possible to conclude that Snort performs better than NeMODe in terms of time necessary to process the network traffic and detect the specific intrusion.

Table 7.8: Snort results (in μs)

Signature	**log size**	**specific rule** (μs)	**preproc.** (μs)[1]	**total time** (μs)	**detection rate (%)**
Port	400	755	2121	2876	100
scan	100	215	462	677	100
SSH	400	24	1425	1449	100
password	100	13	619	632	100
SYN	400	131	2071	2202	100
flood	100	37	807	844	100
DNS	400	n.a.	1042	1042	0
spoof	100	n.a.	329	329	0
DHCP	400	34	1020	1054	100
spoof	100	24	354	378	100
ARP	400	2	1018	1020	100
spoof	100	1	353	354	100

1 - preprocessor(μs)

7.5 Conclusion

In this Chapter we presented and evaluated the results obtained while using all versions of NeMODe to solve all network situations described over this entire Chapter.

The results obtained present valuable information, showing that we can use declarative programming, more specifically, constraint programming methodologies, to perform Network Intrusion Detection.

Although all these tests were performed with network traffic log files, the results obtained show that we are ready to start performing Network Intrusion Detection on live network traffic, a conclusion which is reinforced by the results obtained while using the sliding network traffic window version of Adaptive Search in the x86, which achieved a high rate of detection for the desired network situations, at reasonable, although simulated, network traffic speeds.

The results obtained in all tests, demonstrate that one limiting factor to obtain good performance results is the size of the problem, both the size of the network traffic window and the size of the network intrusion signatures, but, while using a sliding window, this limitation is not as important, since there is no need to use network traffic window such a large window as in the static version, since the window is constantly being updated.

While comparing our work to other approaches to Network Intrusion Detection, more specifically Snort, we get to the conclusion that NeMODe is less efficient in terms of detection time. Still, some of the network attacks have to be modeled using specific tailored preprocessors, not allowing to *program* the attacks using the Snort rule based language at all. Also, when it is possible to model the specific attacks in the Snort rule based language, the description is often limited, awkward and counter-intuitive.

In some cases, as in the DNS spoofing, Snort provides ready to use rules, but they were unable to detect the specific attack in our network traffic log files, mostly because the lack of generality in specific Snort rules. Because of this, if the attack deviates slightly from the rule specification, it will not be detected.

For other attacks, such as for the ARP spoofing, Snort provides very efficient preprocessors, but they require a lot of maintenance, becoming unusable in large scale networks.

Chapter 8

Conclusions and Future Work

This Chapter presents the conclusions of this work as well as some future work to be done.

8.1 Conclusions

We presented NeMODe, a declarative approach to Network Intrusion Detection Systems providing network intrusion detection mechanisms based on several Constraint Programming approaches. NeMODe also provides a declarative Domain Specific Language allowing for an easy modeling of the network intrusions to be detected.

The system presented in this work allows for a declarative description of network situations which spread across several network packets, and based on this description, generate several detection mechanisms based on different Constraint Programming methodologies, namely propagation based solvers, Constraint-Based Local Search and Boolean Satisfiability Solvers.

In this work, we have demonstrated that we can model Network Intrusion Detection signatures using these methodologies, in order to perform the detection of signature-based attacks.

Each approach to Constraint Programming used in this work poses specific problems in modeling a Network Intrusion Detection problem. Constraint Based solvers allow a relatively easy modeling of the problem. On the other hand, it is quite difficult to efficiently restrict the domain of the network packet variables, so that a solution to the problem be composed exclusively of packets found in the actual traffic, without sacrificing performance.

Modeling in Constraint-Based Local Search is easy in the sense that the constraint specification is straightforward, but since AS is quite sensitive to the heuristic used to model the problem, it turns out difficult to figure out the best heuristics, thus, making the modeling of Network Intrusion Detection problem in Adaptive Search quite complex. In AS, ensuring that the domain of the network packet variables is correct comes for free, due to the way we modeled the network packet variables.

Modeling a Network Intrusion Detection problem in SAT is quite linear after all constraint functions have been implemented in order to encode the necessary CNF rules. The major problem in SAT is the modeling of such constraint functions, which generate quite large and complex sets of Boolean rules. In MiniSat, due to the specificity of the SAT problems, we don't need to worry about the domain of the network packet variables, since it is taken care of by the encoding the of problem.

The Domain Specific Language provided by NeMODe proved capable of a descriptive modeling of network intrusions and adequate to generate detections mechanisms for all Constraint Programming approaches used in this work, enabling a parallel detection process, using all the mechanisms concurrently, in search for the faster solution.

The use of a sliding network traffic window allows to simulate the analysis of real time network traffic, as well as detecting network situations that are *wider* than the network window used by the solver, allowing to detect network attacks that span a considerable time interval. Also, this window can be easily adapted to work on a live link, and has enough performance to do Intrusion Detection in real time.

The architecture used in the "sliding window" version uses two threads: one for feeding the solver with packets and another to solve the problem. This revealed to be a right decision, mostly because with this approach we can easily change the network traffic source from `tcpdump` log files to other type of log files or even to live traffic, using either `libpcap` or another approach to capture network traffic. Also important, this approach does not limit the network bandwidth. Even if the solver is not able to analyse all traffic, it is able to detect some attacks.

Although all tests were performed with network traffic log files, the results obtained show that we are ready to start performing Network Intrusion Detection on live network traffic, a conclusion which is reinforced by the results obtained while using the sliding network traffic window version of Adaptive Search in the x86, which achieved a high rate of detection for the desired network situations, at reasonable, although simulated, network traffic speeds.

The results obtained in all tests, demonstrate that one limiting factor to obtain good performance results is the size of the problem, both the size of the network traffic window and the size of the network intrusion signatures, but, while using a sliding window, this limitation is not as important, since there is no need to use a network traffic window as large in the static version, since it is constantly being updated.

While comparing our work to other approaches to Network Intrusion Detection, more specifically Snort, we get to the conclusion that NeMODe is less efficient in terms of detection time. Still, some of the network attacks have to be modeled using specific tailored preprocessors, not allowing to *program* the attacks using the Snort rule based language at all. Also, when it is possible to model the specific attacks in the Snort rule based language, the description is often limited, awkward and counter-intuitive.

The performance figures achieved by all detection mechanisms implemented is preliminary, due to the need of modeling the network as complex combinatorial problems. Still, we do believe the performance can be improved either by modeling the problems in different ways or by fine tuning the solvers to the needs of our problems.

8.2 Future Work

As for future work, there is plenty to be done in order to improve the work presented in this thesis. The most important steps to take in a near future are:

1. Look for better Adaptive Search heuristic functions to improve the Constraint Local Based Search detection mechanism.
2. Experiment different encodings of the network intrusions as SAT problems to improve MiniSat detection mechanism.
3. Incorporate the "static" part of a SAT problem into MiniSat in order to avoid the processing of static rules, thus improving the performance of MiniSat on large scale problems.
4. Improve the Adaptive Search "Sliding network traffic window" to work with real live network traffic, thus allowing to monitor networks in a real environment.
5. Adapt the Gecode and MiniSat back-ends to work with a "Sliding network traffic window", enabling them to monitor real live network traffic.
6. Include more network entities and properties to allow the detection of a broader range of network situations.
7. Improve the network intrusion specification language to allow the description of a wider range of network intrusions.
8. Improve parallel specification and solving.

References

[1] Pedro Salgueiro and Salvador Abreu. Modeling Distributed Network Attacks with Constraints. In F. Brazier, Kees Nieuwenhuis, Gregor Pavlin, Martijn Warnier, and Costin Badica, editors, *Intelligent Distributed Computing V*, volume 382 of *Studies in Computational Intelligence*, pages 203–212. Springer Berlin / Heidelberg, 2012. ISBN 978-3-642-24012-6. URL `http://dx.doi.org/10.1007/978-3-642-24013-3_20`.

[2] Pedro Salgueiro, Daniel Diaz, Isabel Brito, and Salvador Abreu. Using Constraints for Intrusion Detection: the NeMODe System. In Ricardo Rocha and John Launchbury, editors, *PADL'11 - Thirteenth International Symposium on Practical Aspects of Declarative Languages*, volume 6539 of *Lecture Notes in Computer Science*, pages 115–129, Berlin, Heidelberg, January 2011. Springer-Verlag. ISBN 978-3642183775.

[3] Pedro Salgueiro and Salvador Abreu. Network Intrusion Detection with Constraints. In A. Santos, P. Carvalho, M. Nicolau, and et al., editors, *Actas da 10ª Conferência sobre Redes de Computadores(CRC 2010)*.

[4] Pedro Salgueiro and Salvador Abreu. On using Constraints for Network Intrusion Detection. In Luís S. Barbosa and Miguel P. Correia, editors, *Actas do INForum 2010 - II Simpósio de Informática*, pages 637–648, Universidade do Minho, Braga, Portugal, 2010.

[5] Pedro D. Salgueiro and Salvador P. Abreu. A DSL for intrusion detection based on constraint programming. In Oleg B. Makarevich, Atilla Elçi, Mehmet A. Orgun, Sorin A. Huss, Ludmila K. Babenko, Alexander G. Chefranov, and Vijay Varadharajan, editors, *Proceedings of the 3rd international conference on Security of information and networks*, SIN '10, pages 224–232, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0234-0. doi: http://doi.acm.org/10.1145/1854099.1854145. URL `http://doi.acm.org/10.1145/1854099.1854145`.

[6] Pedro Salgueiro and Salvador Abreu. Network Monitoring with Constraint Programming: Preliminary Specification and Analysis. In Salvador Abreu and Dietmar Seipel, editors, *Applications of Declarative Programming and Knowledge Management*, volume 6547 of *Lecture Notes in Computer Science*, pages 223–238. Springer Berlin / Heidelberg, 2011. ISBN 978-3-642-20588-0. 18th International Conference, INAP 2009, Évora, Portugal, November 2009, Revised Selected Papers.

[7] Pedro Salgueiro and Salvador Abreu. Network Monitoring with Constraint Programming: Preliminary Specification and Analysis. In Salvador Abreu and Dietmar Seipel, editors, *Proceedings of the 18th International Conference on Applications of Declarative Programming and Knowledge Management*, pages 37–52, Universidade de Évora, Évora, Portugal, 2009.

[8] Pedro Salgueiro and Salvador Abreu. Constraint-Based DSL for Computer Network Monitoring. In *Doctoral Symposium on Artificial Intelligence, Fourteenth Portuguese Conference on Artificial Intelligence*, 2009.

[9] K.S.P. Arun. Flow-aware cross packet inspection using bloom filters for high speed data-path content matching. In *Advance Computing Conference, 2009. IACC 2009. IEEE International*, pages 1230 –1234, 6-7 2009. doi: 10.1109/IADCC.2009.4809191.

[10] S. Axelsson. Intrusion detection systems: A survey and taxonomy. 2000.

[11] Y. Zhang and W. Lee. Intrusion detection in wireless ad-hoc networks. In *Proceedings of the 6th annual international conference on Mobile computing and networking*, page 283. ACM, 2000.

[12] Elizabeth D. Zwicky, Simon Cooper, and D. Brent Chapman. *Building internet firewalls - internet and web security (2. ed.)*. O'Reilly, 2000. ISBN 978-1-56592-871-8.

[13] B. Mukherjee, L.T. Heberlein, and K.N. Levitt. Network intrusion detection. *Network, IEEE*, 8(3):26–41, 1994.

[14] D.E. Denning. An intrusion-detection model. *Software Engineering, IEEE Transactions on*, (2):222–232, 1987.

[15] C. Dowell and P. Ramstedt. The computerwatch data reduction tool. In *Proceedings of the 13th National Computer Security Conference*, pages 99–108, 1990.

[16] S.E. Smaha. Haystack: An intrusion detection system. In *Aerospace Computer Security Applications Conference, 1988., Fourth*, pages 37–44. IEEE, 1988.

[17] T.F. Lunt. Real-time intrusion detection. In *COMPCON Spring'89. Thirty-Fourth IEEE Computer Society International Conference: Intellectual Leverage, Digest of Papers.*, pages 348–353. IEEE, 1989.

[18] JR Winkler and WJ Page. Intrusion and anomaly detection in trusted systems. In *Computer Security Applications Conference, 1989., Fifth Annual*, pages 39–45. IEEE, 1989.

[19] JR Winkler. A unix prototype for intrusion and anomaly detection in secure networks. In *Proceedings of the 13th National Computer Security Conference*, pages 115–124, 1990.

[20] M. Sebring, E. Shellhouse, M. Hanna, and R. Whitehurst. Expert systems in intrusion detection: A case study. In *Proceedings of the 11th National Computer Security Conference*, pages 74–81, 1988.

[21] K. Ilgun. Ustat: A real-time intrusion detection system for unix. In *Research in Security and Privacy, 1993. Proceedings., 1993 IEEE Computer Society Symposium on*, pages 16–28. IEEE, 1993.

[22] P. Porras. Stat–a state transition analysis tool for intrusion detection. 1993.

[23] J. Hochberg, K. Jackson, C. Stallings, JF McClary, D. DuBois, and J. Ford. Nadir: An automated system for detecting network intrusion and misuse* 1. *Computers & Security*, 12(3):235–248, 1993.

[24] L.T. Heberlein, G.V. Dias, K.N. Levitt, B. Mukherjee, J. Wood, and D. Wolber. A network security monitor. 1990.

[25] S.R. Snapp, J. Brentano, G.V. Dias, T.L. Goan, T. Grance, L.T. Heberlein, C.L. Ho, K.N. Levitt, B. Mukherjee, D.L. Mansur, et al. A system for distributed intrusion detection. In *Compcon Spring'91. Digest of Papers*, pages 170–176. IEEE, 1991.

[26] G. Vigna and R.A. Kemmerer. Netstat: A network-based intrusion detection approach. In *Computer Security Applications Conference, 1998, Proceedings., 14th Annual*, pages 25–34. IEEE, 1998.

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

[28] Jay Beale. *Snort 2.1 Intrusion Detection, Second Edition.* Syngress Publishing, 2004. ISBN 1931836043.

[29] V. Paxson. Bro: a system for detecting network intruders in real-time* 1. *Computer networks*, 31(23-24):2435–2463, 1999.

[30] V. Jacobson, C. Leres, and S. McCanne. libpcap, Lawrence Berkeley Laboratory, Berkeley, CA. *Initial public release June*, 1994.

[31] H. Song and J.W. Lockwood. Efficient packet classification for network intrusion detection using FPGA. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 238–245. ACM New York, NY, USA, 2005.

[32] R. Sommer. Bro: An open source network intrusion detection system. *Proceedings of the 17. DFN-Arbeitstagung
"uber Kommunikationsnetze*, pages 273–288, 2003.

[33] V. Paxson. Bro: a system for detecting network intruders in real-time* 1. *Computer networks*, 31(23-24):2435–2463, 1999.

[34] A. Viswanathan, A. Hussain, J. Mirkovic, S. Schwab, and J. Wroclawski. A semantic framework for data analysis in networked systems. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, pages 10–10. USENIX Association, 2011.

[35] M. Attig and J. Lockwood. A framework for rule processing in reconfigurable network systems. In *Field-Programmable Custom Computing Machines, 2005. FCCM 2005. 13th Annual IEEE Symposium on*, pages 225–234. IEEE.

[36] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. Markatos, and S. Ioannidis. Gnort: High performance network intrusion detection using graphics processors. In *Recent Advances in Intrusion Detection*, pages 116–134. Springer, 2008.

[37] C.J. Coit, S. Staniford, and J. McAlerney. Towards faster string matching for intrusion detection or exceeding the speed of snort. In *DARPA Information Survivability Conference & Exposition II, 2001. DISCEX'01. Proceedings*, volume 1, pages 367–373. IEEE, 2001.

[38] M. Couture, B. Ktari, M. Mejri, and F. Massicotte. A declarative approach to stateful intrusion detection and network monitoring. In *Proceeding of the Second Annual Conference on Privacy, Security and Trust, Fredericton, New Brunswick, Canada*, 2004.

[39] J.P. Pouzol and M. Ducasé. From declarative signatures to misuse ids. In *Recent Advances in Intrusion Detection*, pages 1–21. Springer, 2001.

[40] S. Kumar and E.H. Spafford. A software architecture to support misuse intrusion detection. In *Proceedings of the 18th national information security conference*, pages 194–204, 1995.

[41] S. Kumar and E.H. Spafford. A pattern matching model for misuse intrusion detection. In *Proceedings of the 17th national computer security conference*, volume 10, page 11. Baltimore, MD, USA: NIST National Institute of Standards and Technology/National Computer Security Center, 1994.

[42] M. Crosbie, B. Dole, T. Ellis, I. Krsul, and E. Spafford. Idiot-users guide. *COAST Laboratory, Purdue University*, 1398, 1996.

[43] F. Cuppens and R. Ortalo. Lambda: A language to model a database for detection of attacks. In *Recent advances in intrusion detection*, pages 197–216. Springer, 2000.

[44] N. Habra, B. Charlier, A. Mounji, and I. Mathieu. Asax: Software architecture and rule-based language for universal audit trail analysis. *Computer Security-ESORICS 92*, pages 435–450, 1992.

[45] P.G. Neumann and P.A. Porras. Experience with emerald to date. In *Proceedings of the 1st conference on Workshop on Intrusion Detection and Network Monitoring-Volume 1*, pages 8–8. USENIX Association, 1999.

[46] F. Rossi, P. Van Beek, and T. Walsh. *Handbook of constraint programming.* Elsevier Science, 2006.

[47] R. Barták. Constraint programming: In pursuit of the holy grail. In *Proceedings of WDS99 (invited lecture), Prague, June*, pages 205–224, 1999.

[48] S.C. Brailsford, C.N. Potts, and B.M. Smith. Constraint satisfaction problems: Algorithms and applications. *European Journal of Operational Research*, 119(3): 557–581, 1999.

[49] D. Waltz. Understanding line drawings of scenes with shadows. In *The psychology of computer vision*, 1975.

[50] I.E. Sutherland. Sketch pad a man-machine graphical communication system. In *Proceedings of the SHARE design automation workshop*, pages 6–329. ACM, 1964.

[51] J.L. Lauriere. A language and a program for stating and solving combinatorial problems. *Artificial intelligence*, 10(1):29–127, 1978.

[52] G.J. Sussman, G.L. Steele, et al. Constraints–a language for expressing almost-hierarchical descriptions. *Artificial intelligence*, 14(1):1–39, 1980.

[53] A.K. Mackworth. Consistency in networks of relations. *Artificial intelligence*, 8 (1):99–118, 1977.

[54] M. Wallace. Practical applications of constraint programming. *Constraints*, 1 (1):139–168, 1996.

[55] H. Gallaire. Logic programming: Further developments. In *IEEE Symposium on Logic Programming*, pages 88–99, 1985.

[56] J. Jaffar and J.L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 111–119. ACM, 1987.

[57] A. Aggoun and N. Beldiceanu. Overview of the CHIP compiler system. *Mit Press Series In Logic Programming*, pages 421–435, 1993.

[58] D. Diaz and P. Codognet. Design and implementation of the gnu prolog system. *Journal of Functional and Logic Programming*, 6(2001):542, 2001.

[59] J.F. Puget. A c++ implementation of clp. 1994.

[60] D. Sam-Haroud and B. Faltings. Consistency techniques for continuous constraints. *Constraints*, 1(1):85–118, 1996.

[61] K.R. Apt. Local consistency notions. In *Principles of Constraint Programming*, pages 135–176. Cambridge Univ Pr, 2003.

[62] A.K. Mackworth. On reading sketch maps. In *Proceedings of the 5th international joint conference on Artificial intelligence-Volume 2*, pages 598–606. Morgan Kaufmann Publishers Inc., 1977.

[63] R. Mohr and T.C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28(2):225–233, 1986.

[64] Roger Mohr and G. Masini. Good old discrete relaxation. In *Proceedings of the 8th European Conference on Artificial Intelligence*, pages 651–656, 1988.

[65] C. Bessière and M.O. Cordier. Arc-consistency and arc-consistency again. In *Proceedings of the eleventh national conference on Artificial intelligence*, pages 108–113. AAAI Press, 1993.

[66] K.R. Apt. Search. In *Principles of Constraint Programming*, pages 299–349. Cambridge Univ Pr, 2003.

[67] R.M. Haralick and G.L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial intelligence*, 14(3):263–313, 1980.

[68] D. Sabin and E. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Principles and Practice of Constraint Programming*, pages 10–20. Springer, 1994.

[69] C. Schulte and P.J. Stuckey. Speeding up constraint propagation. *Lecture Notes in Computer Science*, 3258:619–633, 2004.

[70] Gecode interfaces web page at `http://www.gecode.org/interfaces.html`, October, 2011.

[71] Gecode/R Team. Gecode/R: Constraint Programming in Ruby. Available from http://gecoder.org/.

[72] Aliceml web page at `http://www.ps.uni-saarland.de/alice/`, August, 2011.

[73] Geoz web page at `http://cic.puj.edu.co/wiki/doku.php?id=grupos:avispa:geoz`, August, 2011.

[74] P. Wuille and T. Schrijvers. Monadic constraint programming with gecode. In *Proceedings of the 8th International Workshop on Constraint Modelling and Reformulation*, pages 171–185, 2009.

[75] Gelisp web page at `http://gelisp.sourceforge.net/`, October, 2011.

[76] P. Van Hentenryck and L. Michel. *Constraint-based local search.* MIT Press, 2005.

[77] L. Michel and P.V. Hentenryck. A constraint-based architecture for local search. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 83–100. ACM, 2002.

[78] P. Codognet and D. Diaz. Yet another local search method for constraint solving. *Lecture Notes in Computer Science*, 2264:73–90, 2001.

[79] A. Biere and IOS Press. *Handbook of Satisfiability.* IOS Press, 2009. ISBN 1441616780.

[80] M. Gavanelli. The log-support encoding of csp into sat. In *Proceedings of the 13th international conference on Principles and practice of constraint programming*, pages 815–822. Springer-Verlag, 2007.

[81] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960.

[82] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Design Automation Conference, 2001. Proceedings*, pages 530–535. IEEE, 2001.

[83] T. Walsh. Sat v csp. *Principles and Practice of Constraint Programming–CP 2000*, pages 441–456, 2000.

[84] I.P. Gent. Arc consistency in sat. In *Proceedings*, volume 77, page 121. IOS Pr., 2002.

[85] S. Kasif. On the parallel complexity of discrete relaxation in constraint satisfaction networks. *Artificial Intelligence*, 45(3):275–286, 1990.

[86] S. Prestwich. Local search on sat-encoded colouring problems. In *Theory and Applications of Satisfiability Testing*, pages 26–29. Springer, 2004.

[87] N. Sörensson and N. Een. Minisat v1. 13-a sat solver with conflict-clause minimization. *SAT*, 2005:53, 2005.

[88] N. Een and N. Sörensson. Minisat v2. 0 (beta). *Solver description, SAT Race*, 2006, 2006.

[89] Satzoo web page at `http://een.se/niklas/Satzoo/`, August, 2011.

[90] Satnik web page at `http://vlsicad.eecs.umich.edu/BK/Slots/cache/www.cs.chalmers.se/~nik/`, October, 2011.

[91] J.P. Marques Silva and K.A. Sakallah. Grasp-a new search algorithm for satisfiability. In *Computer-Aided Design, 1996. ICCAD-96. Digest of Technical Papers., 1996 IEEE/ACM International Conference on*, pages 220–227. IEEE, 1996.

[92] Daniel Diaz and Philippe Codognet. *The Adaptive Search Manual*, 1.0, for Adaptive Search version 0.9.0 edition, 02 2003.

[93] M. Mernik, J. Heering, and A.M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys (CSUR)*, 37(4):316–344, 2005.

[94] A. Raja and D. Lakshmanan. Domain specific languages. *International Journal of Computer Applications IJCA*, 1(21):105–111, 2010.

[95] Jon Bentley. Programming pearls: little languages. *Commun. ACM*, 29:711–721, August 1986. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/6424.315691. URL `http://doi.acm.org/10.1145/6424.315691`.

[96] D.E. Knuth. *TEX: the Program.* Addison-Wesley Pub. Co., 1993.

[97] Douglas T. Ross. History of programming languages i. chapter Origins of the APT language for automatically programmed tools, pages 279–338. ACM, New York, NY, USA, 1981. ISBN 0-12-745040-8. doi: http://doi.acm.org/10.1145/800025.1198374. URL `http://doi.acm.org/10.1145/800025.1198374`.

[98] John W. Backus. The syntax and semantics of the proposed international algebraic language of the zurich acm-gamm conference. In *IFIP Congress*, pages 125–131, 1959.

[99] A. Van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.

[100] A.V. Aho, B.W. Kernighan, and P.J. Weinberger. Awk-a pattern scanning and processing language. *Software: Practice and Experience*, 9(4):267–279, 1979.

[101] L.E. McMahon. Sed-a non-interactive text editor. *UNIX Programmer's Manual*, 2.

[102] J. Martin. Fourth-generation languages. volume 2. representative 4gls. 1985.

[103] T.J. Biggerstaff and A.J. Perlis. Software reusability: vol. 1, concepts and models. 1989.

[104] A. Van Deursen and P. Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–17, 2004.

[105] J. Neighbors. *The Draco approach to constructing software from reusable components*, pages 525–535. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1986. ISBN 0-934613-12-5. URL `http://portal.acm.org/citation.cfm?id=31870.31900`.

[106] Mark G. J. van den Brand, Arie van Deursen, Jan Heering, H. A. de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A. Olivier, Jeroen Scheerder, Jurgen J. Vinju, Eelco Visser, and Joost Visser. The asf+sdf meta-environment: A component-based language development environment. In *Proceedings of the 10th International Conference on Compiler Construction*, CC '01, pages 365–370, London, UK, 2001. Springer-Verlag. ISBN 3-540-41861-X. URL `http://portal.acm.org/citation.cfm?id=647477.727788`.

[107] Rickard E. Faith, Lars S. Nyland, and Jan F. Prins. Khepera: a system for rapid implementation of domain specific languages. In *Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL), 1997*, pages 19–19, Berkeley, CA, USA, 1997. USENIX Association. URL `http://portal.acm.org/citation.cfm?id=1267950.1267969`.

[108] R. M. Herndon, Jr. and V. A. Berzins. The realizable benefits of a language prototyping language. *IEEE Trans. Softw. Eng.*, 14:803–809, June 1988. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/32.6159. URL `http://dx.doi.org/10.1109/32.6159`.

[109] Lloyd H. Nakatani and Mark A. Jones. Jargons and infocentrism. In *First ACM SIGPLAN Workshop on Domain-Specific Languages*, pages 59–74, 1997. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.51.858`.

[110] tcpdump web page at `http://www.tcpdump.org`, April, 2009.

[111] Salvador Abreu, Daniel Diaz, and Philippe Codognet. Parallel local search for solving constraint problems on the cell broadband engine (preliminary results). *CoRR*, abs/0910.1264, 2009.

[112] Nmap web page at `http://nmap.org`, June, 2011.

[113] Medusa web page at `http://www.foofus.net/~jmk/medusa`, May, 2011.

[114] hping web page at `http://www.hping.org`, May, 2011.

[115] D. Norton and P.A. Version. An Ettercap Primer. *GIAC Security Essentials Certification Practical*, 2004.

[116] S. Mathew, D. Britt, R. Giomundo, S. Upadhyaya, M. Sudit, and A. Stotz. Real-time multistage attack awareness through enhanced intrusion alert clustering. In *Military Communications Conference, 2005. MILCOM 2005. IEEE*, pages 1801–1806. IEEE, 2005. ISBN 0780393937.

[117] W.J. Noonan. *Hardening network infrastructure*. McGraw-Hill Osborne Media, 2004. ISBN 0072255021.