Network Intrusion Detection with Constraints

Pedro Salgueiro
Universidade de Évora and CENTRIA FCT/UNL
Portugal
email: pds@di.uevora.pt

Salvador Abreu
Universidade de Évora and CENTRIA FCT/UNL
Portugal
email: spa@di.uevora.pt

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

***Index Terms*—Constraint Programming, Intrusion Detection Systems, Domain Specific Languages**

I. 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] or Constraint Based Local Search Programming (CBLS) [2], 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) [3], which enables an easy description of intrusion signatures that spread across several network packets, which 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 [4]. It will also have the capabilities of running several solvers in parallel, in order to benefit from the earliest possible solution.

Throughout this paper, we mention technical terms pertaining to TCP/IP and UDP/IP network packets, such as *packet flags, ACK, SYN, RST, PSH, URG, acknowledgment, source port, destination port, source address, destination address, payload*, which are described in [5].

This paper is organized as follows. Section II 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 III demonstrates how to model and perform Intrusion Detection using Constraint Programming. Section IV details the DSL provided by NeMODe and provides some examples. Section V shows the experimental results obtained by NeMODe. Section VI evaluates NeMODe and Section VII presents the conclusions and future work.

II. State of the art

A. Intrusion Detection Systems

Intrusion Detection Systems(IDS) play an 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 [6]. In this work, we adopted an approach based on signatures.

Snort is a widely used Intrusion Detection System that relies on pattern-matching techniques to detect the network attacks [7]. Snort is a very efficient Intrusion Detection System but 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, such as the `Stream4` and `Flow` pre-processor.

Most of the recent work in intrusion detection systems has been focused on the performance [8], but there has been also some work [8], [9] that focus on the method used to match the network packet signatures and the type of signatures that can be detected, using alternative search methods that allows the search of signatures that spreads across several packets, which is one of the limitations of Snort and most other intrusion detection systems.

B. 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.

Constraint Based Local Search: Constraint Based Local Search (CBLS) [2] 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) [10] is a Constraint Based Local Search [2] 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 [11].

Gecode: Gecode [12] is a constraint solver library based on propagation [1], 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 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.

C. Domain Specific Languages

Domain Specific Language(DSLs) [3] 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 [13], 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.

III. 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 [1] packets and 12 variables for UDP packets [2], which represent the significant fields of a network packet necessary to model the intrusion signatures used in our experiments. For both TCP and UDP network packets, the individual variables of the tuples represent the time-stamp, the source/destination addresses, the source/destination ports and the packet number, used to match the packet with its data. The TCP packets have more significant fields than UDP packets, so these have some more variables, which represents the extra TCP flags and the packet sequence numbers. This number of fields may increase over time with the evolution of the work and the use of more complex intrusions.

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

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

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

Listing 1 Representation of a network CSP

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

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, in a total of z fields for each network of the n packets, 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 packets P. $Data$ is the payloads of the network packets in 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 packets belonging to P 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, $P_i = (P_{j,1}, \ldots, P_{j,z}) \in P$, that models the desired situation, thus identifying the network packets that identify the intrusion being detected.

IV. NeMODe - A DSL to describe network signatures

In this work we present a declarative, intuitive domain-specific programming language [3] for 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. A more complete description of this DSL as well as other examples is presented in [14], which is an extended description of the work described in this paper.

The key characteristic of NeMODe 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.

NeMODe is a front-end to several back-ends, one to each intrusion detection mechanism, allowing to generate several detection mechanisms from 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, which allows to obtain different results from each solver. Depending on the characteristics of the problem, some solvers could produce a better and faster solution that others, allowing to choose the first solution to be produced.

NeMODe presents five 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 each network variable should verify. The *connectives* are statements that relate two or more network variables, forcing them to verify some relations. The *definition* is a simple way of storing primitives or connectives under a variable to be used later. The *use* of definitions, forces a previous definition to used. Finally, the macro statements, are helpers that avoid unnecessary code repetition and ease the description of the signature.

The following list presents the set of primitive (*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 TCP or UDP packet.
- Force a packet to have specific a TCP flag set.
- Force a packet not to acknowledge any packet.
- Force a packet to contain a given string on its payload.
- Force a packet to have a specific src/dst port.
- Force a packet to have a specific src/dst ip address.

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

- Force a tcp packet to acknowledge other packet.
- Restrict the temporal distance between packets.
- Force two packets to be related.
- Force the src/dst port of a packet to be equal to the src/dst port of other packet.
- Force the src/dst ip address of a packet to be equal to the src/dst ip address of other packet.
- State that one piece of payload of a packet should be equal to other piece of the payload of other packet.

NeMODe provides a special type of statements to help users specify network signatures with minimum work, the *definition* statements. These statements allows to store a set of properties over a set of network entities and give it a name and using them later on the program. Listing 2 shows an example of a simple *definition* where some properties over two network packets are stated, in this particular case, the variable A should be a TCP/IP packet, and have its syn flag set. These set of properties are *stored* in variable C, which can later be used. Those definitions by them self don't have any effect, they are only applied when used or referred. In order to use those definitions, simply refer the variable to which the set of properties was assigned or use it in a *macro* statement, explained next.

Listing 2 Example of a definition

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

The *macro* statements provide mechanisms to help the user describe the situation, by avoiding unnecessary code repetition. This *macro* statements can be used to repeat a set of properties assigned to a variable, and give a name to that repetition, allowing future references to each property of each instance of the repetition i.e., `R:=repeat(3,C)`. Other type of *macro* statements are the ones that are applied to the repetitions *stored* in a variable, such as state the maximum/minimum allowable time interval between each instance of the repetition, i.e., `max_duration(R) < secs(60)` or the maximum/minimum overall interval time that a repetition can take, i.e., `max_interval(R) < secs(60)`. Listing 3 illustrates a simple use of this macro functions. Other *macro* statement is the `connection` statement, which forces two network packets to belong to some the same connection, in any direction.

When using the `repeat` statement, as in line 2 of Listing 4, each instance of the repetition as well as its variables keeps accessible, referring it as the *nth* instance and then referring the variables name, i.e., `R[1]:A`. Listing 4 shows an example, where the statement `nak` is applied to variable A of the first instance of the repetition R.

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. II-B each of these approaches are explained.

A. Examples

So far, we have worked with some simple network intrusion signatures: (1) a DHCP spoofing, (2) a DNS spoofing, (3) a SYN flood attack, (4) a Portscan attack and (5) a SSH Password brute-force attack. All of these intrusion patterns were be described using NeMODe and the generated code was successful in finding the desired situations in the network traffic logs. In this paper we present only the DNS spoofing and the SYN flood attack. The Portscan attack and the SSH Password brute-force attack is explained in [14], while the DHCP spoofing attack is explained in [15].

Listing 3 Example of a `macro function`

```
C = { tcp_packet(A), syn(A) },
R:=repeat(3,C),
max_duration(R) < secs(60)
```

Listing 4 Accessing a variable

```
1:     C = { tcp_packet(A), syn(A)},
2:     R := repeat(3,C),
3:     nak(R[1]:A)
```

DNS spoofing: DNS Spoofing is 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 5 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 are the first two bytes of the packet payload.

Listing 5 A DNS Spoofing attack programmed in NeMODe

```
 1  dns_spoofing {
 2      udp_packet(A), dst_port(A) == 53
 3
 4      udp_packet(B), src_port(B) == 53,
 5      dst(B) == src(A), dst_port(B) == src_port(A),
 6
 7      udp_packet(C), src_port(C) == 53,
 8      dst(C) == src(A), dst_port(C) == src_port(A),
 9
10      B != C,
11      data(B,0,2) == data(A,0,2),
12      data(C,0,2) == data(A,0,2)
13  } => {
14      alert('DNS Spoofing attempt')
15  };
```

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 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 6 shows how a SYN flood attack can be described using NeMODe. Lines 2-4 describes a TCP/IP packet with the SYN flag set and assigns set those of properties to variable C. In line 6, the *macro* statement *repeat* is used to repeat the properties of definition `C` by 30 times, and assign those repetitions to variable `R`. Line 7 states that the time interval between each repetition of C should be less than 500 micro-seconds.

Listing 6 A SYN flood attack programmed with NeMODe

```
1  syn_flood {
2     C = {
3           tcp_packet(A), syn(A), nak(A)
4     },
5
6     R := repeat(30,C),
7     max_interval(R) < usecs(500)
8  } => {
9     alert('SYN flood attack attempt')
10 };
```

B. 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.

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.

Generating a Gecode program: This goal is achieved by generating code based on Gecode constraint propagators that describe the desired network signatures. We created a custom library that defines functions that combine several stock Gecode 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.

V. Experimental Results

While developing this work, several experiments were done. We have tested the examples of Sect. IV-A, 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 were produced for all network signatures. The code generated by NeMODe was 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.

Table I
Average time(in seconds) necessary to detect the intrusions using Gecode

Intrusion to detect	Gecode (seconds)
SYN flood	0.0566
DNS Spoofing	0.0069
DHCP Spoofing	0.0082

Table II
Average time(in seconds) necessary to detect the intrusions using Adaptive Search

Intrusion to detect	A.S (seconds)
SYN flood	0.0466
DNS Spoofing	0.3512
DHCP Spoofing	0.3924

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` [16], which is a packet sniffer, during an actual attack to a computer, which was induced to simulate the real attacks described in this work.

DNS spoofing attack: In the DNS spoofing attack, we used tcpdump to capture a log file, composed of 400 network packets, while a computer was under an actual attack. We used Ettercap to perform the DNS spoofing attacks. The attack was programmed in NeMODe, which successfully generated code for Adaptive Search as well as for Gecode and successfully detected the intrusions.

SYN flood attack: In the SYN flood attack a log file of 100 network packets was created with the help of tcpdump while a computer was under a SYN flood attack. The attack was programmed in NeMODe which in turn generated code for Adaptive Search and Gecode. This code was then used to successfully detect the intrusion.

A. Results

Table I presents the time(user time, in seconds) required to find the desired network situation for each of the attacks presented in this work and also for a DHCP Spoofing attack, described in [15], using Gecode. Table II presents the same results using Adaptive Search. The execution times presented in both tables are the average times of 128 runs.

VI. Evaluation

The performance of the prototypes described in Sec. V shows a multitude of performance numbers relative to the intrusion detection mechanisms used for each network signature. Although the tests were executed using two different computers; under this conditions; Gecode usually performs better than Adaptive Search, except in the SYN flood attack. 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. 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.

Even without 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 turns out 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. Although other intrusion detection systems like Snort could detect the attacks presented in this work, they can not describe the problems with the expressiveness used by NeMODe or even relate the several packets that make part of the attack.

VII. 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 is somewhat tricky or even impossible to make using 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] P. Van Hentenryck and L. Michel. *Constraint-based local search.* MIT Press, 2005.

[3] A. Van Deursen and J. Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.

[4] Gecode Team. Gecode: Generic constraint development environment, 2008. Available from http://www.gecode.org.

[5] Douglas Comer. *Internetworking With TCP/IP Volume 1: Principles Protocols, and Architecture, 5th edition.* Prentice Hall, 2006.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] P. Codognet and D. Diaz. Yet another local search method for constraint solving. *Lecture Notes in Computer Science*, 2264:73–90, 2001.

[11] 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.

[12] C. Schulte and P.J. Stuckey. Speeding up constraint propagation. *Lecture Notes in Computer Science*, 3258:619–633, 2004.

[13] V. Paxson. Bro: a system for detecting network intruders in real-time* 1. *Computer networks*, 31(23-24):2435–2463, 1999.

[14] 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.

[15] Pedro Salgueiro and Salvador Abreu. On using Constraints for Network Intrusion Detection. In *INForum 2010 - Simpósio de Informática*, Braga, Portugal, 2010.

[16] tcpdump web page at http://www.tcpdump.org, April, 2009.