Using Constraints for Intrusion Detection : the NeMODe System

Pedro Salgueiro[1], Daniel Diaz[2] and Isabel Brito[3], and Salvador Abreu[1]

[1] Departamento de Informática, Universidade de Évora
and CENTRIA FCT/UNL, Portugal
{pds,spa}@di.uevora.pt

[2] University of Paris 1-Sorbonne, Paris, France
Daniel.Diaz@univ-paris1.fr

[3] Departamento de Engenharia, Escola Superior de Tecnologia e Gestão,
Instituto Politécnico de Beja, Portugal
isabel.sofia@estig.ipbeja.pt

Abstract. In this work we present NeMODe a declarative system for Computer Network Intrusion detection which provides a declarative Domain Specific Language for describing computer network intrusion signatures that could spread across several network packets, which allows to state constraints over network packets, describing relations between several packets, and providing several back-end detection mechanisms which relies on Constraint Programming (CP) methodologies to find those intrusions.

Keywords: Constraint Programming, Intrusion Detection Systems, Domain Specific Languages

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 the users 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 the users 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 [1], enabling the description of these situations in a natural, declarative and expressive way. To help the description of those network situations, we created a declarative, very expressive, Domain Specific Language (DSL) [2], enabling an easy description of intrusion signatures that spread across several network packets, allowing to state constraints over network entities and express relations across several network packets. This DSL will then translate the *program* into constraints that will be solved by more than one constraint solving techniques, including Constraint Based Local Search and Propagation-based systems such as Gecode [3]. It also have the capabilities of running several solvers in parallel, in order to benefit from the earliest possible solution. We have already made some preliminary work on using network constraints to perform intrusion detection [4], and have also developed a preliminary implementation of such a DSL[5,6]. In this work we present a new and more complete version version of such a DSL (NeMODe) as well as a its complete specification to a better comprehension.

This paper is organized as follows. Section 2 presents the state of the art and a brief description of Intrusion Detection Systems, Constraint Based Local Search, Adaptive Search and Domain Specific Languages. Section 3 demonstrates how to model and perform Intrusion Detection using Constraint Programming. Section 4 details the DSL provided by NeMODe and provides some examples. Section 5 shows the experimental results obtained by NeMODe. Section 6 evaluates NeMODe and Section 7 presents the conclusions and future work. Throughout this paper, we mention technical terms pertaining to TCP/IP and UDP/IP network packets, such as *packet flags ,URG, ACK, PSH, RST, SYN, FIN, acknowledgment, source port, destination port, source address, destination address, payload*, which are described in [7].

2 State of the art

2.1 Intrusion Detection Systems

Intrusion Detection Systems(IDS) play a very important role in computer network security, which focus on traffic monitoring trying to inspect traffic to look for anomalies or undesirable communications in order to keep the network a safe place. There are two major methods to detect intrusions in computer networks; (1) based on the network intrusion signatures, and (2) based on the detection of anomalies on the network [8]. With Signature Based Intrusion Detection, intrusions are described using their signatures, particular properties of network packets used by the intrusion. These properties are then looked in the network traffic to find the desired intrusion. In Anomaly-Detection Based, the systems models the *normal* behavior of the network using statistical methods and/or data mining approaches. The network behavior is then monitored, and if it is considered anomalous according the network model, there is a great probability of and attack. In this work, we adopted an approach based on signatures.

Snort [9] is a widely used Intrusion Detection System that relies on efficient pattern-matching techniques to detect the desired intrusion signature. Snort is primarily designed to detect signatures that can be identified in a single network packet. Although it provides some basic mechanisms to write rules that spread across several network packets, the relations between those network packets are very simple and limited.

Snort presents some pre-processors that help to 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 later 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 and often counter-intuitive.

Most of the work in the area of Intrusion Detection Systems consists in the development of faster detection methods [10]. The work described in [11] is such an example, which implements a regular expression matching algorithm using graphics hardware (GPUs) to perform intrusion detection. There is also some work focused on how the network signatures are described detected. [10] presents an algorithm and an implementation method for performing flow aware content search based on Bloom Filters which allows to search signatures that spread across several packets. In [12], the authors present a declarative approach to specify intrusion signatures which are represented as a specialized graph, allowing the description of signatures that spread across several network packets.

2.2 Constraint Programming

Constraint Programming (CP) is a declarative programming paradigm which consists in the formulation of a solution to a problem as a *Constraint Satisfaction Problem* (CSP) [1], in which a number of variables are introduced, with well-specified domains and which describe the 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 for a particular set of bindings for the variables, resulting in a *solution* to the CSP.

There are several types of constraint solvers, in this work we use: (1) Propagation Based solvers; and (2) Constraint Based Local Search(CBLS).

Propagation-Based solvers Using Propagation-Based [1] solvers, the problem is described by stating constraints over each variable that composes the problem, which states what values are allowed to be assigned to each variable, then, the constraint solver will propagate all the constraints and reduce the domain of each network variables in order to satisfy all the constraints and instantiate the variables that compose the problem with valid results, thus reaching a solution to the initial problem. Gecode [13] is a constraint solver library based on propagation, implemented in C++ and designed to be interfaced with other systems or programming languages.

Constraint Based Local Search CBLS [14] is a fundamental approach to solve combinatorial problems such as Constraint Satisfaction Problems. CBLS is a method that can solve very large problems, although not a complete algorithm and unable to provide a complete or optimal solution. Usually, this approach initiates with an initial, candidate solution to the problem which is then iteratively improved though small modifications until some criteria is satisfied. The modifications to the candidate solution is usually driven by heuristics that guide the solver to a solution.

Adaptive Search (AS) [15] is a Constraint Based Local Search [14] 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 in Adaptive Search are based on variable and constraint error information which seeks to reduce errors on the variables used to model the problem. AS computes the error of all constraints in which it appears, projecting the errors on each individual variables. Based on this information, the variable with the highest cost is the one that will be chosen to change its value. After the variable with the highest cost have been calculated, the *min_conflict* [1] heuristic is used to select the new value to that variable, which is the value that provides the minimum total error to the next solution. Adaptive Search has recently been ported to Cell/BE, presented in [16].

2.3 Domain Specific Languages

Domain Specific Languages(DSLs) [2] allows to easily create programs to a specific and well defined domain with efficiency, generating easy to understand and maintain programs, by using a specific *jargon*. Most IDSs, like Snort and Bro [17], also a widely used IDS, provide custom languages to describe the signatures, but they are usually scripting languages, based mostly on pattern matching and regular expressions, *counter-intuitive*, and don't use a declarative approach, making them less expressive.

3 Intrusion Detection with Constraints

Our approach to intrusion detection relies on describing the desired signatures through the use of constraints and then identify a set of packets that match the target network situation in the network traffic window, which is a log of the network traffic in a given time interval.

The network intrusion needs to be modeled as a Constraint Satisfaction Problem (CSP) in order to use the constraint programming mechanisms. A CSP which models a network situation is composed by a set of variables, V, which represents the network packets involved necessary to describe the network situation; the domain of the network packet variables, D; and a set of constraints, C, which relates the variables in order to describe the network situation. We call such a CSP a network CSP. On a network CSP, each network packet variable is a tuple of integer variables, 19 variables for TCP/IP [4] packets and 12 variables for UDP packets [5], which represent the significant fields of a network packet necessary to

[4] Here, we are only considering the "interesting" fields in TCP/IP packets, from an IDS point-of-view.

[5] Here, we are only considering the "interesting" fields in UDP packets, from an IDS point-of-view.

model the intrusion signatures used in our experiments.

The domain of the network packet variables, D, are the values actually seen on the network traffic window, which is a set of tuples of 19 integer values (for the TCP variables) and 12 integer values (for the UDP variables), each tuple representing a network packet actually observed on the traffic window and each integer value represents each field relevant to intrusion detection. The packets payload is stored separately in an array containing the payload of all packets seen on the traffic window. The correspondence between the packet and its payload is achieved by matching the packet number, i, which is the first variable in the tuple representing the packets and the i^{th} position of the array containing the payloads.

Listing 1 shows a representation of such CSP, where P represents the set of network packet variables, where $P_{n,z}$, is each of the individual integer variables of the network packet variable, in a total of z fields for each network of the n variables, with $z = 19$ for TCP packets and $z = 12$ for UDP packets.

D is the network traffic window, where $D_i = (V_{i,1}, \ldots, V_{i,z}) \in D$ is one of the real network packets on the network traffic window, which is part of the domain of the packet variables P.

$Data$ is the payloads of the network packets present in the network window, where $Data_i$ is the payload of the packet $P_i = (V_{i,1}, \ldots, V_{i,z}) \in D$.

The associated domains of the network packet variables is represented by $\forall P_i \in P \Rightarrow P_i \in D$, forcing all variables belonging to P to obtain values from the set of packets in the network window D.

A solution to a network CSP, if it exists, is an assignment of network packet values, $D_i = (V_{i,1}, \ldots, V_{i,z}) \in D$, to each packet variable, $P_i = (P_{j,1}, \ldots, P_{j,z}) \in P$, that models the desired situation, thus identifying the network packets that identify the intrusion being detected.

Listing 1 Representation of a network CSP

$$
\begin{aligned}
&P = \{(P_{1,1}, \ldots, P_{1,z}), \ldots, (P_{n,1}, \ldots, P_{n,z})\} \\
&D = \{(V_{1,1}, \ldots, V_{1,z}), \ldots, (V_{x,1}, \ldots, V_{x,z})\} \\
&Data = \{Data_1, \ldots, Data_x\} \\
&\forall P_i \in P \Rightarrow P_i \in D
\end{aligned}
$$

4 NeMODe - A DSL to describe network signatures

In this work we present a declarative, intuitive domain-specific programming language for the Network Intrusion Detection [2] of NeMODe, which talks about network entities, their properties and relations between them, allowing to describe network intrusion signatures, and, with base on those descriptions, generate Intrusion Detection mechanisms.

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

Maintaining the declaritivity and expressiveness of the 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 DSL is a front-end to several back-ends, one to each intrusion detection mechanism. This allows to generate several recognizers based on different constraint solver methods, from a single description. With several recognizers, it is possible run each of them in parallel, allowing to select the first produced solution, as the behavior of each solver depends on the problem being solved.

NeMODe provides two back-end detection mechanisms; (1) based on the Gecode constraint solver; and (2) based on the Adaptive Search algorithm. Each of these detection mechanisms are based on Constraint Programming techniques, but they are completely different in the way they perform the detection, and also the way the signatures are described. In Sec. 2.2 each of these approaches are explained.

4.1 NeMODe specification

A NeMODe program is composed by an optional set of initial declarations, followed by a network ***case***(line 1 of Listing 2), which describes the network situation to be modeled.

Those initial declarations is a comma separated list of declarations port numbers and/or hostnames, which can later be used later on the description of the problem, making the program more *readable*, by referring to hostnames instead of ip addresses, and port or service names instead of port numbers.

A network ***case*** is the main part of a NeMODe program and is composed by two parts; (1) the ***solver_list***, (line 3 of Listing 2) containing the description of the intrusion signature to be found and the identification of the tool which will be used to solve the problem; and (2) the actions to take when the desired network situation is detected, the ***stmt_action_list***(line 1 of Listing 6). There are two types of solvers,(line 5 of Listing 2), the ***filter*** and the ***solver***. The ***solver*** is used to describe and solve complex network intrusion signatures, while the ***filter*** is only used to perform simple filtering tasks, accomplished by using a packet analyzer tool, such as *tcpdump* [18].

A ***solver***(line 5 of Listing 2) is composed by 3 parts; (1) the network traffic source; (2) the identification of the tools that will be used to perform the filtering; and (3) the description of the filtering/solving process. The result of this filtering process is then stored in a variable, which could later be used as an input to other filtering stage. The most important part of a NeMODe program is the list of statements, ***stmt_list*** (line 8 of Listing 2), where the signatures are described.

Listing 2 NeMODe simplified grammar - The beginning of a program

```
1  case → ID { solver_list } => { stmt_action_list };
2
3  solver_list → solver | solver_list , solver
4  solver → ID = filter ( ID , ID ) { primitive_list }
5          | ID = solve ( ID , ID ) { stmt_list }
6
7  stmt_list → stmt | stmt_list , stmt
8  stmt → primitive | connective | ID  = { stmt_list } | ID  | macro_stmt | logic_stmt
```

There are 6 types of statements(line 8 of Listing 2); (1) the ***primitive*** statements; (2) the ***connective*** statements; (3) the ***definition*** statements; (4) the ***use*** statements; (5) the ***macro*** statements; and (6) the ***logical*** statements.

Listing 3 NeMODe simplified grammar - The most important statements

```
1  primitive → primitive_type ( var  )
2             | data ( var ) ~= STRING  |  data ( var , NUMBER ) == STRING
3             | address eq_op ID | address eq_op ip_address
4             | port eq_op NUMBER | port eq_op ID
5
6  primitive_type → tcp_packet | udp_packet | urg | ack | psh | rst | syn | fin | nak
7
8  connective → ack ( var ) eq_op var
9               | port eq_op port | address eq_op address
10              | time rel_op time
11              | data ( var , NUMBER , NUMBER ) ==  data ( var , NUMBER , NUMBER )
```

The ***primitive*** statements (line 1 of Listing 3) allows to force some specific properties of a network packets to hold true. This statements allows to force a network packet to be tcp/udp packet; to have any of its tcp flags set; not to acknowledge another tcp packet; force a packet to have a specific data on its payload; and assure that a network packet have a specific source/destination address or a specific source/destination port.

The ***connective*** statements (line 8 of Listing 3) allows to relate two network packets by forcing the existence of some relations between the two of them. They allow to force: (1) a tcp packet to acknowledge other tcp packet; (2) a destination/source port of a packet to be equal/different to other 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; and (5) two network packets to have a temporal relation, such as their temporal distance to be inferior to a given amount of time.

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

The ***definition*** statements (line 8 of Listing 2) 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 on the program, being only the definition of a variable.

The ***use*** statement (line 8 of Listing 2) is just the simple use of a definition previously defined. As for the ***macro*** statements (line 1 of Listing 4), these are built with the purpose of avoiding the repetition of unnecessary code.

Listing 4 NeMODe simplified grammar - The *macro* statements

```
1  macro_stmt → ID := repeat
2             | interval  ( var ) eq_op time | duration ( var ) eq_op time
3             | connection ( var , var )
4
5  repeat → repeat ( NUMBER , var )
```

The ***repeat***(line 5 of Listing 4) statement is one of the available ***macro*** statements, which allows to repeat a previously defined variable a given number of times. That *repetitions* are then stored under a variable, i.e. *R := repeat(3,C)*, so that later be possible to state constraints over a specific variable of an *iteration* of the *repetition*.

The *macro* statement ***duration*** (line 2 of Listing 4) forces the overall duration of a *repetition* to a be higher or lower than a certain amount o time, i.e. `duration(R) < secs(60)`. As for the *macro* statement ***interval***(line 2 of Listing 4), it forces the time between two *iterations* of a *repetition* to be higher/lower than a given amount of time, i.e. `interval(R) < secs(60)`. Finally, the last *macro* statement, ***connection***(line 3 of Listing 4), forces two network packets to be related, so that the source/destination of one packet be the destination/source of other packet.

The last type of statements is the ***logical*** statements(line 1 of Listing 5), which allows to specify logic operations(`and`, `or`) over ***primitives*** and ***connective*** statements.

Listing 5 NeMODe simplified grammar - Logic statements

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

The ***stmt_action_list***(line 1 of Listing 6) part of a ***case***, allows to describe the actions to take when an intrusion is detected, which is a coma separated list of statements, being allowed to use a previously described ***primitive*** and/or ***connective*** statements, as well as the ***alert*** statement. This list of statements allows to specify a set of properties over a set of network packets, being possible to relate them with variables used in the description of the network intrusion signature. Those new variables can later be used in the ***alert*** statement, together with some *strings* to alert the network administrator for an intrusion.

Listing 7 describes some basic entities, such as port addresses, ip address and time, used in several types of statements.

Listing 6 NeMODe simplified grammar - Action statements

```
stmt_action_list → stmt_action | stmt_action_list , stmt_action
stmt_action → primitive | connective | actions

actions → alert ( alert_arg_list )
alert_arg_list → alert_arg | alert_arg_list , alert_arg
alert_arg → var | STRING
```

Listing 7 NeMODe simplified grammar - Basic entities

```
port → dst_port ( var )  | src_port ( var )
address → src ( var ) | dst ( var )
time → usecs ( NUMBER ) | secs ( NUMBER ) | time_arith
time_arith → time ( var ) | NUMBER
            | time ( var ) arith_op time_arith
```

Variables

NeMODe variables, (line 1 of Listing 8), are always upper case, and can be categorized in several types: (1) the initial ***declarations*** variables; (2) the ***solver/-filters*** variables; (3) the ***definitions*** variables; (4) the ***repetitions*** variables and the network packet variables. The declaration of the variables is implicit, being defined the first time they are referenced or used.

Listing 8 NeMODe simplified grammar - Variables

```
1  var → ID  | repeat_var | filter_var
2  repeat_var → ID [ NUMBER ] : ID
3  filter_var → ID . ID | ID . repeat_var
```

Variable scope

A NeMODe program is composed by several scopes, the first one the program itself, then, a second scope for the ***solvers/filters***, and inside each solver there might exist a third scope, the repetition of a definition. At each scope level, it might be necessary to access a variable of a higher scope level. Accessing a higher scope level variable is transparent if there is no other variable with the same name on the current scope level, otherwise there is the need to access that variable using a special syntax.

Accessing a variable inside a *repetition* To access a variable defined in a ***definition***, assigned to a variable, one starts to refer the ***repetition*** variable, then the number of the ***iteration*** and finally the variable name, e.g. `r[2].A`.

Accessing a variable inside a *solver* Sometimes it is necessary to access a variable defined inside a ***solver*** or ***filter***, to do this, one starts to refer the filter and then the desired variable, which can be either a simple variable, e.g. `gecode.A` or a variable inside a ***repetition***, e.g. `gecode.R[2].A`.

4.2 Examples

So far, we have worked with some simple network intrusion signatures: (1) a DHCP spoofing, (2) a DNS spoofing and (3) a SYN flood attack. All of these intrusion patterns can be described using NeMODe and the generated code was successful in finding the desired situations in the network traffic logs. A Portscan attack and an SSH Password brute-force attack are further explained in [5].

DHCP spoofing DHCP Spoofing is a Man in The Middle(MITM) attack, where the attacker tries to reply to a DHCP request faster than the legit DHCP server of the local network, allowing the attacker to provide false network configurations to the target host, such as the default gateway, forcing all traffic from/to the target to pass though an attacker controlled machine, allowing it to capture or modify the important data. This kind of intrusion can be detected by looking for several answers to a single DHCP request, originated in different machines, although, if the attacker spoofs its addresses, invalidates this detection method. A NeMODe program to model a DHCP spoofing is shown in Listing 9. Line 2 describes the packet that initiates a requests a DHCP, line 3 the first reply to the request and line 4 the second reply the DHCP request. Finally, on line 6 is stated that packets `B` and `C`(the first and second reply) should have different source addresses.

Listing 9 A DHCP Spoofing attack programmed in NeMODe

```
1  dhcp_spoofing {
2      udp_packet(A), dst_port(A)==67,
3      udp_packet(B), dst_port(B)==68,
4      udp_packet(C), dst_port(C)==68,
5
6      src(B) != src(C)
7  } => {
8      alert('DHCP Spoofing attempt')
9  };
```

DNS spoofing DNS Spoofing is also a Man in The Middle (MITM) attack. In this attack, the attacker tries to provide a false DNS query posted by the victim, if succeeded the victim could access a machine under the control of the attacker, thinking that it is accessing the legit machine, allowing the attacker to obtain crucial data from the victim. In order to arrange this attack, the attacker tries to respond with a false DNS query faster than the legit DNS server, providing a false IP address to the name that the victim was looking for. This kind of attacks is possible to detect by looking for several replies to the same DNS query. Listing 10 shows how this attack can be programmed using NeMODe. Line 2 describes the packet that makes the DNS request. Lines 4-5, describes a first reply to the DNS request and lines 7-8 describes the second reply. Lines 10-12 states that packets `B` and `C` should be different and that the *DNS id* of the replies should be the equal to the DNS request, which is the first two bytes of the packets data.

SYN flood attack A SYN flood attack happens when the attacker initiates more TCP/IP connections than the server can handle and then ignoring the replies from the server, forcing the server to have a large number of half open

Listing 10 A DNS Spoofing attack programmed in NeMODe

```
dns_spoofing {
    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')
};
```

connections in standby, which leads the service to stop when this number reach the limit of number of connections. This attack can be detected if a large number of connections is made from a single machine to other in a very short time interval. Listing 11 shows how a SYN flood attack can be described using NeMODe. Lines 2-3 describes a TCP/IP packet with the SYN flag and assigns those properties to variable `C`. In line 4, the *macro* statement ***repeat*** is used to repeat the properties of definition `C` 30 times, and assign it to variable `R`. Line 5 states that the time interval between each repetition of C should be less than to 500 micro-seconds.

Listing 11 A SYN flood attack programmed with NeMODe

```
1 syn_flood {
2   C = { tcp_packet(A),
3         syn(A), nak(A) },
4   R := repeat(30,C),
5   max_interval(R) < usecs(500)
6 } => {
7   alert('SYN flood attack attempt')
8 };
```

4.3 Code Generation

The current implementation of NeMODe is able to generate code for the Gecode solver and for the Adaptive Search algorithm. These two approaches to constraint solving are completely different as well as the description of the problems, forcing us to have several code generators for each of back-end available. We were able to minimize this difference by creating custom libraries for each constraint solver so that the code generation process is not completely different for each back-end. Fig. 1 represents the architecture of the system; starting with a NeMODe program, which is parsed into a semantic model, then it is generated code to the appropriate back-ends used. Them, the generated code receives as input the network traffic and produces a valid solution, if the described intrusion exists on the current network traffic.

Generating a Gecode program: This goal is achieved by generating code based on Gecode constraint propagators that describe the desired network

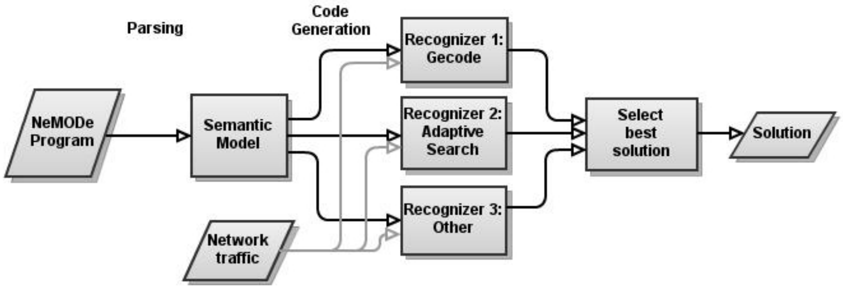

Fig. 1. NeMODe system architecture

signatures. We created a custom library that defines functions that combine several stock arithmetic Gecode constraints with element and extensional constraints to define custom, network related *macro* constraints. The same library includes definitions for a few network-related constraint propagators useful to implement some of the constraints needed to describe and solve IDS problems.

Generating an A.S. program: The task of generating Adaptive Search resumes to create the proper error functions so that Adaptive Search be able to solve the problem; the `cost_of_solution` and `cost_on_variable`. To ease the generation of this functions, a small library was created which implements small error functions, specific to the network intrusion detection domain, which are then used to generate the code for the error functions.

5 Experimental Results

While developing this work, several experiments were done. We have tested the examples of Sect. 4.2, a DHCP Spoofing attack, a DNS Spoofing attack and a SYN flood attack. All these network intrusions were successfully described using NeMODe and valid Gecode and Adaptive Search code was produced for all network signatures and then executed in order to validate the code and ensure that it could indeed find the desired network intrusions.

The code generated for Gecode was run on a dedicated computer, 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 Adaptive Search code, it run on an 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. The reason to run both detection mechanisms in different machines with a completely different architecture is because Adaptive Search has recently been ported to Cell/BE, and we choose this version of Adaptive Search to run our experiments, forcing us to use the QS21 dual-Cell/BE blades, which is incompatible with the implementation of Gecode, forcing us to use a machine with x86 architecture to run Gecode.

In all the experiments we used log files representing network traffic which contains the desired signatures to be detected. These log files were created with the help of *tcpdump*, which is a packet sniffer, during actual attacks to a computer to simulate the real attacks described in this work.

DHCP and DNS spoofing attacks: We programmed these two attacks using the DSL of The attack was provided by NeMODe, which successfully generated code for Adaptive Search as well as for Gecode and successfully detected the intrusions. Both problems were modeled using 3 udp network packets, each one composed of 12 integer variables, in a total of 36 integer variables. The search space for both this problems was a set of 400 udp network packets, each composed of 12 integer values, in a total of 4800 values.

SYN flood attack: In the SYN flood attack, we programmed with the DSL of NeMODe which in turn generated code for Adaptive Search and Gecode. This code was then used to successfully detect the intrusion. The problem was modeled by 30 tcp network packet variables, each comprised of 19 integer variables, in a total of 570 integer values. The the search space of the problem was composed by 100 tcp network packets, each composed of 19 integer values, in a total of 1900 values.

Table 1 presents the time(user time, in seconds) required to find the desired network situation for the attacks presented in the present work, using both detection mechanisms, Gecode and Adaptive Search. The times presented are the average of 128 runs.

Table 1. Average time(in seconds) necessary to detect the intrusions using Gecode and Adaptive Search

Intrusion to detect	**Gecode (seconds)**	**A.S (seconds)**
DHCP Spoofing	0.0082	0.3924
DNS Spoofing	0.0069	0.3512
SYN flood	0.0566	0.0466

6 Evaluation

The experimental results described in Sec. 5 shows that the performance varies in a great scale depending on the problem and the recognizer. Table 1 shows that that Gecode usually performs better than Adaptive Search, except in the SYN flood attack. The SYN flood attack performed better in Adaptive Search due to the fact that the network packets of the attack are close together and there aren't almost any other packets between the packets of the attack. The results obtained with Gecode, are quite good, allowing us to start the detection of intrusions in real network traffic instead of log files. Adaptive Search inferior performance figures are explained by the lack of good heuristic, as precise tuning of the sensitive algorithm of AS has yet to be done.

As for the DSL provided by NeMODe, it revealed to be very expressive and powerful, allowing an easy description of all the three network intrusions and generate valid code that could detect the desired network situation. Although other intrusion detection systems like Snort could detect 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.

7 Conclusions and Future Work

The work presented in this paper presents NeMODe, a system for Network Intrusion detection, which provide a declarative Domain Specific Language that generates intrusion detection recognizers based on Constraint Programming, more specifically, using Gecode and Adaptive Search. NeMODe presents a very expressive DSL that allows to describe network intrusion signatures by expressing relations between network packets simply by stating constraints over network packets.

This work shows that it is possible to use a single signature description based on CP to generate several recognizers, each one based on a different CP paradigms, and with that recognizers detect the desired intrusions.

We proved that we can easily describe network signature attacks that spread across several network packets, which can not me done in friendly and declarative way in systems like Snort. Although the intrusions mentioned in this work can be detected with other intrusion detection systems, they are modeled/described with out relating the several network packets of the intrusion, much of the times using a single network packet to describe the intrusion, which could in some situations produce a large number of false positives.

A very important future work is to model more network situations as a CSP in order to evaluate the performance of the system while working with a larger diversity of problems. Although the DSL allows to describe a broad range of attacks, it still needs more flexibility to cope with more types of signatures and include more back-ends. We also need to better evaluate the the work presented in this paper by comparing the obtained results with systems like Snort.

Also a very important future step is to start performing network intrusion tasks on live network traffic link, allowing to apply this method in a real network to assess its performance.

Acknowledgments

Pedro Salgueiro acknowledges FCT –Fundação para a Ciência e a Tecnologia– for supporting him with scholarship SFRH/BD/35581/2007. The IBM QS21 dual-Cell/BE blades used in this work were donated by IBM Corporation, in the context of a SUR (Shared University Research) grant awarded to Universidade de Évora and CENTRIA.

References

1. F. Rossi, P. Van Beek, and T. Walsh. *Handbook of constraint programming.* Elsevier Science, 2006.

2. A. Van Deursen and J. Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.
3. Gecode Team. Gecode: Generic constraint development environment, 2008. Available from http://www.gecode.org.
4. Pedro Salgueiro and Salvador Abreu. Network Monitoring with Constraint Programming: Preliminary Specification and Analysis. In *Proceedings of the 18th International Conference on Applications of Declarative Programming and Knowledge Management*, 2009.
5. Pedro Salgueiro and Salvador Abreu. A DSL for Intrusion Detection based on Constraint Programming. In *SIN 2010: Proceedings of the 3rd International Conference on Security of Information and Networks*, New York, NY, USA, 2010. ACM.
6. Pedro Salgueiro and Salvador Abreu. On using Constraints for Network Intrusion Detection. In *INForum 2010 - Simpósio de Informática*, Braga, Portugal, 2010.
7. Douglas Comer. *Internetworking With TCP/IP Volume 1: Principles Protocols, and Architecture, 5th edition.* Prentice Hall, 2006.
8. 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.
9. 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.
10. 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.
11. Giorgos Vasiliadis, Michalis Polychronakis, Spiros Antonatos, Evangelos P. Markatos, and Sotiris Ioannidis. Regular expression matching on graphics hardware for intrusion detection. In *RAID '09: Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*, pages 265–283, Berlin, Heidelberg, 2009. Springer-Verlag.
12. 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.
13. C. Schulte and P.J. Stuckey. Speeding up constraint propagation. *Lecture Notes in Computer Science*, 3258:619–633, 2004.
14. P. Van Hentenryck and L. Michel. *Constraint-based local search.* MIT Press, 2005.
15. P. Codognet and D. Diaz. Yet another local search method for constraint solving. *Lecture Notes in Computer Science*, 2264:73–90, 2001.
16. 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.
17. V. Paxson. Bro: a system for detecting network intruders in real-time* 1. *Computer networks*, 31(23-24):2435–2463, 1999.
18. tcpdump web page at http://www.tcpdump.org, April, 2009.