A DSL for Intrusion Detection
Based on Constraint Programming

Pedro D. Salgueiro
pds@di.uevora.pt

Salvador P. Abreu
spa@di.uevora.pt

Universidade de Évora, Évora and
CENTRIA FCT/UNL, Caparica
Portugal

ABSTRACT

Intrusion Detection Systems (IDS) are increasingly important in computer networks, allowing the early diagnosis and detection of anomalous situations, which could otherwise put network performance at risk or even compromise the security or integrity of user data.

In this work we present NeMODe, a domain specific language for network intrusion detection that allows to describe network intrusions that spread across several network packets, relying on Constraint Programming (CP), a programming methodology that starts with a declarative description of the desirable network situations and, based on that description, a set of parameterizations for network intrusion detection mechanisms will execute to find those intrusions.

Categories and Subject Descriptors

C.2.3 [Computer Communication Networks]: Network Operations - Network Monitoring; F.4.1 [Mathematical Logic And Formal Languages]: Logic and constraint programming

General Terms

Security, Languages

Keywords

Constraint programming, domain-specific languages, intrusion detection systems

1. INTRODUCTION

Computer networks are composed of numerous complex and heterogeneous systems, hosts and services. Users rely on networks to perform their regular tasks and have to interrupt these when, for some reason, the network has a problem. In order to prevent computer network degradation or even any failure, the network needs to be constantly monitored in order to diagnose its functionality and integrity. This may be accomplished with a system and network monitor application, such as Nagios [4] or an Intrusion Detection System (IDS) e.g. Snort [11].

On a computer network, some aspects must be verified in order to maintain the quality and integrity of the services provided. The description of some 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 [12] or Constraint Based Local Search Programming (CBLS) [17], enabling the description of these situations in a declarative and expressive way.

To help the description of those network situations, we created NeMODe, a Domain Specific Language (DSL) [16], that enables the easy description of intrusion detection signatures that spread across several network packets, which will then translate the program into constraints that will be solved by possibly more than one constraint solving techniques, including Constraint Based Local Search and Propagation-based systems such as Gecode [15]. It will also have the capabilities of running the several solvers in parallel, in order to benefit from the earliest possible solution.

Besides the task of describing a network intrusion scenario, we need to efficiently solve the CSP which results therefrom. The Cell Broadband Engine, or Cell/BE [8] provides an heterogeneous multi-core, parallel computation environment, on a single chip, which allows one to improve the performance of complex problem solving by means of parallel execution. Network monitoring is a challenging task, as it needs to analyze a large amount of network traffic in real time, a situation which we expect to benefit from the performance increase which can be expected from multi-core processors such as Cell/BE.

Throughout this paper, we mention technical terms pertaining to TCP/IP network packets, such as packet flags, ACK, SYN, RST, acknowledgment, source port, destination port, source address, destination address, which are described in [7].

1.1 Network Monitoring

System and Network Monitor Applications are designed to perform network monitoring tasks on a computer network.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIN’10, Sept. 7–11, 2010, Taganrog, Rostov-on-Don, Russian Federation. Copyright 2010 ACM 978-1-4503-0234-0/10/09 ...$10.00.
These applications are constantly monitoring network devices and services in order to detect possible anomalies. Besides those that perform network service status monitoring, there are applications that focus on traffic monitoring trying to inspect traffic to look for anomalies or undesirable communications, known as Intrusion Detection Systems (IDS).

Intrusion Detection systems is being studied for years in order to prevent and detect intrusions and anomalies in computer networks. Basically, 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 [19]. In this work, we adopted an approach based on signatures.

Snort [11, 5] is a widely used Intrusion Detection System that relies on pattern-matching techniques to detect the network attacks, in particular, Snort uses a multi-pattern matching algorithm, an optimized version of the Aho-Corasick algorithm [14], which allows the efficient detection of multiple network attack signatures at once, without sacrificing the performance of the system.

Snort is a very efficient Intrusion Detection System but it is primarily designed to detect network attacks which have a signature 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.

There are some pre-processors for Snort that help to relate separate network packets; Streami 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 the rule that describes the desired signature. The Flow pre-processor also allows snort rules to relate with other rules by using the flowbits keyword. With this pre-processor, one rule can set some flag, later other rule can check if that flag is/ is not 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.

There has been a lot of work in the area of Intrusion Detection Systems, but most of it consists on the development of new, faster and more efficient methods for detecting the network signatures of the network attacks, sometimes implementing the pattern-matching algorithms in dedicated hardware to speed up the detection, in order to be able to keep up with the increasing speeds of today’s computer networks [3]. In [14], the authors present a hardware implementation of a multi-pattern matching algorithm, using FPGA technology for real-time network intrusion detection in fast computer networks. [18] presents a work that implements regular expression matching using graphics hardware (GPUs) to perform intrusion detection.

Although most of the recent work in Intrusion Detection Systems consists on improving the performance, there are other works focused on developing new methods to match the network packet signatures and enabling the detection of other types of signatures. In [3], an algorithm and an implementation method for performing flow aware content search based on Bloom Filters is presented. This approach allows one to search for signatures that spread across several packets, which is one of the limitations of Snort and most other intrusion detection systems. In [10], 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.

1.2 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) [12], 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 constrains, 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.

Constraint Based Local Search

Constraint Based Local Search (CBLS) [17] is a fundamental approach to solve combinatorial problems such as Constraint Satisfaction Problems. CBLS is a method that can solve very large problems but its not a complete algorithm, and is unable to provide a complete or optimal solution.

Usually, this approach starts with an initial, tentative solution to the problem, which is iteratively improved though minor modifications until a termination criterion is satisfied.

Adaptive Search

Adaptive Search (AS) [6] is a Constraint Based Local Search algorithm, taking into account the structure of the problem and using variable-based information to design general heuristics which help solve the problem.

Adaptive Search iteratively repairs the tentative solution, trying to reduce the error functions used to model the problem, in order to obtain a valid solution to the problem.

Adaptive Search receives as input a set of variables and their associated domains, a set of constraints with the associated error functions, a function to project constraint errors of each variable and an objective function to minimize. Its output is an assignment of values to variables, which is a valid solution to the problem, i.e. on for which all constraints are satisfied.

Adaptive Search is a good algorithm to detect network intrusions, as a solution to an intrusion detection problem is a subset of the packets seen on the network traffic, and a solution to a problem modeled in Adaptive Search is an ordered permutation of the domain of the problem, which, when applied to the intrusion detection domain, will be the network traffic window. Adaptive Search has recently been ported to Cell/BE, presented in [2].

Gecode

Gecode [13] is a constraint solver library based on propagation [12], implemented in C++ and designed to be interfaced with other systems or programming languages.

Using Propagation-Based constraint solving, the problem is described by stating constraints over each variable that comprises 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.

2. NETWORK MONITORING WITH CONSTRAINTS

Our approach to intrusion detection relies on being able to describe the desired signatures through the use of constraints, identify the set of packets that match the target network situation.

The problem needs to be modeled as a Constraint Satisfaction Problem (CSP) in order to use the constraint programming mechanisms. Such a CSP will be composed of a set of variables, \( V \), representing the network packets, their associated domains, \( D \), and a set of constraints, \( C \) that relates the variables in order to describe the network situation. On such a CSP, each network packet variable is a tuple composed of 19 \( i \) integer values which represent significant fields of a network packet, necessary to intrusion detection.

The domain of the network packets, \( D \), are the values actually seen on the network traffic window, which is a set of tuples of 19 integer values, each tuple representing a network packet actually observed on the traffic window and each integer value represents the fields that are relevant to network monitoring. Such a CSP can be represented in the following form, where \( P \) represents a set of network packet variables, \( D \), a set of network packets representing the network traffic window, and \( \forall P_i \in P \Rightarrow P_i \in D \) the associated domains of each variable:

\[
P = \{ (P_{1,1}, \ldots, P_{1,19}), \ldots, (P_{n,1}, \ldots, P_{n,19}) \}
\]

\[
D = \{ (V_{1,1}, \ldots, V_{1,19}), \ldots, (V_{m,1}, \ldots, V_{m,19}) \}
\]

\[
\forall P_i \in P \Rightarrow P_i \in D
\]

After the problem is modeled as a CSP, the model will be solved using a CSP solver to find a solution to such a problem. If a solution found in the domain of the network packets that compose the traffic, that situation was found in the network. So, the solution to such a CSP will be a set of network packets that composes the network attack that was described.

2.1 Modeling a port-scan attack as a CSP

One of the network situations that we have modeled and solved using constraint programming was a port-scan attack. Such a network situation can be modeled as a CSP in the form of \( P = (X, D, C) \), where \( X \) is a set with all the variables that compose the problem, \( D \) the domain of each variable and \( C \) the constraints of the CSP. \( X \) is a set of integer variables representing all the packets described by the CSP. \( X \) can be defined as follows, where \( i \) represents the number of packets that are being looked at:

\[
X = \{ (P_{i,1}, \ldots, P_{i,19}) \ldots, (P_{k,1}, \ldots, P_{k,19}) \}
\]

Consider now the definition of the domain \( D \): let \( D_{(i,n)} \) be the domain of the corresponding variable \( P_{(i,n)} \), \( M \) the set of all packets in the network traffic log and \( k \) the number of packets seen on the network traffic window. We have:

\[
D = \{ D_{(1,1)}, \ldots, D_{(1,19)}, \ldots, D_{(k,1)}, \ldots, D_{(k,19)} \}
\]

\[
M = \{ (V_{1,1}, \ldots, V_{1,19}), \ldots, (V_{k,1}, \ldots, V_{k,19}) \}
\]

\[
\forall D_{(i,n)} \in D, \ \forall P_{(i,n)} \in X, \ \ \ P_i = (P_{(i,1)}, \ldots, P_{(i,19)}) \Rightarrow P_i \in M
\]

The constraints \( C \) that compose a such a CSP are defined as follows:

\[
\forall n = 2 \ast i, \forall n > 0, \forall P \in M \Rightarrow
\]

\[
P_{(n,14)} = 1, \ P_{(n,18)} = 0, \ P_{(n+1,13)} = 1,
\]

\[
((P_{(n,2)} = P_{(n+1,2)}) \land \ldots \land (P_{(n,6)} = P_{(n+1,6)})) \lor
\]

\[
((P_{(n,2)} = P_{(n+1,6)}) \land \ldots \land (P_{(n,6)} = P_{(n+1,11)})), \quad (4)
\]

\[
((P_{(n,0)} < P_{(n+1,0)}) \lor
\]

\[
((P_{(n,0)} = P_{(n+1,0)}) \land (P_{(n,1)} < P_{(n+1,1)})) \quad (5)
\]

\[
((P_{(n,0)} = P_{(n+1,0)}) \land (P_{(n+1,1) - P_{(n,1)} < 500)}) \lor
\]

\[
((P_{(n+1,0)} = P_{(n,0)} + 1) \land
\]

\[
(1000000 - P_{(n,1)} + P_{(n+1,1)} < 500)), \quad (6)
\]

\[
\forall n = 2 \ast i, \forall n > 1, \forall P \in M \Rightarrow
\]

\[
((P_{(i,0)} > P_{(i-2,0)}) \lor
\]

\[
((P_{(i,0)} = P_{(i-2,0)}) \land (P_{(i,1)} > P_{(i-2,1)})) \quad (7)
\]

Some of the constraints applied to network packet variables are temporal constraints, forcing the network packets appear in the right order. Due to this fact, some of the constraints are not applied to the first set of network packets, since there is not a previous set of packets to force a correct order, so, on the listing above, from rule (1) to rule (6), are applied to all constraints to all network packet sets, except the ones that forces the packets to be ordered.

Rule (2) adds constraints so that the first packet of set has the SYN flag set and doesn’t have the ACK flag set.

Rule (3) adds a constraint to the second packet of each set that forces them to have the RST flag set.

Rule (4) adds constraints to force the two packets of the set to be related, so that they make part of a TCP connection, the first one being the one that initiates the TCP connection, and the last one the one that closes the connection.

Rule (5) adds a set of constraints so that the second packet of the set appears after the first one, in a chronological order.

Rule (6) adds constraints to force the maximum distance between the first and the second packet of a set of two packets is less than 500ms.

Rule (7) adds a constraint to all network sets, except the first one, which will force them to be ordered in a chronological order.

3. NeMODe — A DSL TO DESCRIBE NETWORK SIGNATURES

In this work we present a simple, declarative, intuitive Domain Specific programming language [16] for the Network
Intrusion Detection domain called NeMODe. This language talks about network entities, network entity properties and relations of network entities and network entity properties, which allows the description of network intrusion signatures, and with base on those description generate Intrusion Detection mechanisms which can be applied to a computer network and detect the desired network attacks described by their signatures.

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

NeMODe will act as a front-end to several back-ends, where each back-end will be a different mechanism to detect network intrusions, using different intrusion detection mechanisms, which allows to generate several intrusion detection mechanisms to detect attacks, using a single description.

Having a single specification to several constraint solvers allows the search of a solution using different methods of search, allowing to run each of those methods in parallel. Searching the solution for a given network signature in parallel, allows to obtain different results from each constraint solver, possibly some of them are produced faster than others, depending on the characteristics of the problem. Also, some approaches can be better solving some kind of problems than others, so its a good idea to run all methods in parallel, and then use the best solution found, which can simply be the first solution to be found.

NeMODe presents several groups of statements: (1) the primitives of the language, (2) the connectives, (3) definitions, (4) the use of such definitions and (5) macro statements. The primitives are the basic statements of the language, which state simple properties that the network packets should verify. The connectives are statements that relate two or more network entities, forcing them to verify some relations. The definition is a simple way of storing primitives or connectives under a variable and give it a name. The use, as its name suggests, is simply using something previously defined, by referring the variable used on the definition. Finally, the macro statements, are helpers to avoid unnecessary code repetition and ease the description of the signature.

The following list presents the network entities that can be used to describe network signatures in the current implementation of NeMODe:

- network packets, i.e. packet(A)
- source and destination ports, i.e. src_port(A)
- source and destination ip addresses, i.e.source(A)
- time of arrival of a packet, i.e. time(A)

NeMODe is implemented around operations over network entities and their properties, such as operations that force some of the network packet properties to be true or false, operations that allows an easy way to express the relationship between several network entities, such as a network packet to be the one that closes the connection initiated by other network packet, or to state that a network packet should appear a given number of seconds after some other network packet, by using standard arithmetic and logical operations over the network packets.

Much of the network signatures seen in real life applications are based in contents of the network packet, in the packet payload, so it makes sense that NeMODe provides some mechanisms to search string patterns inside the packet's payload, which allows the search of text strings or even binary code that exists inside the network packets. These can (and should) be used with all the other tools provided by NeMODe to describe the network packets properties and relations with the tools that scan the network packet contents.

At the current implementation of NeMODe, this feature is still not available, although it is already implemented at the grammar level, it still does not produces any code.

The following list presents the set of primitives (predicates) available in the current implementation of NeMODe which allows to state properties of network packets that should be verified.

- Force a variable to be a packet i.e. packet(A)
- Force a variable to be a TCP packet i.e. tcp(A)
- Force a packet to have its SYN flag set, i.e. syn(A)
- Force a packet to have its RST flag set, i.e. rst(A)
- Force a packet not to acknowledge any packet, i.e. nak(A)

Follows a list of the connective statements, which are used to relate several network entities.

- Force a packet to acknowledge other packet, i.e. ack(A)
- Restrict the temporal distance between packets, i.e. time(B) - time(A) < secs(500)
- State that there should be a relation between two packets, i.e. related(A,B)
- Force the source or destination port of a packet, i.e. dst_port(A)==22
- Force the source or destination address of a packet, i.e. src(A)==ip(194.132.121.102)

NeMODe provides a special type of statement that helps the users to specify network signatures with minimum work, the definition statements. With these statements it is possible to store a set of properties over a set of network entities and give it a name, being possible to use it later by the given name on any part of the program.

Listing 1 shows an example of a simple definition where some properties over two network packets are stated, in this particular case, the packet A should have its syn flag set and packet A should acknowledge packet B. These properties will be stored in variable C, which can later be used.
The **macro** statements provide mechanisms to help the user describe the situation, by avoiding unnecessary code repetition. This **macro** can be used to repeat a set of properties that were assigned to a variable, and give a name to that repetition, by assigning it to a variable. Other type of **macro** statements are the ones that are applied to the repetitions stored in a variable, such as state the maximum or minimum allowable time interval between each instance of the repetition, or the maximum/minimum overall interval time that a repetition can take. Listing 2 illustrates a simple use of this **macro** functions.

The following is a list of the **macro** statements available, in the current implementation of **NeMODe**.

- Repeat a set of properties \( N \) times, and give it a name, \( R \), i.e. \( R := \text{repeat}(3, C) \)
- State the maximum/minimum time duration between the first and last instance of a repetition of a set, i.e. \( \text{max\_duration}(R) < \text{secs}(60) \)
- State the maximum/minimum time interval between each instance on a repetition of a set, i.e. \( \text{max\_interval}(C) < \text{secs}(60) \)

When using the **repeat** statement and assigning it to a variable, each instance of the repetition as well as its variables keeps accessible, referring it as the \( nth \) instance and then referring the variables of such instance. Listing 3 shows such an example, where the statement \( \text{nak} \) is applied to variable \( A \) of the first instance of the repetition \( R \)

### Listing 3 Accessing a variable inside a **definition**

```plaintext
C = {
    packet(A), tcp(A),
    syn(A),
} C
R := repeat(C, 3),
nak(R[1]:A)
```

#### 3.1 Available back-ends

A significant characteristic of **NeMODe** is that it is a front-end to several back-end implementation mechanisms, which in turn effect the detection of network intrusion signatures, thus allowing one to have a single specification of the network intrusion signatures that can be mapped into several detection mechanisms, which may then be run in standalone or in competitive parallel mode.

At the moment, two back-end detection mechanisms are implemented; (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, as well as the description of the signature is made. In Sec. 1.2 each of these approaches are explained.

### 3.2 Structure of a **NeMODe** program

An input program written in this language is composed by 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.

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

#### 3.3 Examples

So far, we have worked with some simple network intrusion signatures: (1) a port-scan attack, (2) an ssh password brute-force attack 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.

**Port-scan attack**

Listing 4 shows a simple example of **NeMODe**, which describes a port-scan attack. From line 1 to line 11, the signature of the attack is described, from line 13 to line 15 are the “semantic actions” to be taken if the attack pattern is found on the network. Lines 2 to 9 constitute the definition of a set of rules, grouped under a **block** which has been named \( C \). Line 10 states that the packets that match the rules in block \( C \) are expected to occur 5 times. This sets of occurrences are assigned to variable \( R \). Furthermore, in line 11 we state that the maximum distance between each instance of those blocks in repetition \( R \) should be less that 500 micro-seconds.

### Listing 4 A port-scan attack using **NeMODe**

```plaintext
1:  portscan {
2:      C = {
3:         packet(A), packet(B),
4:         tcp(A), tcp(B),
5:         syn(A),
6:         nak(B), rst(B),
7:         time(A) - time(B) < usecs(100),
8:         related(A,B)
9:      },
10:      R := repeat(5,C),
11:      max\_interval(R) < usecs(500)
12:    } => {
13:      drop(D),
14:      src(D) == src(R[1]:A),
15:      drop(D)
16:    };
```
SSH brute-force password attack

Listing 5 shows another simple example NeMODe that describes an SSH password brute-force attack. The main differences from this example to the previous one, presented in Listing 4, is in line 5, that states that the destination port of packet A should be 22.

Also, in line 10, max_duration(R) < secs(60), it is introduced a new property which states that the maximum duration of the 3 repetitions defined in line 11 (R) should should be less than 60 seconds, meaning that the time interval between the first packet of the first instance of block R and the last packet of the last instance of block R, should be at most 60 seconds.

**Listing 5 An ssh brute force attack using NeMODe**

```
1: ssh_brute_force {
2:  C = {
3:      packet(A),
4:      tcp(A),
5:      dst_port(A)==22,
6:      syn(A),
7:      nak(A)
8:    },
9:  R := repeat(3,C),
10:  max_duration(R) < secs(60)
11: } => {
12:    packet(D),
13:    src(D) == src(R[1]:A),
14:    drop(D)
15:  };
```

SYN flood attack

**Listing 6 A SYN flood attack using NeMODe**

```
1: ssh_brute_force {
2:  C = {
3:      packet(A),
4:      tcp(A),
5:      syn(A),
6:      nak(A)
7:    },
8:  R := repeat(30,C),
9:  max_interval(R) < usecs(500)
10: } => {
11:    packet(D),
12:    src(D)==src(R[1]:A),
13:    drop(D)
14:  };
```

Listing 6 shows how a SYN flood attack can be described using NeMODe. Line 3 states that A should be a network packets, in line 4 is stated that they should be tcp packets in line 5, a constraint is applied to state that packet A its SYN flag set and in line 6, packet A is forced to not acknowledge any other packet. In line 8 and 9 it is stated that the network packets that match the properties defined in block C, defined in line 2, should appear 30 times, and the time interval between each occurrence should be inferior to 500 micro-seconds.

From line 11 to line 13 the action to take is described, if the intrusion id found. In this case, all packets that have a source address equal to the source address of the attacker will be dropped.

### 3.4 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, having different ways to express the problems and different ways to find a solution to the problems. So, it is necessary to have several code generators for each of back-end available, one for Gecode and one for Adaptive Search. Although these tools are completely different, we were able to minimize this difference by creating custom libraries to each of those constraint solvers so that the code generation process would not be completely different for each back-end.

Standard programming language implementation tools were used to implement parts of NeMODe, such as Flex and Bison that were responsible to make all syntactic and semantic analysis of NeMODe program. To generate the code, we choose GNU Prolog, as it presents very good characteristics to rapidly implement prototypes as well as a great flexibility to work and manipulate tree data-structures like, such as the Abstract Symbol Table, essential to generate code. So, Bison generates GNU Prolog code representing the Abstract Symbol Table, which is then analyzed in GNU Prolog, generating source code for Adaptive Search and Gecode.

The Bison and Flex parts of the code generation are the same to either of the back-end mechanisms, as its only task is to assure that the input code is correct and generate an Abstract Symbol Table. After that, the code generation is specific to each back-end. Follows the description of these task to each of the back-ends.

**Generating Adaptive Search code**

The task of generating Adaptive Search resumes to create the proper error functions that are necessary so that Adaptive Search be able to solve the problem; the cost_of_solution and cost_on_variable.

In order to ease the generation of this error functions, a small 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 used to generate the cost_of_solution and cost_on_variable.

Listing 7 shows an excerpt of the cost_of_solution function for Adaptive Search generated code for the port-scan attack, described in Listing 4.

**Listing 7 Adaptive Search, cost_of_solution code excerpt**

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

Listing 8 shows an excerpt of the cost_of_variable function for Adaptive Search generated code for the port-scan attack, described in Listing 4.
4. SEARCHING INSIDE A PACKET

Many network signatures are based on the contents of the network packet, in the packet payload [7], so it makes sense to be able to search string patterns inside the packets, allowing the search of text strings or even binary code on the packet content’s. This search mechanism should be used with the detection mechanisms described in this work to complement each other in order to create a more powerful and complete tool to perform network intrusion detection.

The current state of this work does not have the capabilities performing this kind of search, but we have already done some experiments also using the Constraint Programming paradigm to perform these searches. We tried HAMPI [9], a solver for string constraints, which produces a string that matches the constraints that were set. This solution revealed a poor performance when compared to the rest of the detection mechanisms implemented in this work, and if implemented and combined with the rest of the work, it would degrade the performance in a big scale.

We are now looking for a different approach to search inside the packets payload.

5. EXPERIMENTAL RESULTS

While developing this work, several experiments were done. We have tested the examples of Sect. 3.3, a port-scan attack, an ssh brute-force password attack and SYN flood attack. All these network intrusions were successfully described using NeM0DE and valid Gecode and Adaptive Search code were produced for each of all three network signatures.

The code generated by NeM0DE to each of the presented network situations was then executed in order to validate the code was valid code and 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, presented in [2], and we choose this version of Adaptive Search to run our experiments, forcing us to use the Qs21 dual-Cell/BE blades. This machine is incompatible with the implementation of Gecode, since its a x86 implementation and the machine is a Cell/BE architecture, so, we choose a Proliant DL380 G4 to run the code generated for Gecode.

The modeling of network situations uses two types of constraints, on one hand the ones that restrict the network packet to have some special characteristics, and on the other, those that impose temporal restrictions between network packets. Using these two types of constraints allowed us to divide the problem in two stages. In a first stage, specific constraints are applied to the network packets to filter the ones that have some specific properties, filtering the network packets that match the desired signature, excluding the temporal constraints. On a second stage is applied a set of constraints to the result of the first stage, which impose temporal constraints between some of the network packet variables in order to obtain a final solution to the problem, and thus, detecting the desired network intrusion signature, if it exists.

The two approaches were implemented using slightly different methods, while in Gecode the problems are modeled and solved in a single stage, in Adaptive Search they are modeled and solved using two stages.

We have also tried to swap the order of the 2 stages in Adaptive Search, but this solution revealed to be worse, since the first stage was producing too many results due to the existence too many network packets that matched the temporal constraints.

In all the experiments we used log files representing network traffic which contains the desired signatures to be detected. This log files were created using tcpdump [1], which is a packet sniffer, during an actual attack to a computer,
which was induced to simulate the real attacks described in this work.

**Port-scan attack** For the Port-scan attack, we have created a log file composed of 400 network packets while a computer was being under a Port-scan attack, which was used for both intrusion mechanisms, Gecode and Adaptive Search. We used a signature describing 52 packets, so a solution to this problem would be a set of 52 packets from a set of 400 network packets.

**SSH password brute-force attack** For the SSH password brute-force attack, we created a log file composed of 182 network packets while a computer was trying to brute-force password guessing the password of an SSH server running on another computer. This log file was used in both Gecode and Adaptive Search versions of the SSH password brute-force attack. We used a signature describing 5 network packets, so a solution to this problem would be a set of 5 packets from a set of 182 network packets.

**SYN flood attack** A log file of 100 network packets was created for the SYN flood attack a computer was under an actual SYN flood attack, which was used in both Gecode and Adaptive Search versions of the SYN flood attack detection mechanisms. We used a signature describing 30 network packets, so a solution to this problem would be a set of 30 packets from a set of 100 network packets.

### 5.1 Results

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

<table>
<thead>
<tr>
<th>Intrusion to detect</th>
<th>Gecode(seconds)</th>
<th>Adaptive Search(seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Port-scan</td>
<td>0.1273</td>
<td>0.6749</td>
</tr>
<tr>
<td>SSH password</td>
<td>0.0126</td>
<td>0.6125</td>
</tr>
<tr>
<td>SYN flood</td>
<td>0.0566</td>
<td>0.0466</td>
</tr>
</tbody>
</table>

### 6. EVALUATION

The performance of the prototypes described in Sec. 5 shows a multitude of performance numbers relative to the intrusion detection mechanisms used for each network signature.

Looking closely to the results in Table 1, it is possible to see that Gecode performs better than Adaptive Search. This difference is explained by the fact that Adaptive Search needs a very good heuristic functions to improve its performance. We created some heuristics based on the network situations we are studying which improved the performance of Adaptive Search, but still can’t reach the performance of Gecode.

Even with a perfect heuristic of Adaptive Search, the results obtained are quite encouraging. As for Gecode, the results obtained are quite good. With these results, we are now ready to start the detection of intrusions in real network traffic instead of log files.

As for NeMODe, it revealed to be a success, since it was possible to easily describe all the three network intrusions and generate valid code that could detect the desired network situation.

### 7. CONCLUSIONS AND FUTURE WORK

The work presented in this paper presents NeMODe, a Domain Specific Language to describe network intrusion signatures that generates intrusion detection mechanisms based on Constraint Programming, more specifically, using Gecode and Adaptive Search.

The results obtained in this work show that it’s possible to transform the description of a network situation into several intrusion detection mechanisms, based on Constraint Programming, from a single description and then use those mechanisms to detect the desired intrusion using the generated code, demonstrating the viability of using Constraint Programming in network monitoring tasks.

We have also shown that it is feasible to use Adaptive Search running on a Cell/BE microprocessor as well as use Gecode to monitor and perform intrusion detection in a computer network.

Also, we showed that we can easily describe network signature attacks that span several network packets, which is somewhat tricky or even impossible to do using systems like Snort. While a port-scan attack can be detected in Snort, its signature can not be described, it is necessary to use to the already built pre-processors.

This work is still at an early stage of development, we expect there to be plenty of room for improvement: we have reached an efficiency level that may be suitable to start performing network monitoring tasks on live network traffic link, meaning that and important step will be to apply this method in a real network to assess its performance.

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.

NeMODe will be extended to be more flexible, allowing to describe other network properties and a broader range of attack signatures. Also, it will include more back-ends, allowing the detection of intrusions using several methods using a single description.

### 8. 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. The work described herein was partly supported by the FCT/Egide Pessoa bilateral grant “CONTEMP”.

### 9. REFERENCES


