COMPILATION FOR INTRUSION DETECTION SYSTEMS

A thesis presented to

the faculty of

the College of Engineering and Technology of Ohio University

In partial fulfillment

of the requirements for the degree

Master of Science

Andrew Lydon

March 2004

This thesis entitled

COMPILATION FOR INTRUSION DETECTION SYSTEMS

BY

ANDREW LYDON

has been approved for

the School of Electrical Engineering and Computer Science

and the Russ College of Engineering and Technology by

Carl Bruggeman

Assistant Professor of Electrical Engineering and Computer Science

R. Dennis Irwin

Dean, Russ College of Engineering and Technology

LYDON, ANDREW. M.S. March 2004
Electrical Engineering and Computer Science

Compilation for Intrusion Detection Systems (231 pp.)

Director of Thesis: Carl Bruggeman

Within computer security, intrusion detection systems (IDSs) are the subject of extensive and varying research. Distributed IDSs have additional research problems. This thesis contributes a way of using compilation of a multi-layered language to simultaneously solve multiple issues confronting distributed IDSs. The target of the compilation is the configuration of existing IDSs with run time support. The language for compilation has two layers: a lower layer for signature and other computationally limited matching including anomaly based matching and a higher layer for general computations. This compiler is implemented and shown to be sufficient to produce arbitrary IDSs using existing IDSs for input rather than custom system software. Graceful degradation and reasonable performance during denial of service attacks have been added on top of existing IDSs using this framework.

Approved:

Carl Bruggeman

Assistant Professor of Electrical Engineering and Computer Science

TABLE OF CONTENTS

APPENDIX

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

TRADEMARKS

Chez Scheme is a trademark of Cadence Research Systems.

N-Code is a trademark of NFR Security, Incorporated.

UNIX is a registered trademark of UNIX System Laboratories, Incorporated.

1. Introduction

Computer security is filled with hype. Many movies, books, and newspaper articles paint doomsday scenarios created by a lack of computer security. The trade press describes computer security as one of the current boom areas, which may be true from a perspective of the money flowing into the field. The reality of computer security is not as glamorous. A study using the data from CERT indicated that for the years 1989-1995, the frequency of attacks relative to the number of machines on the Internet was rather rare and falling [27]. Yet despite this study published in 1997, the media hype has continued.

Much of the interesting computer security research was performed in the 1970s as part of operating system research. Intrusion detection is considered to have been started by either Andersen's 1980 paper [4] or Denning's work [18, 17]. Intrusion Detection became a hot field in the mid to late 1990s and judging by a large bibliography of IDS papers [38], has undergone an exponential expansion in terms of number of papers written and work spent developing IDS systems.

The intrusion detection field can be divided into several subfields. Efficiency and fast IDS work consumes a considerable portion of the energies of the implementors. Many researchers try every feasible approach to IDS detection cores. Producing more accurate detection is an important open problem, but research papers tend to only discuss the core approach. Data fusion between multiple alerts is an important subfield, but only a few solid contributions have been made to it. These are the central problems to producing IDS systems with increased usability.

Distributed and heterogeneous IDS systems have additional areas of research.

Heterogeneous systems of IDSs produce syntax and semantic issues for output compatibility. The syntax issue has been extensively analyzed as part of various standardization efforts. Configuration compatibility has been examined as part of IDS language issues along with the theoretical aspects of the IDS configuration languages. Distributed systems produce the standard questions of scalability, reliability, and error tolerance. All of these problems are important for producing an effective distributed heterogeneous IDS system.

This thesis contributes a language for distributed heterogeneous IDS systems that solves the semantic issues for output compatibility. This language is examined with respect to computability. It contains sufficient abstraction to facilitate the incorporation of work in the efficient IDS, data fusion, and site configuration intrusion detection subfields. A compiler has been constructed that includes features to automate significant portions of the system administration functions necessary for distributed IDS systems. Furthermore, the language is sufficient for developing new types of single node IDS systems without resorting to system programming.

Rather than proceeding from an examination of the existing literature, this thesis can be viewed as analysis of common computer science questions with respect to intrusion detection. These common computer science questions are: how does the system scale?; what happens under high load?; how to distribute the system?; is there an easier way to program them?; and is there an easier way to use them?.

Compared with existing approaches to IDS model translation, this system is substantially more general and allows for more general translation [14]. It automates configuration tasks to a greater extent than [67]. It sidesteps the semantic alert problem by a more useful method than [47].

1.1 Organization

This thesis is organized as follows: Chapter 1 contains a brief introduction; Chapter 2 contains a survey of the background; Chapter 3 contains an outline of the problem, proposed solution and theoretical aspects of the solution; Chapter 4 details implementation results; and Chapter 5 contains the conclusion and suggestions for future work.

2. Background

This chapter contains background information for placing the contributions of this thesis in context. It does not cover every possibly relevant paper and approach. One incomplete bibliography of intrusion detection papers from 1980 to 2001 includes over 600 entries, so it is not feasible to research every slightly relevant paper or approach [38].

The sections of this chapter and the approaches detailed therein are somewhat unrelated. This reflects the trend in intrusion detection research. The field of intrusion detection is sufficiently large that researchers can explore many different questions. Contributions are made to many different aspects of intrusion detection simultaneously. Unfortunately, many contributions are often lost as they are not directly related to the next researchers area of interest.

This chapter starts with a section on the evaluation of IDS efforts. After dealing with classification schemes, it discusses different IDS approaches including implemented distributed IDSs, network layer IDSs, specification based IDSs, rule and finite automata based IDSs, and database based and miscellaneous detection methods. The remainder of the chapter discusses other trends in IDS research including application sensors, autonomous agents, data fusion, plan recognition, and intrusion tolerance.

2.1 IDS Evaluation

Evaluation of the field is important to accurately gauge progress and the current state. Evaluations of IDS systems have found that they do not detect novel attacks well or at all. A second consequence of the evaluations is the importance of the false positive rate. Initially the false positive rate was not advertised for marketing reasons, but in practical operations, it is very important to an effective deployment of IDS sensors.

An objective IDS benchmark is difficult to obtain. Somewhat similar to processor benchmarks, there are many ways of tuning systems to perform well on the benchmarks by making unreasonable assumptions that only hold for the benchmarks [52]. Ignoring the false positive rate or controlling the traffic characteristics are popular ways of obtaining unrealistic benchmark scores.

MIT Lincoln Laboratories performed evaluations of IDSs for DARPA that are considered to be the most complete and objective that have been performed and published [35, 26, 30, 15, 32]. These evaluations were performed in both 1998 and 1999. Many of the failures of IDSs identified in the evaluations have not been fully addressed.

The MIT evaluations effectively took stock of the field, then were discontinued. One conjecture as to why they were discontinued is that by providing a reasonably objective evaluation, it was harder to market IDSs and claim huge gains in research. Basically, because the IDS systems were doing poorly, the evaluations were discontinued. In any case, these evaluations significantly advanced the field by clarifying and standardizing some of the methods of evaluating IDS systems. They also pointed out the large areas for research without placing the blame for failure in those areas on individual vendors.

For the MIT evaluations, a model Air Force bases computer network was created that was intended to be representative of the network traffic on a real air force base. In particular, the following were made to be statistically similar:

> “1. The overall traffic level in connections per day. 2. The number of connections per day for the dominant TCP services. 3. The identity of many web sites that are visited from internal users. 4. The average time-of-day variation of traffic as measured in 15-minute intervals. 5. The

> general purpose of telnet sessions. 6. The frequency of usage of UNIX commands in telnet sessions. 7. The use of the UNIX time command to obtain an accurate remote time reference. 8. The frequency of occurrence of ASIM (An air force network security monitor) keywords in telnet sessions, mail messages, and files downloaded using FTP. 9. The frequency of occurrence of users mistyping their passwords. 10. Inclusion of an SQL database server that starts up automatically after a user telnets to remote server. "[35, p. 166]

Attacks were generated as part of the traffic. The 1999 data included three weeks of training data for training AI based approaches. The first two weeks of training data did not have any attacks or probes. This was to allow for training anomaly based IDSs. Most of the traffic was generated by automatic software. Some of the attacks were run by humans due to the complexity of writing a fully automated attack. See [26] for more details.

The attacks generated for 1999 included many of the 1998 test data attacks and new attacks for 1999. Attacks against Windows NT machines were added in 1999 and were not detected well by any of the IDSs [32]. In 1999, stealthy versions of many attacks present in the 1998 data were added. These were not detected well by any of the systems either. The new attacks developed for 1999 were not detected by any of the evaluated IDS systems [15].

The MIT data has been critiqued for insufficient bandwidth per day when the evaluation results are applied to other sites [37]. The IDSs were evaluated according to generating ten or less false alarms per day. When traffic is scaled up to levels seen at other sites, more false alarms may be generated than may be acceptable. A moderate to low bandwidth model was chosen for the MIT evaluation so that the network data would be of a size manageable for transmission over the Internet.

The proceedings of RAID 2000 published an article critiquing the MIT evaluations

[37]. Although many of the complaints concerning the MIT IDS evaluations are valid, the lack of effective alternatives reduces the impact of the criticism. For example, the MIT evaluation used Receiver Operating Characteristics (ROC) to generate a metric that accounted for the false positives in the detection rate. ROC is significantly more complicated than what is used in practice. Whether or not it is a good metric is a debatable question. Simply critically reading the results of the MIT evaluation should produce a reasonable amount of caution in using the criteria. Pressuring the MIT evaluators to release more information about their tests is the main contribution of this article, but even in this sense it may not have been successful. It appears that this article along with similar complaints by industry have lead to a cancellation of the MIT IDS evaluation efforts.

2.2 Classification Schemes

Classification schemes have been promulgated for the classification of attacks and the classification of IDSs. Attack classification can be done on either the bugs that are exploited or the mechanisms involved in the attack. Classification of IDS systems are useful for familiarization with the state of IDS research.

The result of classifying the attacks by the bugs that are exploited produces insight into the types of software failures that can be avoided by systematic engineering strategies. For example, buffer overflow attacks do not occur for languages like JAVA that do bounds checking on every access. It illustrates that in at least this aspect of security, choosing JAVA for the programming language will prevent the problem from occurring in practice.

Classification by exploited bugs is useful for improving the software that is exploited, but it is not as useful as other classifications of security bugs for the use of intrusion detection. The limitation is that even when the system bugs stem from similar software bugs they may need different techniques for detecting their exploitation. For example, corner cases may have been improperly handled in two system areas. Detecting one of the corner case exploits may require a host based IDS while detecting the other may require a network based IDS to examine the traffic. From a software engineering perspective, both bugs are very similar. But from a intrusion detection perspective, the bugs are dissimilar.

As part of the 1998 DARPA IDS evaluation, a large number of computer attacks were categorized to both classify the performance of IDS systems on these attacks and ensure a realistic distribution of attack types. The attack classification taxonomy was used by Kristopher Kendall to give an attack database of sufficient size for the DARPA evaluation. The taxonomy is based on four factors: first, the privilege levels involved in the attack; second, the "methods of transition or exploitation"; third, the "transitions between privilege levels"; fourth, the actions of the attacker [30, 69].

John Howard produced a taxonomy that was used to classify several thousand actual intrusions using the CERT data. His taxonomy uses five characteristics to classify the attack: type of attackers, tools used, type of access, result of attack, and the objective of the attack. This taxonomy is useful for examining entire intrusions rather than single attacks. The taxonomy is based on the process of the attack rather than the specific computer mechanism used in the attack. This is a broader criteria than is needed for the analysis of computer weaknesses, but is very useful when examining the context and effects on corporations and people. His work also includes a discussion of some of the other taxonomies used for attacks [27].

Stefan Axelsson has an extensive classification of intrusion detection systems [7]. His taxonomy for IDSs first differentiates IDSs by the detection technique, then it differentiates by particular system characteristics. The system characteristics used for secondary differentiation are near real time detection, batch or continuous data processing, source of audit data, response type to intrusions, locus of data processing and data collection, security of the IDS itself, and the interoperability with other IDSs. This taxonomy is useful for both choosing an IDS and examining the IDS field for the techniques that have been attempted [7].

2.3 Implemented Distributed IDSs

Implemented distributed IDSs give a picture of the state of large scale intrusion detection. If a highly effective reasonably priced distributed IDS could be developed, many problems in large scale intrusion detection could be addressed once for the system. As implemented distributed IDSs relate to this thesis, the problem of a lack of standardized semantics for alerts does not have to be addressed if only one system is used. Thus producing a single distributed IDS system is in some ways a simpler problem. Commercial companies have a significant stake in ensuring that for distributed IDS systems, their single system is chosen. This section discusses the common but crucial shortcomings of these systems then briefly discusses some of the most advanced implemented research distributed IDS systems.

The common shortcomings of these systems are in the interconnections between nodes and the resources used by the system. Some systems try to only give an interface, but do not specify the topology of the system. The topology has a significant impact on the reliability of the system and on the network resources that are used by the system. Without specifying a topology further analysis cannot be performed. Some system specify a hierarchical tree topology, which if the internal nodes do not have a way of consolidating the alerts, can lead to a DOS of the root node. The network resources and scalability of distributed IDS systems has rarely been addressed. If an implemented system is to scale well, the scalability should be planned for in the design.

The interconnections for a system that attempts to correlate attacks across multiple sensors are a critical design decision. It requires trade-offs or at least carefully documented engineering. If the relevant data is sent, then all of the attacks may be able to be detected, but at the possibility of a bottleneck. If it is not sent, then there are holes in the system coverage. Many of the research approaches chose to take the bottleneck over the possibility of incomplete coverage.

EMERALD [48, 41, 64, 65, 47] from SRI International is the latest generation of distributed IDS in the development of IDES and NIDES. EMERALD is arguably the most complete advanced intrusion detection system. Much of the professional team has extensive experience in intrusion detection. It uses both signatures in an expert system and statistics to detect attacks. Considerable portions of EMERALD have a well defined API with multiple modules. Indeed, the EMERALD software has been significantly developed and even deployed in a few instances. EMERALD has used as the core IDS for later research resulting in IDS software by SRI. For example, the alert correlation research performed by SRI was added to the EMERALD software for testing.

The architecture of EMERALD is relatively straightforward compared to many of the other architectures proposed for distributed IDS. This has several benefits: easier for implementation, easier to analyze and predictable.

EMERALD addresses scalability by facilitating interoperability. It predated the IETF intrusion detection standardization effort and later was modified to be compatible. Although the connection mechanism is defined, the connection topology is left unspecified. Most likely existing deployments configure it as a hierarchy. Distribution of messages in an EMERALD system is subscription based. A subscription is configured at start up time by the node. The detection framework is claimed to be general enough to be used to operate with the output from other nodes, hence is a meta-level framework. It is not clear if it is ever actually configured to be used as a meta-level framework, or if this were an option that they wanted at system creation time that turned out to not be useful.

CARDS [72] is the implementation of ideas in "Abstraction based Intrusion Detection in Distributed Environments" [44]. The main idea in this signature based approach is to automatically break the signature down into smaller components that can be distributed and recombined to match the signature.

The approach is interesting, but the technical treatment is difficult to follow. It may be a promising approach, when and if many issues get resolved. It is not yet sufficiently developed to facilitate an analysis of network performance or other distributed performance issues. It is not clear how it performs for denial of service attacks or other overload attacks. The implementation as described does not appear to be complete, indeed a paragraph in the back of the CARDS paper implies that it is only underway, but not finished. The interconnections between nodes are glossed over.

GrIDS [59] is a graph based IDS. It attempts to fuse together the information from various nodes into a graph of the entire activity of the system. This allows large scale attacks to be easily noticed. Because of the similarity of this approach to network management, it can have an easy to understand user interface and would be straightforward to integrate with network visualization and monitoring tools. Unfortunately, the framework is not powerful enough to build an IDS with a high level of coverage. Nor is it useful for most automated monitoring tasks, with a few exceptions: the detection of worms, the detection of network scans, and the detection of telnet chains. Extensive scalability has not been explicitly addressed.

JiNao [29] is an IDS system that is designed for use with OSPF network routing. It attempts to determine both when a router has been compromised and which router has been compromised. It uses both misuse detection and anomaly detection. Automatic responses based on this information have been partially implemented.

As the JiNao is described in paper [29], it is not clear that it is an airtight solution. It seems to rely on a process not getting compromised, but if someone chooses to read data from that process, then the security features may be overridden. Assuming that it works properly, it is another way of detecting intrusions in routers combined with some avoidance mechanisms.

Snort has been modified to perform as a distributed intrusion detection system as part of a masters thesis [22]. It was modified so it sent out IETF standard intrusion detection messages. A primitive central monitoring and collection system was added.

The goal of the project was to produce a useful system, rather than perform pure research.

ASAX was an early host based IDS system. It ran on SunOS under the C2 security logging level. The implementation is described in detail and appears to be fairly complete. It has distributed data collection. As described in [40], only a few rules were specified to scan for and the rules were of a restricted scope. Although these limits on the rules limit the initial usefulness of the system, they probably significantly contributed to easing the implementation effort, which was completed [40].

Although not describing an individual distributed IDS system, the research in "Designing a Web of Highly Configurable Intrusion Detection Sensors" [67] deals with some of the problems in distributed intrusion detection systems. The system described in the paper aims to simplify dynamically configuring a large number of intrusion detection sensors. This is a useful task for the system administrator. The crux of the system is modeling the dependencies of the IDSs in a database. Then when a configuration is proposed or changes are proposed, it can be checked to ensure its validity and generate an ordering for the activation of the sensors.

2.4 Network Layer IDS

The fastest network IDS systems only process the network layer. By not reconstructing TCP and often not even examining the payloads of the packets, it is possible to build IDS systems that can keep up with very high data rates. Finite automata, neural networks, and other AI techniques have been used for the core detection engine.

A particularly high performance network IDS is described in [56]. The results that are claimed in this paper need to be carefully quantified. For example, it is claimed that the system has "real-time performance at up to 500 Mbps even when run on a standard PC." Later the result is qualified out to be "only the time spent within the intrusion detection system." Qualitatively, this means that I/O takes the majority of the time, hence a speedup by even a factor of two cannot be obtained by further optimizing the IDS with respect to pattern matching assuming that the same data is needed in a similar order.

The approach to the specification of rules is interesting. The rules are regular expressions that are compiled into finite automata. Rather than build the knowledge of the protocols into the compiler which outputs C++, it is left to be specified in the rule files. Although it is claimed that this makes it easily extensible, this makes the specification of rules depending on TCP stream reassembly difficult to impossible. In particular, all of the rules specified in their implementation are low level network attacks.

INBOUNDS is a network based IDS system developed at Ohio University in Athens, Ohio. [62, 10, 51, 9, 50, 12] INBOUNDS stands for the “Integrated Network-Based Ohio University Network Detective Service.” Although INBOUNDS is technically a framework for detection modules, currently the core detection module is an anomaly detector using self organizing maps (SOM). A self organizing map is an AI technique that involves training a function for data clustering. The data that the system examines are the length and timings of individual TCP sessions.

SRI has a Bayes network based approach to intrusion detection at the TCP transmission level [64]. It does not reconstruct the TCP session, but does examine the TCP headers. The results of the component against the MIT data indicated that although the system proved effective against DNS, it is only “moderately effective against stealthy probe attacks”.

2.5 Specification Based IDSs

Specification based IDSs work by specifying the allowable system calls for every executable file. Each actual system call is checked against the allowable parameters to determine if an unallowable behavior has taken place or to prevent it from taking place. For example, it may be specified that `in.fingerd` cannot `exec /bin/sh`.

Either at run time when `in.fingerd` attempts to `exec /bin/sh`, it is prevented and an alarm is generated or through later log post processing it is found that `exec /bin/sh` by `in.fingerd` took place and an alarm generated.

These approaches require the specifications of abnormal behavior to be specified in advance, usually by the application developer. For run time specification based systems, which are the most interesting, each system call is checked against these specifications before the call is processed. For example, it may be specified that the `ps` command cannot `exec /bin/sh`. At run time this is supposed to be verified.

The advantage of this approach is that some internal bugs that cause security flaws can be detected and contained on software that the system administrator may not have the source to. It may also be simpler to implement for some cases, assuming that the framework for this method has already been implemented.

The disadvantages of specification based IDSs are more prominent. Checking specifications on every system call would slow the system down. Although it may be an insignificant amount for a short list of specifications, a long list could prove to be a significant bottleneck. Considering the level of detail of some of the proposed specifications, for example the process can not write to `/etc/passwd`, the specification list has a large potential to bloat.

Furthermore, not all attacks can be specified with most of the proposed specification languages. For example, exploiting race conditions, like the binmail exploit, cannot be done in these stateless specification languages. The addition of state would significantly increase the complexity of the specifications.

Often it would be simpler to have the application developers simply test their software for the possible failure points. One advocate of specification based IDS states that reasonable specifications can be developed in the tens of man-hours for most applications. In this same amount of time, automated tests for corner cases and manual tests for possible security failure points could be developed. If the specifications are too loose, more security holes are possibly left uncovered, but if the specifications are too tight, the application cannot work as desired. In any case, some types of security flaws fall closer to misuse. For example, a text editor should not be banned from editing `/etc/passwd`. It is more likely to be a system administrator rather than a cracker.

Uppuluri and Sekar built a specification based IDS and tested it in both the 1999 DARPA on-line evaluation and against the 1999 off-line DARPA/MIT Lincoln Labs data set [63]. Their system could not detect all of the attacks. Some of the attacks could not be detected due to falling under misuse rather than violations. For example, attempts to try the default guest account were considered to be within the normal operation of telnet, but according to the standard interpretation of the MIT data, were to be considered attacks. This was remedied by the questionable method of adding specifications to disallow this normal behavior. Their system did not suffer from any false positives when run over the test data. It has conjectured by multiple researchers that specification based IDS will have a low false positive rate. But this is probably only true when the specifications are carefully drawn up by an expert.

The specification language developed for their IDS uses regular expressions with negation. The "string" is matched by matching events in a temporal fashion. Their paper includes examples and their development methodology for producing specifications [63].

Calvin Ko and others attempted to make the specifications more general than the form that they often currently take [31]. The large area of concern that they intend to address is the generality of attacks. For example, Snort signatures are strings that are matched against the data in network packets or in a reassembled TCP stream in more recent versions. The signature that is published is representative of the most common compilation of the attack. If the cracker were to insert extraneous assembler instructions or use different registers, the string would not match. This is common for virus software, but is relatively new for intrusion detection [57]. His implementation does not seem to significantly vary from other specification based approaches [31].

2.6 Rule and Finite Automata Based IDS

The PhD thesis of Sandeep Kumar is considered to be the seminal exponent of signature based IDS [34]. In this work, Kumar classifies many of the types of signatures for detectable intrusions with respect to the complexity of the pattern matching used. By giving examples he illustrates that the classification is meaningful. He is explicit in his treatment of time.

His hierarchy is

1. Existence (for example, SUID with poor permissions)
2. Sequence (for example, race conditions)
3. Regular expressions
4. Other

His work includes a method for quickly matching signatures up to regular expressions that was fully developed by other graduate students at Purdue into the IDIOT IDS.

Related to Kumar's work is the approach of Alessandri to use the features of the attacks to classify IDSs [2]. This classification is used to analyze where the strengths and weaknesses in the system are. Then it is proposed to use multiple diverse IDS systems to blend together coverage. Other failure points of IDSs are discussed, for example rate overload, but not addressed in the evaluation language.

A fast IDS system for audit trail analysis is described in [66]. The emphasis of the approach is on resource usage and speed. The rules are coded in a limited pattern matching language which is then translated to C and compiled. The processing of the logs is done on each machine and then fed into a server which groups them. The implementation is in a prototype stage after the successful completion of some feasibility tests. It is an open source project and like other open source projects, it is not clear if it will get to the point at which it can successfully approach the state of the art. It looks to be the development of a limited tool. That is certainly not anything bad, but does not really qualify as research.

Kruegel and Toth distribute signatures among many nodes to obtain fault tolerance in the IDS. Their approach does remove many node failure points, but network bottlenecks and scalability are not addressed [33].

Kruegel and Toths approach may fall prey to the same scalability problem as CARDS. Toward the end of the paper are both theoretical and experimental results pointing toward the failure points in scalability [33]. Judging by their network traffic formulas, their decentralized solution will consume an enormous amount of traffic for complex patterns. They claim that the patterns are usually very simple, which may be a correct claim, but it will be difficult to tell unless one attempts to make a reasonably complete set of patterns. Even assuming simple patterns, it would probably exacerbate a DOS attack. They have a term for the number of network messages that is quadratic in the number of events per day or other time segment. This would cause intensive network congestion from even a small DOS attack designed to trigger this. It is unclear how the system would be able to recover.

LAMBDA is an attempt to generically model signature attack rules [14, 39]. More specifically, it is the first attempt described by the authors, to put forth a declarative language that can model attacks in an overarching manner. The examples indicate that there is a considerable amount of art involved in choosing the signatures and rules for modeling an attack. In particular, it will probably give worse performance than if the rules were developed for a single IDS. Current IDS performance indicates that the rules are a problem area. Too many false alarms indicates that the rules must address fewer cases. The failure of many of the rules on slow and stealthy attacks as indicated by the MIT evaluation data shows that the rules need to be broadened in some aspects.

The language in LAMBDA is built by combining primitives by temporal and boolean operators. Clock skew is not addressed with the temporal operators. The goal of the signatures is to model the pre-conditions and post-conditions of an attack, but the examples indicates that everything but the final result of the attack is inserted into the pre-conditions of the attack. Thus in practice, this is nothing but waiting for a match of all of the pre-conditions. Hence this boils down to a restriction to existence or possibly regular expression based signatures.

Later modifications to LAMBDA concentrate on technical aspects of the language. They do not attempt to broaden the language to model more, but rather attempt to more exactly encapsulate the current attacks with respect to the optimizations that can be performed for their detection [39].

Another approach to an overarching declarative signature based language is described in [49]. "This article firstly proposes Sutekh, a declarative signature language providing a combination of functionalities at least as complete as the union of what is offered by other declarative systems.... Sutekh provides sequence, alternative, partial order, negation, event correlation by logical variables, condition verification and alert triggering" [49, p. 1-2]. The paper goes on to give ways to translate these signatures for use by other systems, in particular ASAX and P-BEST, which is the expert system in EMERALD. Although all of these rules can be translated to other other system, the converse is not true. In particular, since P-BEST is an expert system, the order of the rules is important. This aspect is not explicitly captured in Sutekh. Furthermore, Sutekh appears to only be glorified regular expressions. The diagrams for the translation into the various languages incidentally illustrate this point.

If it is assumed that all attack signatures can be represented as a regular expression, then this is a promising approach. Unfortunately, this is point is debatable to state the least. Anomaly based approaches resist representation as regular expressions with a reasonable number of terms due to the numeric expressions involved. Even a true expert system based approach can not be represented as a regular expression.

2.7 Database Approaches

Databases have been employed in network intrusion detection to record all of the packets for later study and processing. This has several advantages. First, all of the suspicious data is present for later manual analysis. Second, by relaxing the real time constraints, the detection mechanisms do not have to deal with as much partial matching state problems. Normally, if a partial match is detected, the remainder of the match must be searched against as new data appears. This is a source of significant complexity for detection engines. Thirdly, after an intrusion is detected, all traffic from the attacking machine to the monitored network can be examined to determine the sequence of the attack and other possible targets identified during the probe. Finally, after an intrusion has been detected through other means, the network traffic can be analyzed to improve the IDS configuration.

Stephen Northcutt advocates this approach. His book Network Intrusion Detection: An Analyst's Handbook[45] is based on the systems he has deployed. His approach is to use signature based sensors and to tune the sensors to the site to reduce the false positive rate. A database of all of the traffic for two to three days and a database of selected header information for two to three months are kept. By building indicies overnight, these can be quickly searched to reconstruct and analyze an attack sequence.

The size of the data to be stored is the main disadvantage of database approaches. This usually requires a dedicated machine with a small RAID array.

A second trade off of the database based approaches is that they do not run in real time. This is a limiter for using these approaches for automatic response. Northcutt claims that for human responses, this is only a marketing problem. IDSs are sold with quick reporting as a feature, but after the first few weeks it is apparent that low false positives is more important [45].

2.8 Miscellaneous Detection Methods

This section contains miscellaneous approaches to intrusion detection. It should increase the perception that people are working on most things related to intrusion detection and are willing to examine anything.

Staniford-Chen of University of California at Davis describes an approach to tracing intruders in his computer science master's thesis [58]. It attempts to produce thumb prints of interactive TCP sessions. These thumb prints both provide privacy in that the actual information in the session is not revealed and they provide a way to attempt to determine if two TCP sessions are the same. The issue of determining which TCP sessions may be the same is a way of attempting to trace back telnet chains.

At MIT, a system was built to try to identify the source code of user to root attacks [13]. The system attempts to detect source code that increases privileges by feeding statistics on the occurrence of keywords within the code and within the comments and strings of the code into a neural network. First the code was characterized as C code, shell or other. Then the code was categorized into the non-mutually exclusive categories of comments, strings, code without strings, and code. Then the occurrence of various keywords, (regular expressions), from each category was tabulated and normalized to account for the length of the file. This was the output that was fed into the neural network, at first for training and later for testing. The keywords were partially chosen one at a time due to what best characterized the training set after accounting for the current keywords.

This system was effective in characterizing the outputs when applied to the test set. At a less than or equal to 10% miss probability, the false positive probability was less than or equal to 10% for C code. For shell code the results were still good, but not nearly as good as the case for C code. Unfortunately for future work following on this approach, this detection mechanism can be easily defeated by modifying the source code for the attack to obtain desired statistics.

2.9 Application Sensors

A natural trend from network and host based intrusion detection is to increase the sources of data to be fed into the IDS. One way of doing this is to integrate the security reporting into the applications themselves. This has the advantage that for data relevant to the application, the application is easily aware of its' internal state, so can more easily judge whether or not the attack constitutes an a serious event or not.

There are several disadvantages to the application sensor approach. One prominent disadvantage is the large increase in complexity of the IDS. An IDS that took data from many application would be costly to build and troublesome to keep updated with changes in the application software. A second large disadvantage is the probability of significantly slowing down the applications due to the need to report every event. Interprocess communication is not particularly quick and if the data must be sent before further processing to ensure it reaches the IDS even if the process is compromised it would significantly slow down the application.

SRI has tried this approach as detailed in [3]. The implementation was to add security monitoring to the Apache web server and integrate the messages into the EMERALD framework.

The advantages of this approach are that more data is available, the data can be of a finer granularity and the interpretation of the data by the application is more likely to be known. For example, the internal data in encrypted sessions may be available to the security module if it lies in the application. The SRI researchers state that this is an advantage for security reasons. It may be a disadvantage for privacy reasons and may generate additional security bugs. The interpretation of the data is a major advantage, for example in monitoring a web server, the HTTP escape sequence decoding is already done. The state of the application need not be guessed as it can be directly inferred by the security module.

One disadvantage of this approach is that it could slow down the application. Syslog is notoriously slow and logging to the IDS system may not be faster. Furthermore, it would require significant development efforts to integrate it with the major applications. Their implementation of a module for Apache did not cause a significant slowdown under light loads. This was due to performing the security analysis in a different process with buffers between the two. This architecture potentially introduces a facility to crash the IDS via disobeying locking and semaphore conditions on the shared data segment. Even assuming the development of an IDS with the ability to handle potentially disruptive data in the shared data segment, ensuring that the IDS saw the security data in the event of an intrusion within the application would require the IDS to copy the data for the event before the application processes to the point of losing control to the potential security event. This requirement would force the application to cede run time to the IDS every time a message is placed in the queue for the IDS, which will significantly slow down the application. It may be the case that the designers of application based IDS are willing to ignore these interesting and particularly hard cases in the hope that things will just happen to work out most of the time.

A similar approach to the SRI work is described in [70], but they integrates all of the information into their own IDS. Implementation is "still volatile and likely to be reworked, parts of it have been running..." They have implemented an Apache module and a modified ftp server.

2.10 Autonomous Agents

Autonomous agents are a framework for a solution to a distributed task. This framework has developed as a general Artificial Intelligence (AI) technique. Fundamentally, an agent is just a program. To be precise, one AI textbook defines an agent as "something that perceives and acts"[55, p. 7]. An autonomous agent is an agent with the ability to learn [55]. The AI community represents this program by the agent abstraction so they can distinguish the architectures in which these agents fit together, often through biological analogies. The biological analogy is applied to build the solution to the problem, rather than through classical computer science driven architecture.

Often the agents have some way to communicate with each other. The architecture in which to communicate is usually specified, but the decision on which agent to communicate with is left to the agent itself. Hence an agent based approach can be very general while leaving many important implementation details unspecified. The agent based approaches to distributed IDSs often fall victim to this short coming.

The COAST laboratory at Purdue has done considerable work to advance the autonomous agent based approach to intrusion detection [8, 25]. This approach offers the prospect of eliminating many of the failure points and bottlenecks of hierarchical approaches. Their implementation of agent based approaches are still in a preliminary stage. In particular, data reduction and the usage of network bandwidth have not been examined.

The first paper from COAST describing autonomous agents for intrusion detection is [8]. This paper describes the proposed architecture, which has the agents send all of their data through a hierarchical communication network. In the next paper describing their improvements to this architecture, the agents send their interests through the hierarchical communication network, rather than sending all of their data [25]. After transmitting their interests, the communication is then conducted in a peer to peer fashion. This paper is a proposal to build an implementation, rather than a description of an attempted implementation. Some critical detail are rather fuzzy, for example, it would be useful if more detail were given in how to determine the interests of the agents.

It is not clear that either of these architectures are effective. In particular, it is not clear that the second proposed architecture addresses the hierarchical limitations of the system. In the second proposed architecture, the data is now not constrained by the hierarchy, but the interest requests may consume just as much network bandwidth. Furthermore, without constraints, it could be possible to overload one of the nodes by having it send too much data. This fix to the overload problem likely suffers from the same shortcomings as it was intended to fix. It is not clear that having autonomous agents without specifying communication mechanisms and movement determination algorithms for mobile agents really solves anything. It is a lot of terminology, but it leaves the fundamental engineering problems unaddressed.

The Intrusion Detection Agent (IDA) [5] system attempts to gain distributed scalability by delegating the detection of intrusions to autonomous agents. Scalability concerns from this approach have been dismissed in favor of the basic work of getting the approach working. Unfortunately, scalability may be both hard to analyze and difficult to achieve with the fully distributed autonomous agent approach. Furthermore, as only highly limited sets of detection criteria have been implemented, it is not clear that this approach is feasible for a more complete system.

Once taking the autonomous agent approach, the researchers concentrate on implementing a system. Their system as described in this paper is still in a very basic form, but it does seem to be an attempt to do what they claim.

As for the scalability, it seems to scale worse in some aspects than both the centralized and the fully distributed approaches. A rough guess is that it is hierarchical with respect to the output of the agents, so it will have the output of the agents as the bottleneck. There are also many potential problems with the “Tracing Agent” and “Information-gathering Agent”. The architecture is similar to the architecture of Morris’s 1985 Internet worm. With agents going everywhere, it remains to be seen how it keeps from exhausting the resources of the hosts and network.

“Mobile Agents in Intrusion Detection and Response” [28] examines the benefits and security problems of using mobile agents for intrusion detection. The analysis of a security problem introduced by mobile agents is a useful contribution of this paper. Because they have not attempted an implementation, rather than build upon this work, others will probably rework it from scratch when exploring different designs for trade-offs.

Some researches use the biological analogy rather than computer science to try to drive the research. For example, “A Distributed Intrusion Detection and Response

System based on Mobile Autonomous Agents using Social Insects Communication Paradigm" [21] is pushing the boundaries in exporting the biological analogy. The interactions and topology described follow an analogy with an ant farm controlled by pheromones. Although much of the system has been built in TK/TCL, some highly critical components have not been implemented. Nor has it been partially tested in practice. Furthermore, it is unclear how the intrusion detection modules, which have not been implemented, will be able to use the qualities of this framework to effectively perform intrusion detection better than the could if they were employed as non-connected site monitors. Although the analogy is somewhat interesting, there is nothing gained by the analogy.

The US Air Force had a project in Dayton, Ohio that used the human immune system as an analogy for an IDS system. "CDIS (Computer Defense Immune System) is a multi-agent, hierarchical, distributed computational immune system modeled after the biological archetypes of the immunological process."[71, p. 118] Typical of computational biological (genetic) attempts to handle large amounts of data, the process uses a tremendous amount of computation time to handle a task. It is trained to do attack analysis at the network layer.

2.11 Data Fusion

Data fusion is an important area of research for building large scale systems of IDSs. Practical large scale IDSs can generate thousands of alerts per day, even when not subject to denial of service attacks. Fusing alerts together can substantially reduce the number of alerts seen by the operator, which is something of commercial interest especially when operator time is specified as an expense. More common than an actual denial of service attack, rules to detect the current Microsoft worm of the month would generate immense amounts of alerts.

One of the most interesting data fusion approaches was attempted by Robert Goldman and others at Honeywell [24]. They attempt to address the problem of data fusion among multiple IDS employed as a distributed IDS. One of the key premises of their approach is that each site has different security priorities, therefore should have a different kind of data fusion. Much of the data fusion system must be configured by the site administrator. The heart of the intrusion report aggregation system is storing the components of the signatures in a hierarchical manner. A prototype has been built.

SRI implemented an effective alert correlation mechanism as part of an addition to the EMERALD system [65]. It uses Bayes inference and a database of constants for feature similarity to attempt to hierarchically correlate alerts as part of the same attack. By correlating the alerts, the operator has less overall messages to deal with and can get a better picture of the attack. The system nearly gave an order of magnitude drop in number of alerts that the operator saw during an attack. “We realize a reduction of one-half to two-thirds in alert volume in a live environment, and approach a fifty fold reduction in alert volume in a simulated attack scenario.” [65, p. 67] The alert mechanism was defined as an extension to the IETF/IDWG proposed standard. There were also constants to cut off partial matches.

Another approach to alert correlation is described in a RAID paper by Debar and Wespi [16]. The approach is a framework for hierarchical aggregation of intrusion detection alerts. The framework builds off existing commercial software for communication (CORBA) in distributed systems. It builds wrappers around the IDS sensors that are not already directly integrated into the framework. The messages in the system were developed by an OO approach that is still evident in the paper. These wrappers and the higher level nodes can switch to providing a summary of how many alerts have taken place to prevent the overloading of the higher level nodes.

The correlation that takes place in this system is rather primitive. “There are currently two kinds of correlation relationships between events: duplicates and consequences.” [16, p. 95]. The aggregation that takes places is to aggregate the security levels.

2.12 Plan Recognition

Plan recognition is attempting to infer the goals of the attacker at various stages of the attack. This is often viewed as useful to determine the potential severity of the attack outside of the technical severity due to what was compromised. For example, a hacker simply trying to gain access to as many machines as possible is a substantially less severe corporate threat than an attacker who is trying to gain access solely to the accounting databases for the purpose of financial gain. Yet from the technical point of view the hacker simply trying to gain access is a more severe problem. Plan recognition attempts to address this disjuncture by accounting for the inferred plan of the attacker in determining the severity of the attack. Furthermore, it attempts to guess the probable actions of the attacker by using the inferred plan.

Automatic plan recognition is widely considered to be desirable, but is not automated in practice. Manual heuristic plan recognition is performed by system and security administrators routinely. They examine an attack and attempt to judge what the goal of the attack was. This is done as part of the attack reconstruction. Both automatic plan recognition and manual plan recognition have shortcomings.

Automatic plan recognition is subject to two main shortcomings. The first shortcoming is that the data on the system must be interpreted. Specifically, if an attacker is examining data in a users home directory, the access to that data must somehow be interpreted within the attackers plan. This would either require the system administrator to somehow inform the automatic plan recognition software of the relevance of the data or it would require the plan recognition software to make judgments on coarse attributes of the data. The second shortcoming is that the plans to be recognized are often of a coarse granularity to deal with the first shortcoming. When a coarse granularity is used, the plans returned by the software are not as useful. Specifically, if the plan is very general, it does not assist in either attack reconstruction or action prediction. In addition, if plan recognition were used for attack prediction, the initial portion of the attack could be modified so that it would appear to have a different plan than intended.

The main shortcoming of manual plan recognition is the time it takes for a system administrator to perform it. This system administration time is highly significant. It is often the case that rather than throughly investigate an attack, the system administrator will close the security hole and reinstall the OS and software on the systems compromised. This implies that potentially useful data is lost due to the time required to recover it. Manual plan recognition may be used in the rare cases where the attacker is actually caught, so that some degree of severity can be assigned to the attack. In many cases where the attacker is caught, it is likely that regardless of the plan of the attacker, the corporation will pursue the maximum charges or fines allowed, so plan recognition does not help in this instance except where it implies that the attacker was planning to do significantly more damage than they actually did.

One advantage of manual plan recognition over automatic plan recognition is that it more easily allows human factors to be taken into consideration. For example, an attacker that is having trouble using attack scripts can clearly be manually differentiated from an attacker that forms complicated UNIX command line pipes. This allows easily inferring the degree of knowledge of the attacker, which gives insight into the severity of the attack. Furthermore, motivation is difficult to pin down even for humans, so its automatic determination would probably be useless.

Robert Goldman formerly employed at Honeywell constructed a plan recognition system that is described in “Plan Recognition in Intrusion Detection Systems”[23]. The main shortcoming with this system is that the plans recognized are of only a high level of granularity. The coarseness of plans to attempt to be detected were on the order of “get root”, “steal info”, “deface web site”, etc. This information is not particularly useful, yet developing a plan recognition system that works at a level of detail that is useful is a more difficult task than building a more comprehensive IDS.

Other authors have discussed adding plan recognition systems to IDSs, but have not detailed the workings of such a system nor have they implemented one.

2.13 Intrusion Tolerance

The goal of intrusion tolerance is to produce systems that are dependable despite intrusions. Often, the author will state that intrusions will always be present, hence the need to develop systems that incorporate IDS alerts into resource management software so that the system will be resistant to the inevitable intrusions.

It is debatable whether or not intrusions will always be present. Certainly with the current code development practices exercised by Microsoft, intrusions may occur in any OS and networked application they develop. With code developed by others, it is not as clear. One of the BSD systems has gone for multiple years without finding a buffer overflow problem. With respect to bug testing, it is ceded that it cannot be known that the tests are completely comprehensive because there may be a bug in the tests, but it may be the case that they are actual completely comprehensive. Likewise, even a proof of the security of a system is subject to human errors in interpretation and specification, so it is impossible to conclude with mathematical certainty that the proof is correct. But regardless of the limits of demonstrability to humans of the security of the system, the system itself may in fact be secure. Thus a properly designed OS or application can be without security bugs. Reality is often far from this ideal case, hence the desire to create intrusion tolerant systems.

SITAR is an attempt at "a scalable intrusion-tolerant architecture for distributed services"[68]. The architecture uses both an IDS and voting to rule out possibly compromised responses. The prototype implementation is designed around the task of serving web pages.

It seems that the SITAR architecture has a lot of shortcomings. For starters, it seems to fail to work for tasks that have state in them. Distributing tasks that have state is a known problem, but it is the more interesting problem because the stateless problem is so much easier. Furthermore, this system will probably not work if more than some fraction of the machines are compromised, which has a decent likelihood since it is likely that all of the machines will be similar types with the same software load. The implementation results that were published are disappointing. They indicated that even the preliminary prototype has not been built.

Experimental results from an approach to automatic responses to DDOS attacks is described in “Autonomic Response to Distributed Denial of Service Attacks” [60]. It uses the previously developed Cooperative Intrusion Traceback and Response Architecture (CITRA) and the Intrusion Detection and Isolation Protocol (IDIP). These works describe an implemented framework for attempting to trace back denial of service attacks. The framework works in the obvious manner, on the routers and switches, it determines which interface the attack is coming in on and sends a message to the upstream switch if the switch has implemented the system. If the switch is not part of the system, it simply firewalls or rate limits the traffic. Internal Security mechanisms of CITRA and IDIP are not discussed in this paper. Commercial software to do a similar function has been developed by other companies and is on the market.

The experimental results from this paper are not interesting in themselves. The main point of the experimental results is that the system has been built and functions to some degree. The scalability of this system is questionable. Running additional filtering applications on routers and switches could cause unacceptable network slowdowns. The communication in the protocol was not explicitly designed to be scalable, which will probably slow down their attempts to make it scalable.

IBM Zurich has developed an approach to shutting down services for security reasons [53]. The intent of the system is to automate the disabling of services like the automatic updating of software. The system has not been implemented. Some security implications of the scheme and the practical problem of naming are discussed. Considering the automatic updating of software is not widely implemented for valid reasons, it is unlikely that this scheme will come to fruition except at possibly a small number of sites. Fully automatic software updating is often not used because software updates sometimes break the software or temporarily make the software unusable. Microsoft is particularly bad in this regard. Several of the security patches released by Microsoft have broken the software that they were supposed to fix. The result has been that security administrators often do not apply the Microsoft patches unless there is a worm or virus that exploits the bug. In any case, automatic software patching has its own authentication problems due to the necessary administrative permissions and functionality of the patching software. The approach to shutting down services for security reasons can have the advantage that the remainder of the services may still be available, while on the flip side it may unnecessarily shut down services if the attack detection is too aggressive.

3. Proposed Solution

Intrusion detection research often attempts to answer one of the following questions: how to make IDSs faster?; how to produce more accurate results?; how to fuse alert reports?; and how to apply other AI methods to intrusion detection?. Distributed intrusion detection is subject to additional research problems. These problems are: how to make it scale with respect to data, computation, and network transmission?; how to organize the topology?; and how to break down the components of an alert match for matching across multiple nodes?

The overall approach of this thesis is to introduce a compiled language that attempts to simultaneously either directly address or facilitate the incorporation of solutions to many of these problems. It aims to directly address the scalability with respect to data, computation, and network transmission problems. The topology is explicitly configured to facilitate approaches to the topology problems. It aims to facilitate the incorporation of solutions to the speed issue, the accuracy problem, the data fusion problem, and ease the application of other AI methods to intrusion detection.

This chapter is divided into sections that attempt to answer the following questions: from a general perspective, what to develop? (section 1) given the general goal of what to develop, what are the large problems to address and avoid? (section 2) what are the details of the proposed solution and how does it address and avoid these large problems? (sections 3, 4, and 5) what are the properties and advantages of the proposed solution? (section 6) and what are the disadvantages of the proposed solution? (section 7).

3.1 General Goal of Solution

The replacement of assembly programming with higher level programming languages for software development has yielded an enormous improvement in productivity. Data comparing assembly with third generation languages from Brooks [11] indicates a several fold increase in productivity over assembly. Furthermore, programming languages facilitate portability of code and some orthogonality between non-asymptotic efficiency and the application code. These large gains from programming languages for application development give an analogy for a goal for easing the development of large scale IDSs: a compiler for IDSs may have the same advantages.

Currently, each type of IDS is given its own set of rules and configuration in a manner particular to that IDS and intended host. Then the alerts generated by the IDS are post processed by hand or fed into a database for manual search and examination. This has several inefficiencies: for attacks generating many alerts, all of the alerts have to be logged; even versions of IDSs change with later versions often allowing more efficient versions of the same rules; each configured IDS must disregard the traffic that it sends on its behalf to prevent loops, thus should have a host specific configuration if it is to report over the network; and unless the IETF output mode is used to ensure standard syntax with considerable overhead, each IDS has its own output format.

Application programming language compilers usually target assembly or machine code for the machine architecture. For a programming language for use with large scale IDSs, the target would be the configuration of IDSs and the IDS post processing system. Here the post processing system may be running at the same time as the IDS to process the outputs directly in soft real time. It is important to note that it is proposed to target multiple existing IDSs rather than one specific IDS or develop a custom IDS in an application programming language.

By introducing a separate language, the semantics and efficiency can be split. Considering that there has been a large division between those who are trying to make limited IDSs very fast and those who are trying to make IDSs very functional, a division between semantics and efficiency allows efficiency to be pursed without forcing the user to change their rule sets to conform to whatever is the most efficient IDS of the day. Furthermore, for most users, it may represent a gain in efficiency by allowing the software to choose an efficient way of representing the rule for the IDS in question.

This software has two useful stages. The first stage represents each rule by a symbol. Then when generating the configuration files for each type of IDS, the corresponding rule for this IDS is simply looked up in a specified table. By the introduction of this substitution, it allows the abstraction for multiple IDSs to take place. While the substitution may seem to be a trivial operation, constructing the appropriate run time systems involve some work. In any case, this stage gives most of the benefits of the compiler without the extensive work involved in constructing a full fledged multi-target compiler.

The second useful stage is a compiler from one full specification to the various IDSs. This differs from the first stage in that when a rule is not specified in a library file for a particular IDS, it can translate from a rule specification into a rule for the target IDS. This has the advantage that with many types of target IDSs, one can simply write one general rule and be assured that it will be translated to work with the various types of IDSs.

This solution effectively merges with the existing trends in IDS software. One of the most important trends is an increase in the complexity of IDS systems. A version of Snort from a year ago was about 40,000 lines of code. This years version is over 85,000 lines of code. Because of this complexity, it is easier for a contribution to be used if it does not involve re-engineering the existing IDS systems. Another important trend is the increase in utilities for alert post processing. The commercial IDSs put these in their user interface. The free IDSs supply separate tools for this use. Either way, it is recognized that alert post processing needs considerable attention.

This solution provides a basis for further extensions for secure and reliable systems. One natural extension is to integrate the output of the IDS system with resource management software. Further work toward this goal is mostly independent of the remainder of this thesis, thus will be excluded from consideration.

3.2 Problems to Address and Avoid

Some of the major problems addressed in this work are areas of concern for distributed and large scale IDS systems. These areas are failure under high load, implementation complexity for both the software engineer and the system administrator and the efficiency of post node processing software.

Failure under high load needs to be addressed to build systems that are resistant to denial of service attacks. In addition, the necessity of working reasonably at high load implies an attention to the failure mode, rather than simply attempting to make the IDS run faster.

Implementation complexity for the software engineer is a concern due to the quick revisions of the software needed, the necessity of reasonably secure security software and the need to handle an ever increasing number of attacks to detect.

Implementation complexity for the system administrator is particularly important when attempting to run systems of IDS sensors. Simply configuring these sensors by hand would be very time consuming. Furthermore, for effective analysis of attack data, the state of the system must be known at the time the attack occurred.

It is desirable, but beyond the scope of this work, to integrate the results of network IDS monitoring with data representing the configuration of the network and machines on the network. For example, a recent project by Marty Roesch, who wrote Snort, is to do just this. Automated responses to intrusions will also require integrating the system state with the IDS knowledge.

The fast pace of IDS research indicates that for a proposal to be effective, it must be fully implementable in a relatively short time frame of at most three to four years. To achieve this from a research project, attention must be paid so that most additional work can be added in an orthogonal manner.

Some of the major problems to be avoided for this work to be practical to undertake are the failure of the IETF standardization effort to standardize semantics, the alert correlation problem and the implementation of fast secure IDSs. These are important problems for IDS research. Yet solving either of these problems effectively has not been done by anyone in the field. Many attempts have been made for the alert correlation problem as it is so important for large scale IDSs. Still, no standard approach has emerged. Thus choosing an mechanism for alert correlation should be independent of the system developed to allow for both further research into alert correlation and adoption of arbitrary alert correlation methods.

The failure of the IETF standardization effort to standardize the semantics of intrusion alerts has several consequences. The first consequence is that the IDS systems are not transparently replaceable. To replace a node of a large scale IDS with another IDS, it is necessary that replacer node is configured to possibly generate the same reports as the replaced node. The second consequence is that for alert correlation, separate rules must be written for every rule and every type of IDS because the syntactically similar alerts generated by different IDSs mean different things. SRI took exactly this approach for a recent large scale IDS correlation project. They put separate rules for every type of IDS that was part of the large scale IDS [47].

Related to the failure of the IETF standardization effort to produced standardized semantics is the lack of a standard model for intrusion detection. This is a natural failure, if there is no need to produce standard semantics, there is no need to standardize the way of producing alerts. In any case, since many different fundamentally different approaches have been attempted, it is natural that there is no standard way. This implies that many models will need to either be incorporated or ignored to produce a single model for a large scale IDS system.

The implementation of a fast secure IDS requires a large amount of effort. Version 2.0.0 of Snort, which still leaves much to be desired, is about 85,000 lines of C code. Certainly, this complexity in terms of lines of code could be reduced by using an application development language with more features than C. As one measure of the overall complexity though, it indicates that the development of a fast secure IDS is an interesting problem by itself. Thus this problem will be avoided to keep the overall solution within reasonable limits such that it may be accomplished by the resources at hand.

3.3 Proposed Solution Overview

Lest it seem that the following section on model categorization and design choices seem unrelated to the practical problems to address and avoid, this section gives a general overview of the solution. A more complete description is in a later section.

The solution is a multi-layered language. This language corresponds has two portions that correspond to two different models. The lower layer corresponds to signature based modeling. The higher layer is a Turing machine with the data flow of the alerts generated by the lower layer.

Having a lower layer corresponding to signature based rules allows for the underlying IDSs to be used to generate this output. This is what allows the creation of another IDS to be avoided. To allow the usage of many IDSs for the second stage of this software, the transformation between function for different IDSs will need to be programmed. As this transformation is not always possible remaining within the confines of the IDSs languages, this transformation can be completed by adding an code to the upper Turing Machine level to "glue" together simpler alerts. Although this transformation may produce inefficient ways of matching on a target IDS, it allows the matching of rules that would normally be unable to be matched with that IDS.

On the other hand, when the transformation can be performed to the target

IDS without gluing together separate alerts, it leaves the efficiency problem up to the target IDS, which is presumed to be reasonably efficient. For language considerations, the more features allowed in the lower layer language, the more efficient the generated system provided that the glue can be avoided. Unfortunately, adding more features to the lower layer languages increases the odds that some feature cannot be matched against the target IDS. This problem is avoided in the first version of the software because the rules for the target IDS must be specified by the user or in a library. For the second version, the lower layer language must be chosen careful to prevent the user from having to use this version of the software in a way that is similar to the first version.

The language choice for the lower layer language of the translation version (second version) of this system needs an evaluation criteria to effectively judge the design trade offs between the most suitable languages. As the syntax does not need to be specified yet, there is only need to consider the underlying models.

A vital characteristic of the lower level language should be that regardless of the data seen on the network, the IDS should not hang, crash, dump core, etc. The requirement that the IDS should not hang implies that computationally the lower level model cannot be as powerful as a Turing Machine. From a practical perspective, taking 2^{80} computations is hanging, thus the theoretical perspective given in the examination of the choice for the model differs from the classical theoretical examination by the inclusion of practical limits on space and time.

The lower level language should be able to produce output for some common and easily available IDSs. If it was only capable of producing output for a custom IDS or just a few rare IDSs, this would not likely be adopted for large scale systems of IDSs.

In addition, the output from the IDS configured by the lower level language must be sufficient to allow the construction of arbitrary IDSs. This allows the language to encapsulate a generic network IDS over some target IDSs. In particular, using this framework any new IDS could be built directly on other IDSs, thus saving considerable amounts of work.

3.4 Model Choice

To make an informed choice for the model for the solution, the existing IDSs models are categorized in this section. These categories are explored to address their stability up to small changes and the practical computational consequences of remaining within those category. The criteria to choose a model for the solution is given and the core elements of the model are chosen according to that criteria.

3.4.1 Model Categories as a Stable Concept

Every IDS relies on some core matching engine to determine when to generate an alarm. Often this is a regular expression match, but a whole gamut of artificial intelligence techniques have been proposed and used including expert systems, neural networks, and various statistical techniques including Bayesian systems. For the purposes of this thesis, this matching engine will be referred to as the core of the IDS under inspection.

The models of IDS yield a different classification than the normal classification of computational objects into finite automata, pushdown automata, and Turing machine. The reasoning behind using a different classification is to draw forth two important variations which are not treated in most computational theory - access to vital data flows and effective limits on computational space and time.

Access to vital data is important for properly detecting intrusion events. Clearly, if the intrusion event happens in data that the sensor does not see, it should not be detected. Detecting it would be generating a false positive.

One example of the lack of access to vital data would be a web server attack embedded in an escaped URI. An IDS with string matching that does not include a URI unescaping pass (after TCP reconstruction in case it is scattered within several packets), should not be able to detect the attack. Here it assumed that all ways of encoding it as a URI are not given as strings to match. Because there are only finitely many ways of encoding it as an URI, the restriction on space needs to be imposed to prevent an attempt to match it by giving every possible combination to the IDS to match.

A second example of the lack of access to vital data would be trying to detect a host based user to root attack on a machine across the network. If the attack scripts or code has been encrypted for transmission, the network IDS may not be able to distinguish an attack from normal system operation. A host based IDS that monitored Solaris BSM logs could probably distinguish it, but this is another data flow which the network IDS does not have access to.

3.4.2 Effective Limits on Computational Space and Time

The goal of putting effective limits on computational space and time is to allow a classification that ignores small running time or space differences but still provides a categorization useful for practical bounds on space and time. In particular, it will be used to classify IDS models by their computational power. This is important to ensure that the lower levels of the IDS do not hang due to possible unpreventable problems. This subsection is concerned with exploring the concept of effective limits on computational space and time. The application of these concepts is in the next subsection.

A useful and exact definition for reasonable limits on computational space and time is non-trivial to obtain. There are four somewhat obvious definitions to choose from. The first definition is in terms of actual system running time. The second definition is in terms of some number of predefined basic operations. The third definition restricts operations that lead to exponential space or exponential time rather than forcing a counting of every operation. The fourth definition examines the scalability of the system over the length of the input.

The initial definition in terms of system running time has the clearest application. Its specification is the requirement that the operation return in a given amount of time. This definition suffers from the problems involved in timing any piece of software. Until the software is written and tested, it is very hard to predict the timing beyond some constant factor. Even when it is written and tested, other system processes, caching and paging issues make an exact timing difficult to obtain. Furthermore, it is highly desirable to do an analysis of a proposed piece of software without having to write the software. Even once the software is written, it can be expected to undergo variations due to life cycle changes, thus the current timing might not reflect the timing of the software to be deployed.

A second definition that avoids some of the problems of the system running time definition is to define the time requirement in terms of some fixed number of elementary operations. This allows for an analysis of the algorithms without writing the software. This is the same choice as the analysis of algorithms proceeding from general operations rather than the timing of specific instructions on a particular chip. Often the bound will be on the order of 2^{30} operations.

The operations to be assumed to be part of the system must be specified to eliminate time variances due to the emulation of a missing operation on a simpler machine. This presents a problem when trying to apply this approach to obtain a time bound for a particular machine. It is tempting to choose a set of operations that is what the particular machine can perform quickly. But if this is done, it will require the analysis of algorithms to be reworked for the machine in question up to two cases. If its operations are a superset of the operations assumed for the algorithm analysis, any algorithm that is shown to run within the bounds on the more limited machine will run within the bounds on the more powerful machine. Likewise, if a lower bound for an algorithm is obtained on a more powerful machine, then that lower bound holds for a less powerful machine. Since the intended application is concerned with upper bounds, it is better to choose a limited machine description on which to perform the analysis of algorithms.

The requirements on space and time using either of the first two definitions must be fixed in advance. This has a large shortcoming. The large disadvantage is that even a small addition to the code may push the algorithm over the fixed bounds. Thus the set of algorithms satisfying the reasonable bounds is not closed under the addition of even minor operations. In particular, it is not closed by the concatenation of two algorithms, (run one then the other). The concatenation of algorithms for DFAs is not the same as the concatenation of regular expressions.

The practical way around this shortcoming is to simply count every operation and track this count in addition to whether or not it meets the current bounds on operations and space. This is also reasonable from the perspective that the current bounds on time and space are arbitrary, hence an analysis of the algorithm should not depend on the current bounds. Naturally, these examinations are probably not exact due to the difference in effort between giving a good but approximate count and giving an exact count with the software and test cases to demonstrate that the count is exact. As the bounds are likely to be met by a large margin or exceeded by a large margin, good approximate counts are usually sufficient.

A third definition of the effective limits on space and time relaxes the counting required in the analysis. Because it is commonly a power set operation that pushes a proposed IDS operation over the limits on space and time, these operations will be restricted to sets of size less than some arbitrary amount. Power set operations can be used to skirt restrictions on inequivalence due to lack of memory. For example, the construction of a deterministic finite automata (DFA) from a non-deterministic finite automata (NFA) involves taking the power set of the possible states that the NFA can be in. A more IDS specific power set operation is on a DFA that does not have an additional specification of external memory to store a number can avoid this restriction by simply adding states for every possible value of the number. If the number is up to x bits long, it will add at least 2^x more states (times the number of states in which the number needs to be stored). This is not a realistic operation to perform in practice, hence the exclusion of this. Thirty is an appropriate upper bound on the size of a set to do a power set operation upon that is consistent with the other arbitrary amounts chosen.

Using the definition restricting the power set operations, limited operations on an algorithm meeting the bounds gives an algorithm that is still within the bounds. Even the concatenation of two algorithms meeting the specified bounds is still within the bounds. This closure of the time and space restricted algorithms with respect to concatenation brings forth a flaw in this definition. Since it is closed with respect to binary concatenation, it is closed with respect to finite concatenation (via any choice of binary grouping of the finite terms). Since there are not any limits on this finite concatenation, the original power set operations to be avoided can be performed by the following construction: for each algorithm to be concatenated, have it test for a set equivalence of elements and do the corresponding code then return. This is the power set operation, yet has been constructed from algorithms that are clearly within this definitions restrictions. One way of overcoming this restriction is to track the total length of the algorithm expression rather than operations used. This will prevent the power set construction with finite concatenation because the finite concatenation is now subject to the limitation of expression length.

Likewise, tracking all of the ways to do a power set operations involves considerably more effort than tracking individual places where the power set construction is used. Time-wise, nested loops have a running time of at least the time spent in one iteration of the inner loop times the number of times through the inner loop for each time through the outer loop times the number of times through the outer loop. If there are four loops of $2^8 = 256$ iterations each it would exceed a 2^{30} limit. To deal with this problem, it would be necessary to have an estimate on the number of operations used by the algorithm to be iterated so that the total number of iterations can still be limited. To deal with every possible case for the closure and limits of the class of time and space limited algorithms under this definition would degenerate into a chart of the ways in which the time and space bounds could be exceeded by composition of similar functions. As little would be gained from such a chart, this definitions usefulness is limited to heuristics rather than as a basis for a solid classification.

A fourth definition continues on the asymptotic definition trend. It specifies the maximum allowable asymptotic growth of the collection of computational elements with respect to the size of the input. Similar to the definition of a circuit family, the computational elements to be considered are considered as a separate computational element for every size of the input.

Allowing separate computational elements for every size of the input can enforce the need for reasonable restrictions on time and space by restricting the size of the element to a linear function of the input size. By picking a different computational element for every size of the input, any arbitrary function from $\{0,1\}^* \rightarrow \{0,1\}$ can be computed, which includes some functions that are uncomputable by Turning Machines. This is prevented in the uniform circuit family definition by the log space bounded Turing Machine circuit description program requirement. The size of the computational elements for an arbitrary function grow with the inputs at an exponential rate of 2^n, so a restriction on the size of the computational elements will exclude the choice of an arbitrary function.

Unfortunately, the restriction of the size of the computational elements to even a linear function of the input is not sufficient to exclude uncomputable functions as the following example illustrates. Consider the set of functions f_i with $i > 1$ to take the value 0 if either i is not of the form $k * 2^k$, the input is not of the form $c_0^{2^k} c_1^{2^k} ... c_k^{2^k}$ for c_j in $\{0,1\}$, the input is not a Turing machine description bit doubled k repeated times (can just check to see that the selection of every 2^k bit is a Turing machine description) or this input is a Turning machine description bit double k repeated times that either returns 0 or does not halt on the empty input. Conversely and simpler to state, this set of functions f_i returns 1 when $i = k * 2^k$ and the input is a Turning machine description bit doubled k repeated times which halts and returns 1 on the empty input. The size of the computational elements only grows linearly with the size of the input because there are at most 2^k strings of this form and $2^k < k * 2^k = i$, thus the lookup (and bit count checking) can be encoded via a finite automata with less than $c * k * 2^k$ states. This function is uncomputable by a Turing Machine because it would be an oracle for the halting problem.

The halting problem has some related consequences for Turing Machines with restrictions on space and time. Often for IDS inputs, the Turing Machine or other computational element will be run only on fixed size input. For fixed size input with exponential bounds on space, it is tempting to declare all computational elements equivalent. The strategy would be to determine the result of the computation and rather than perform the computation again, simply look up the result. The reason that this strategy does not work is that the halting problem prevents the determination of all of the results of the computations. Thus even though it is logically and mathematically determined whether or not the given input will halt, the transformation cannot be computed in any bounded amount of space and time. Hence even restricted to fixed (but sufficient) size finite input, Turing Machines and Finite Automata are not equivalent.

For the application to IDS systems, computations involving unbounded space and time are not practical. Consistent with the earlier proposed definitions, an upper bound on space and time can be imposed, beyond which the computation can be said to have effectively crashed. Using this upper bound, the results of Turing machine handling fixed size finite input can be stored in a finite automata by a computable transformation. When the Turing Machine does not return within the specified bound, this is stored rather than the result if it were to return under a longer bound because in practice the Turing Machine would exceed the bounds.

The effective limits on computational space impinge on this proposed transformation from Turing Machines fixed space to finite automata through the space needed to store the states of the finite automata. Similar to the lack of a transformation from a NFA to a DFA, under the restrictions of space, there is not a transformation from a Turing Machine down to a DFA. The same example for the transformation will work, namely the language $\{a, b\}^*a\{ab\}^k$ takes at least 2^k states on a DFA, but can be recognized in less than $k + 2$ states on a NFA or with less than $k + 2$ memory cells on a Turing Machine. By the normal embedding, Turing Machines are still computationally stronger than NFAs, which are computationally stronger than DFAs. The language $a^kb^kc^k$ still separates Turing Machines from NFAs. Here the analysis of push down automaton has been neglected due to the lack of IDS systems that use them for computation.

One way to fix the fourth definition is to include the requirement from circuit families of the specification of the description by log space bounded Turing machines. For the family of computational elements, this requirement will often not be explicating checked.

Although not computationally equal under these definitions, the standard state simulation of an n-state NFA by a DFA with n additional memory cells representing the possible states of a NFA is often a reasonable computation. This simulation slows down the running time by a factor of n, giving a total running time $O(|\text{states}| * |\text{input}|)$. The advantage gained by this slow down is the handling of regular expressions. Concatenation of regular expressions expressed as DFAs is significantly more difficult than of NFAs. One consequence of the trade off involved in allowing NFAs for time and space limited computational elements is that the number of states must be given a much smaller bound to prevent the running time from increasing by a large factor.

The inputs to a computational element are traditionally a string either seen one character at a time or preloaded onto a tape. For IDS usage, TCP streams and binary numbers stretch the limits of the traditional input. TCP streams have a beginning, but there is not a requirement for the stream to ever end. Furthermore, the IDS is usually not allowed to hold the entire stream in memory. One way of viewing the stream in the traditional framework is by a preloaded tape that is possibly infinite in one direction. Since measuring the size of the input gives a possibly infinite value, the computational size should be limited by a different criteria.

Binary numbers can be handled by the traditional representation as a string, but their representation matters for computational elements without memory that are presented the number one character at a time. For example, if the number is presented starting from the least significant digit, then the result of adding one to the number can be written on the output using a finite automata without additional storage. But if the number is presented starting from the most significant digit, this is not possible.

One way of fixing the numerical representation problem is to introduce a fixed finite amount of storage to accompany the DFA. If the DFA can read, examine, and write to the storage, then it is a Turing Machine with fixed bounded space.

Obtaining a computational element useful with numbers but weaker than a Turing machine involves additional structure. One approach is to start with a DFA and add numerical operations and storage to it. If the DFA cannot examine the storage, then it prevents the DFA from turning into a Turing Machine. A motivating example is the computation of a trained neural network. If it is assumed that the DFA has primitive numerical operations of $+$, $*$, and threshold, along with storage for the partial results then the neural network can be computed without a further examination of the contents. A more general motivation is to encapsulate numerical computations which do not depend on the inputs in a computational element less powerful than a Turing Machine.

The encapsulation of numerical computations that do not depend on the input in a computational element less powerful than a Turing machine will allow for the examination of much better bounds than are possible with a Turing Machine. A space bounded Turing Machine with x binary memory cells and an input of length n can have a running time as bad as $2^{(x+n)}$. A DFA with memory cells which it cannot examine but can perform numerical operations upon has a worst case running time $O(n)$. Likewise, simulating a NFA with the same additional specification has a running time of $O(n * |\text{states}|)$.

Unless the precision of the numbers is specified, many numerical operations will need to be given to the DFA that normally could be constructed from other more primitive numerical operations. For example, division can be constructed with the normal algorithm without control conditionals by using multiplication, subtraction, and a sign compare with zero that returns one on positive and zero on negative. However, this division algorithm still must know the number of digits in the number to produce, otherwise it will not know when to stop.

Arbitrary precision would cause considerable complexities for the use of numerical operations by finite automata. For the construction of some operations for arbitrary precision out of other operations, the time required for each place of precision may grow faster than linear. This is undesirable. Furthermore, the space required to be reserved may be proportional to the size of the input, thus violating the finite number of memory cells requirement.

Indefinite precision would cause even worse complexities for the use of numerical operations by finite automata. First, indefinite precision can be represented in finite space and time by objects like streams with thunks that give a promise of delayed computation. Like the arbitrary precision case, the running time for number of digits computed may grow at a faster than linear rate.

A difference common to both non-fixed precision cases is the dependence on the operations given. For the fixed precision case, it has been noted that if a linear time algorithm is given, then in most cases it is equivalent under the restrictions of space and time, with the proviso of the examination of the increase in number of states. For operations without a linear time algorithm, the addition of the operation produces a finite automata that is more powerful even without the restriction on equivalence of limited space and time. Naturally, it is realistic to assume that there is some algorithm for the numerical operations added. Otherwise, a solution to the halting problem could be added to the finite automata, which would clearly be unconstructible in practice.

Computability in limited space without restrictions on time or the number of states in the computational element is the domain of context sensitive languages. A Turing Machine with the restriction that moving the tape head past the original input is considered to be crashing is called a linear-bounded automaton (LBA). The nomenclature linear-bounded is applied rather than constant-bounded because the encoding of a n-tape Turing Machine onto a n-tape Turing machine allows an increase in the size of space by a constant factor to be used at the cost of a constant exponential increase in the number of states in the Turing Machine. The languages recognized by LBAs, called context sensitive, are more extensive than the context free grammars, but less extensive than the recursive languages [36].

The exponential increase in the number of states in the transformation from n-tape Turing Machines to 1-tape Turing Machines may be unacceptable if there is a limitation on the number of states in the Turing Machine. In this case, the class of languages recognized by the LBA split into a series of categories depending on the number of states allowed in the Turing Machine. There is a partial order among the categories from the partial order induced upon the 2-tuples of partially ordered bounds.

Like the limit on the transformation from n-tape Turing Machines to 1-tape Turing Machines due to space considerations, there is a limit on the transformation between k-FAs and 1-FAs. A k finite automata (k-FA) is a series of k deterministic finite automata each of whose output is a string that is fed into the next finite automata [46]. Functionally, $l \in L \iff f_k(f_{k-1}(...f_0(l))) = 1$. Without restrictions on the number of states, there is an equivalence between k-FA and a more complicated finite automata. But this equivalence involves the number of states raised to the kth power, hence will often be excluded under the restrictions on space and time. K-FA are reasonable to compute since their impact on running time is a factor of k, but they will not be treated in this thesis as they introduce additional complexity.

For fixed precision numbers, multiplication can be allowed. For arbitrary length numbers, multiplication can produce results that are computationally too powerful. The problem is that by repeated squaring of a number, a number with length exponential in the length of the input can be constructed. Then these exponential length numbers can be used for parallel processing. Even without control based comparisons, the result of the parallel processing could still be stored in the output. Thus it is desired to specify the precision of all of the numbers involved. It will also be assumed to adopt some rounding convention for operations whose results cannot be exactly specified with the chosen precision or with finite precision.

From the classical computation theory standpoint, computation of fixed precision numbers is trivially done by a finite automata with a lookup table, hence no need to include special numerical operations or memory. From the practical standpoint of limits on space and time, a lookup table of 2^{64} elements is too big. The desirability of special numerical operations over general memory due to time bounding concerns has already been noted. Another advantage of fixed precision is that some functions which are not computable by finite automata for arbitrary length numbers are now computable. For example, for fixed precision the representations of $2^x \ni x \in N$ in base 10 is recognizable for up to finite precision using at most $\lceil \log_2 10 \rceil * |\text{input}|$ states. For arbitrary precision this collection of numbers is unrecognizable by finite automata [46].

With the precision specified, operations like square root and division can be performed without additional support. Error handling can be performed by adding a special error number which when computed upon always returns the error number. This is necessary to prevent control exceptions for division by zero or the square root of a negative number while still returning valid results.

3.4.3 Categorization of IDS models

For the purpose of building a complier integrating several IDS systems, it is useful to categorize the IDS models by model features rather than the categorizations of Kumar [34] or Alessandri [2] which are classified by the attacks that the IDS can detect.

Some IDSs give Turing complete programming languages. For example, N-Code within NFR is Turing complete. If the IDSs with Turing complete programming languages have sufficient access to the network data flow, then these clearly belong in their own category as network IDSs capable of recognizing any attack which any network IDS can properly recognize.

If the access to data in NFR were slightly more restricted, it would belong in a different category. Within NFR, to access part of the TCP header, one uses the variable for the field to access. Unfortunately, many of the fields are not defined as variables, so to access an unnamed field, one has to grab the payload of the level below and parse the fields themselves using byte and bit operations. If NFR did not support examining the lower levels or did not support byte and bit operations it would not be able to properly recognize any attack which some network only IDS can properly recognize.

Many interesting signature based IDSs are not even computational finite automata. For example, Snort cannot even handle regular expressions. Neural network approaches tend to have a training phase, then when running, are a function whose computability is fixed and does not depend on the input. Hence when these approaches are running, they are not even able to successfully distinguish languages which finite automata can distinguish.

The languages recognized by the current payload examination features of Snort are more difficult than most regular languages to describe as a regular expression due to the byte_jump option. Prior to the introduction of the byte_jump option for Snort, the languages that the payload examination features of Snort could decide were of the form $\alpha^{\{a_0,b_0\}}c_0\alpha^{\{a_1,b_1\}}c_1...\alpha^{\{a_n,b_n\}}c_n$ where α matches on any single character, c_i matches on one specified character or optionally one case insensitive character, n is fixed and a_i and b_i are optional but fixed. Although Snort has a regexp field, the most recent documentation indicates that it is still under development and any actual usage of it is considered to be an error. These languages do not include all of the regular languages. For example, $(01)^*$ is not of this form. Hence Snort is not as computationally powerful as a programmable finite automata.

With the introduction of the byte_jump option, the languages that the payload examination features of Snort can recognize are more complex to describe. The byte_jump option allows a numerical test of $<$, $>$ or $=$ to be performed on a conversion of the input as little or big endian binary of a given length in bytes or as a string conversion from a specified base. Then once the numerical conversion is performed, the string matching can resume either at a given non-negative offset from the current match, which is equivalent to α^n or at a non-negative offset specified from the number converted. To view the offset specified from the number converted and tested as a regular expression, it would be necessary to give every possible instance which it could encounter. This is possible because the length of the conversion is fixed to be less than or equal to four bytes. Naturally, this makes the regular expression unwieldy and unnatural. For human viewing, a variable with a specified range could be introduced, but for machine viewing, all $c*2^{32}$ possibilities would have to be written out. Without the imposition of reasonable restrictions on space, this would be computationally weaker than programmable finite automata. But with the imposition of reasonable restrictions on space, it is neither strictly weaker nor strictly more powerful than a finite automata. In practice, all of the Snort rules considered by the Snort website to be stable only use small fixed offsets to resume matching from.

The remaining non-content fields in a Snort rule form a logical ‘and’ for the consideration of when a rule matches. The matching of the non-content Snort fields can be accomplished via bit bashing with extracting the proper information from the relevant predefined structures. For example, the TCP flags field specifies a bit-mask, a matching operation of exact, any or all, and the flags. The flags are a human readable way of specifying a bit offset in a TCP header that is significant for the TCP protocol in that it is normally referred to as the TCP flags. The matching operation can be transformed into binary or, and, or equal thus reducing the match for the field into some specified bit bashing and checking against zero.

Expert system based systems can be viewed as a special case of search based systems. Although search is a powerful operation from a computational point of view, the search question must be properly phrased. In particular, lacking operations to transform the data, the computational power of search is limited to the questions that can be phrased by the untransformed data. Depending on the operations available, this may imply that except for the questions that can be phrased by the untransformed data, the computational core of the IDS can still be quite weak computationally.

The order of search is important to modeling search based systems. Due to the limits on computational time, even if there is an answer, not every search based system may be able to find it. Hence the order of search will dictate that some systems may be able to find the answer, but others will not be able to find it even in some time that is $c * \alpha^n$ times the time of a system that found it for reasonably small α. Related to the search order, the specification of rules for search based system is often order dependent. For example, in PROLOG, the rules are added in the order specified in the file, so the base case for recursive search must be specified first to prevent infinite loops. Any language that attempts to model a search based system should deal with this order dependence. Naturally, the direction of search is important.

One way of simplifying this is to assume that the search order and direction are specified in the rules. Then there is only one general search, hence only one model is needed. From an implementation point of view, this solution may need additional aspects to form a complete solution. Production systems often have ways of turning on and off rules, which will need to be incorporated into the general model. In any case, the computational categorization of search based IDSs will still depend on what questions can be phrased to the search engine.

After training, the self organizing map (SOM) for INBOUNDS suffers from a data flow limitation that does not allow it to recognize all of the languages contained within the packet payloads that can be recognized by DFAs. Considering that the inputs to the SOM are six numbers, it is not surprising that it cannot distinguish on payload properties. Since the data flows within the packets are not examined, it cannot differentiate attacks that differ only by within packet data.

Given the input of six numbers, a finite automata without numerical operations would run into the reasonable limits on space restriction. Simply testing the Euclidean distance for six 32 bit numbers would push the number of states required to over 2^{32}. A finite automata with numerical operations can compute the function needed to detect a match for the self organizing map. Even viewed from the input of six numbers, the SOM is not as computationally powerful as a finite automata with numerical operations. Specifically, a finite automata with numerical operations can match the language where all six integers are of even parity or where all six floating point numbers have even parity for their coefficients of 2^{-x} for $1 <= x <= 40$ with at most 1000 states. To do this with the SOM would require more than 2^{32} states (a state is the center of a trained point) which exceeds the reasonable bounds. Therefore, the SOM is computationally weaker than a finite automata with numerical operations.

The input module for the SOM suffers from the same numerical operations problem. If a limited set numerical operations are allowed, then it can be modeled with a finite state machine. It certainly is not as strong as a normal finite state machine because of not examining the packet payloads. Among the numerical operations needed for the input module, addition and subtraction can be implemented without extra support on a finite automata. However, division cannot be implemented without extra support on a finite automata, hence the requirement of extra support for this to be put into an order with the normal categories. It is not asserted that the input module alone suffices for an IDS model. Clearly both the trained SOM and the input module are needed to fully model the SOM module to INBOUNDS.

Neural networks have been used as the central model for IDSs. Trained fixed depth and size neural networks are also subject to the number handling problem with respect to the operations of addition, multiplication, and threshold. Allowing these operations, the size of the expression represented by the neural network is subject to blow up by the exponent of the depth of the network. Rephrasing this, neural networks can be computed by a fixed expression that involves $(\text{width})^{\text{depth}}$ terms. Thus for reasonably small width and depth, neural networks are easy to compute. When the size of this expression exceeds the reasonable size requirement, the neural network can be computed by a finite automata with $2^{\text{width}} * \text{size}$ states. After that bound exceeds the reasonable size requirement, a neural network should be computed with a Turing machine. Considering that there is a transformation from circuit simulation (acyclic circuits as opposed cyclic circuits) to specified neural networks and that acyclic circuit simulation is P-complete, it is not surprising that the computation of reasonably sized neural networks requires using a Turing machine.

3.4.4 Desirable Model Properties

Models of attacks have been developed that depend on the attack itself, rather than the detection of the attack. These methods of modeling produce attack taxonomies that are easy for humans to understand. They usually provide a categorization of the attacks that include a classification of the severity of the attack and the type of bug that allowed such an attack to occur.

In most cases examined, the criteria produced by classifying the type of attack rather than the type of attack detection, is not of sufficient depth to fully characterize the attack for automatic detection. Often the models lack sufficient depth to allow for automatic detection, even in cases where sufficient depth is given, the translation from an attack model to a detection rule via a generic method is an open problem. Therefore, the strategy of basing the model on the attack itself is not pursued here.

Expert systems and statistical anomaly based IDSs find intrusions in fundamentally different ways. This is due to having fundamentally different matching engines. Hence, it is unreasonable or at least beyond the scope of this document to produce one model for a detection in both IDSs.

For feasibility and software engineering reasons, it would be desirable to develop one model of a single attack. To do this, it is necessary to limit the number of IDS matching engines that are examined. So which methods of matching should be chosen?

3.4.5 Choice of Core Model for Solution

The core elements of the model are the key design choice for the language to be built. The choice here determines what can and cannot be simulated in this language and also determines if effective bounds can be placed on the running time. For the language to be developed, rather than syntax without semantics, something must be chosen.

Signature based and statistical anomaly matching stand out for encompassing most existing systems and having many desirable qualities. Signature based matching is finite automata matching. Statistical anomaly matching can be viewed as based on finite automata with numerical operations.

Signature matching allows for rules with very low false positive rates. As Axelson [6] has pointed out, when scaling up to large scale traffic from the data rates in the MIT DARPA evaluation, the false positive rate with respect to traffic examined must be very small to allow for a reasonable probability that the alarm is indeed an attack. Signature based matching with appropriately tailored signatures can satisfy extremely low false positive rates while still providing considerable amounts of coverage.

Signature matching is also used in many low level IDS systems. Much of the research on fast IDSs has used signature based systems. Many of the deployed IDSs are likely to be signature based system, of which Snort stands out as the most likely IDS to be deployed currently. Northcutt mentions several signature based IDS systems as the common IDSs which were deployed in his experience [45].

Statistical anomaly matching is a promising area of IDS research. Systems have been built that find many intrusions that would be hard to characterize in signature-based systems while keeping an acceptable false positive rate on reasonable data sizes. As noted in the theoretical section, finite automata with numerical operations can still have linear upper bounds placed upon their running time.

Finally, as many of the high level IDS components are unspecified, the final layer of the system is chosen to be a Turing Machine. It is asserted that at the highest layer, the application developer should not be limited by computational restrictions. Given an interface to other programming languages, computational restrictions within the language would serve little purpose as they could easily be skirted.

3.5 Proposed Solution Details

By effectively combining signature matching with statistical anomaly matching, many of the engineering challenges can be addressed. Two ideas make this combination particularly useful.

3.5.1 General Statistical Techniques

The first idea is to consider statistical anomaly matching from the computational point of view. From this point of view, a statistical anomaly matching automata can be computed with a numerical finite automata with reasonable limits on space and time. It is not asserted that every statistical function can be computed this way, rather it is asserted that most statistical functions used for IDS anomaly detection can be computed in this fashion.

Taking statistical anomaly computations from the computational point of view has the advantage that computations not normally considered to be anomaly based IDS can be performed with this layer. For example, counters can be implemented

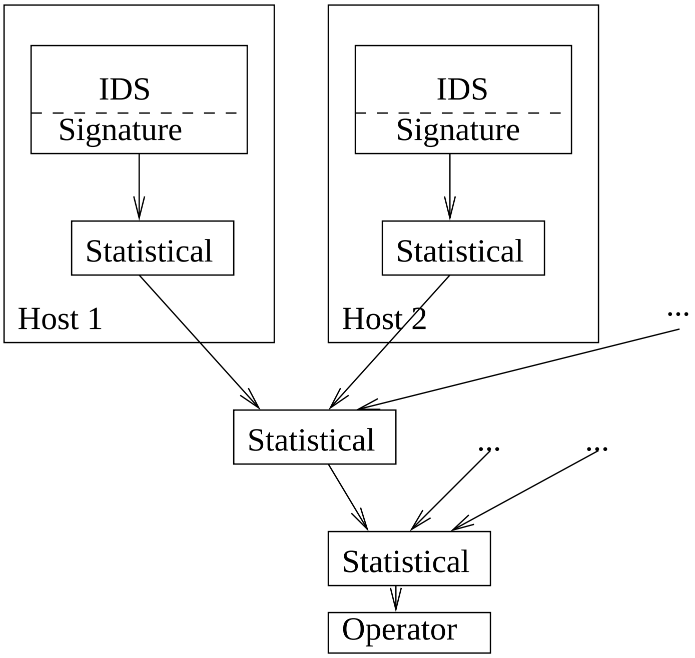

Figure 3.1: System Run Time View

under this computational framework and applied in a fashion that is not anomaly detection. This is useful to track the number of times an alert has been triggered.

3.5.2 Feeding Signature Rules into the Statistical Framework

The second idea is to force the signature framework to feed into the statistical framework. Because of the importance of signature based intrusion detection, a way of ensuring that it is possible to extract all of the alerts from the signature framework if necessary is desirable. Doing so is easy. The identity statistical rule - an occurrence of one is an event - is sufficient to ensure that the statistical framework can be configured to pass through individual signature alerts without loosing data.

As noted in the last subsection, the statistical framework is computationally powerful enough to implement counters. Counters are important for preventing a denial of service (DOS) attack from propagating past the nodes that are under attack through generating a massive number of alerts.

3.5.3 Sufficiency of Framework

Signature based detection with reporting of the actual packet feeding into a Turing Machine is sufficient to detect any attack that can be detected by an other network IDS. This is because of the ability of the Turing Machine layer to examine every packet if the signature based IDS is set to alarm on everything and the statistical layer is set to pass through every alarm. This shows that this system is sufficient to detect every type of attack that any other network IDS can detect. This sufficiency implies that other IDSs can be built upon this framework rather than a custom application reading raw network data in an application programming language.

Detection by examination of everything in the Turing Machine layer does not effectively use the IDS layer to filter or process data. If the IDS layer were used to filter out most of the data or were used directly for processing, then this would show that this is sufficient in practice for most cases. Moreover, if the operations of interest to the Turing Machine layer can be filtered for at the IDS layer, which is often possible, then this framework is almost as efficient as using the Turing Machine layer directly upon the raw network data. Unless the implementor of another Turing Machine IDS does as good as or better job of filtering low level data, this framework will be as efficient in cases where the ratio of alerts to packets is small, which is usually the case.

3.5.4 Run Time Support

This general framework should be supplemented with an appropriate run time architecture. The basic requirements of the framework are to handle booting, tear down, interaction with the user, layer input parsing, and the running of the high level Turing Machine code. More complicated, but useful features could be dynamic reconfiguration of nodes and the system, multiuser display, and interaction and integration with resource management software.

Booting and tear down of the system are required for basic usability. Although not particularly complex to implement for a tree based hierarchy, they save a tremendous amount of time for the system administrator. A proper boot of the system may involve writing a configuration file for every IDS system that includes a pass rule for the traffic that constitutes part of this IDS system. This is sometimes necessary to prevent infinite loops due to reporting on the traffic that the IDS is generating. Booting by hand would be particularly frustrating as there is a partial order in which the IDSs must be brought up. In any case, the sockets between each process must be established. For the tree based case with subnodes corresponding to subprocesses, this is straightforward. Each node in the tree creates processes to handle the subnodes. Of subnodes sometimes correspond to superprocesses, then having the subprocess sometimes start the superprocess is a convenient way of booting the system. If the system configuration is not a tree, the problem is significantly more complex. Specifically synchronization and connection to existing nodes must occur to properly configure the system.

Tear down of the system implies a requirement to track the existing state of the configured system. Then any one of the various tear down schemes can be used. The implementation uses the scheme where each node tells the subnodes that it created to tear down, then tears down itself.

Input parsing is needed to handle the output of the various components. This is part of the run time code, but does not need to change dynamically.

Interaction with the user for control of the system is a software project by itself. The ease of user interaction requirements have been ignored in favor of a primitive interface that allows the user to perform arbitrary functions on the run time system. For this project to be used by less technical users, an easy to use descriptive interface could be developed over the technical interface. This has not been done because it is not central to showing the efficacy of the system.

Multiuser interaction creates two problems: first, it creates the problem of duplicating the alert data and second, it requires synchronization of the commands issued to the IDS system. The duplication problem involves trade offs to ensure that a node does not overflow with data. To do this effectively, data must be dropped in some matter if one of the higher level nodes cannot keep up. The synchronization of concurrent commands to the IDS system has various solutions. The easiest solution to implement is to leave it up to the user to try to avoid concurrent commands or live with the consequences. The next easiest solution is to group all of the commands together and order them before issuing them. This requires designating a central control machine to solve the concurrency problem. Solving this problem without a central control machine is more difficult and is not addressed here because it more properly belongs to the development of distributed resource management software.

The integration of the IDS system with resource management software is an approach that can yield an increase in security for the resource managed system. If the IDS system allows dynamic reconfiguration, it may be possible to control the coverage of the IDS reporting and the resources it takes by automatic software. This is not examined here because of complications involved with automatic control. Specifically, with automatic control it is possible to DOS the system via false alarms.

Dynamic node and system reconfiguration are desirable for several reasons. The first reason is that the system should not have to be restarted every time a new rule is to be added. Restarting the system usually involves a brief time gap in coverage. The second reason is that a large scale system will often have dynamic events, like network, software, and power failure that should be addressed without a reboot of the system. The third reason is for the integration with automatic resource reallocation software. The fourth reason is to allow for upgrades of the software at individual nodes without rebooting the entire system. Finally, the fifth reason is that dynamic node reconfiguration is desirable to increase and decrease the level of examination of a node when either it is under a DOS attack or there is suspicion of some type of attack ongoing but few alarms so far.

The dynamic features require the language in which these components are controlled to have some features. Specifically, it must allow for either interpreted or dynamically linked compiled code to be used if parts are to be reconfigured at run time. Although this can be done with the standard system programming language C, it is much easier to do it with a higher level language with garbage collection and an interpreter. Furthermore, because of the current development state of this work, it is appropriate to use a high level language that speeds system development time. Scheme was chosen as the Turing Machine and implementation language for these reasons.

3.5.5 Other Language Features

The standard object oriented techniques of classes and inheritance are a substantial tool in coping with the complexity of the configuration. Thus it is proposed that the language include these mechanisms.

Another feature of the language that should be addressed is the handling of time. For distributed systems, this is usually problematic. Even using a synchronization mechanism like network time protocol (NTP), the system clocks are only synchronized up to some bound. This implies that within the system, the timestamps do not form equivalence classes. Specifically, it is not transitive: if events A and B happened within some time difference bound and events B and C happened within that same time difference bound, it does not imply that events A and C happened within that same time difference bound. This creates problems when attempting to correlate separate events into one alarm. Rather than address this problem, the current implementation of the language gives a representation of time, but does not address synchronization problems.

One feature that is particularly useful for this system is the ability to base a configuration off an existing configuration for an IDS. Specifically, functions were implemented to take an existing Snort configuration and turn it into a configuration of a single node for this system. This eases the transition from the hand configuration of IDSs to the use of this system.

Resource limited structures were introduced to increase the functionality of the statistical framework. These structures can generate alerts to the next level of the system and continue to function in a transparent but more limited manner when their finite bounds are exceeded. Moreover, by a proper encapsulation with limitations on examination, their usage by the numerical finite automata does not extend the bounds on running time.

The rules for the non-signature based nodes of the system were stored in databases. It was originally thought that this would increase the orthogonality of the system, but in implementation it did not help a lot.

3.6 Properties of This System

Translating from the general framework to the IDS and run time system configuration yields a system with many desirable properties. Some of the desirable properties in which the system can handle particularly easily are reasonable failure under high load, systematic nomenclature, distributed configuration, and options for dynamic reconfiguration.

Acceptable failure under high load is handled particularly easily in this framework. Each signature-based rule can have a corresponding rule in the first layer of statistical detectors that counts the number of times that the rule has been activated and reports the total at a fraction of the rate that the rule is triggered. This would limit the data flow and post node processing time to a fraction of the original time. It would not impact the data collection, only the latency of the reporting.

Under high load or a DOS attack, the nodes of the system would be DOS due to load of the IDS before reporting are still DOS. Without dynamic reconfiguration, this does not increase the speed at which individual IDSs process the data. What it does solve is preventing nodes with high loads from overloading the nodes to which they report. By counting exponential or other back off of the number of alerts can be obtained. This prevents the IDS from significantly increasing the load on the network or forcing the node to which they report to become overloaded as well. Moreover, it does so without losing important data.

A second method of grouping alerts for data reduction under high load would be to put in the security priority of each rule and report the overall security state as the most insecure of the insecurity violations detected. This could be done at each distributed node before transmission over the network, thus saving considerable amounts of network resources.

Another consequence of this architecture is how it supports rule correlation via a systematic nomenclature while simultaneously easing the configuration problem. These are supported by compiling the configuration, complete with all signature and statistical based rules, to a configuration of the running system.

This compilation step solves the systematic nomenclature problem by pushing the nomenclature efforts to library files. It is hoped that through the efforts needed to effectively correlate rules and the common components used by many rules, common interfaces will emerge that can eventually be standardized. In any case, without at least one relatively complete implementation, it is too early to attempt to standardize.

By layering the system on top of existing signature-based systems, significant amounts of implementation effort can be saved. Snort is quickly becoming the defacto standard for network signature-based IDS. By using it to feed into statistical based detection, it can be used in many different anomaly based IDSs. The complexity of developing a signature-based system as fast and configurable as Snort would be a highly substantial addition to the overall system complexity that should be avoided.

Furthermore, by using the existing systems to do the bulk of the computational work, this system is efficient. Since most of the processing time is spent on the lower levels, the use of existing IDSs forces much of the efficiency problem onto these IDSs.

By allowing the integration of existing IDS system configurations into this system, it allows existing IDS configurations to be used in a backward compatible manner. This is useful to ease a transition to this system. Because existing IDSs are at the lower level of this system, it ensures that work done to improve the lower level IDSs will be taken advantage of in this system.

The automation of configuration, booting, and tear down is a considerable gain for system administrators. Considerable work is saved for them by using this system instead of doing these things by hand. Also, if all alert post processing is done by this system, the underlying data need not be stored, hence many privacy concerns can be avoided by restricting the operations allowed for printing and examination of the data.

Finally, this system is implementable within reasonable time frame. It can be built before it becomes obsolete. The first useful stage of this software gives many of the advantages while avoiding much of the implementation complexity.

3.7 Disadvantages of This System

Some of the advantages of this system have corresponding disadvantages. Using a language for translation and system configuration introduces yet another language. Added a layered system on top of existing IDSs increases the overall complexity of the IDS system. Using rule translations increases the ability of the user to specify inefficient rules for the underlying IDSs.

The introduction of another language necessary for use by system administrators may be a disadvantage for two reasons. First, the proposed system will not have a language that is one of the standard system and system administration languages, C/C++, shell or Perl. Hence to make significant changes to the system, the system administrator must learn the language offered by the system. Second, the alert correlation routines must either be written in this language or interfaced in this language. For highly complex alert correlation, interfacing to another set of routines is probably desirable to allow the alert correlation to be written in whatever is most convenient. Unfortunately, the standard language for IDSs seems to be C or C++, so the programmer will need to learn how to interface with this particular language.

The disadvantage is not that the interface is difficult, but that it is not exactly what the programmer will likely be used to.

The addition of this framework over existing IDSs adds another layer of complexity. This may be significant for small systems of IDSs whose overall configuration and post processing complexity is low. It also introduces another point for bugs to be introduced by mistranslation or errors in the run time post processing framework.

Using compilation for IDS may allow using inefficient options. This is a common disadvantage to giving users a more powerful language. It is easier to force the lower levels to do time consuming work for little gain. In particular, when translating IDS rules, the translated rule may be inefficient to run on the target IDS. It will still run, but may slow down the IDS by an unacceptable amount.

4. Implementation Results

This chapter describes the implemented system and individual implementation tests to demonstrate the claimed advantages. It describes major design choices and states where a further determination of a proper design choice would involving solving a substantial problem in itself. The major demonstrations are of graceful failure under denial of service attacks, construction of a substantially different anomaly IDS using primarily the lower level language, and translation between two different signature based IDS systems.

This chapter is organized as follows: section 1 discusses the general structure of the implemented system; section 2 discusses booting and the configuration of the system; section 3 discusses the treatment of time; section 4 discusses methods for handling DOS attacks; section 5 discusses additional useful data structures for the low level language; section 6 discusses the construction of an anomaly based IDS using this system; section 7 discusses the translation between two signature based IDS systems.

4.1 General Structure

This section describes the general structure of the system. It discusses the implementation language choice, the communication between nodes, and the general internal structure of a node.

The system has been developed using the Scheme programming language. This allowed for rapid development due to high order programming language features and garbage collection. These often allowed one general procedure that took subprocedures as arguments to be written instead of many separate procedures. Furthermore, use of primitives like `map` sped development. It is commonly the case for the code structure that an encapsulation gives both functions and data. Here it is convenient that the functions and data are in the same name-space, unlike in common LISP. Although data and function encapsulation can be performed in C++ or other object oriented languages, it requires significantly more code to do so in C++.

Garbage collection was a highly useful feature. Without it, almost twice as much development time would have been needed to handle every instance of memory allocation, deallocation, and error checking. With C, this trend is particularly pronounced. With C++, the complexity of memory self management is still present, but often hidden by the encapsulation. In any case, a lack of garbage collection would have hindered using intermediate results for computational convenience. As bounds have been placed on the memory requirements for low level computations and high level computations may need an arbitrary amount of memory, even for production code the advantages of proper memory self management are small and far outweighed by the disadvantages. As the goal of the implementation was to produce a functioning prototype, there was nothing to be gained by not using garbage collection and a substantial amount to be gained by using it.

Other reasons for choosing Scheme were access to a good compiler, same language as another IDS system under development at OU, and extensive knowledge of Scheme and the compiler by the adviser of this thesis, Carl Bruggeman.

The Chez Scheme compiler was used for the development. It quickly produced code that qualitatively ran reasonably fast. This implied that little time was lost due to using Scheme rather than a more traditional language like C. Thus the executable produced probably ran within a small constant factor of the time an executable written in a language like C would have produced. Furthermore, as it is a prototype, the increases in speed should be due to algorithmic changes, not small optimizations. In any case, the fact that the speed penalty was not a factor was helpful. Most other implementations of Scheme are interpreters, which tend to be an order of magnitude or more slower.

Chez Scheme has also been used for the development of a Scheme based IDS called Network Monitoring Language (NML) by Carl Bruggeman. At points during the implementation, it has been proposed to add configuration for this system to the outputs of the translator. By choosing Scheme, both applications are written in the same programming language, making the integration particularly easy.

A final reason for choosing Scheme was the extensive knowledge of the compiler and programming language by the adviser of this thesis. Although it was not relied upon, it is always helpful to have expertise close by to answer questions in case of difficulties with the language or compiler.

Chez Scheme has several language additions to Scheme that are useful for this project. One feature particular to Chez Scheme is the foreign procedure interface. This interface makes it easy to use C library calls and C procedures to interface with the system calls. This simplified the development of process manipulation (`kill`) and time based procedures (`usleep`). Furthermore, it already has a subprocess interface which was used extensively to spawn subprocesses for the generation of components. Since the Scheme programming standard is rather lacking in Input/Output (I/O) and file system operations, Chez Scheme has primitives for these [20, 19].

The general structure of the configured system is based on message passing. Each leaf node is an IDS that can issue alerts that are printed on standard output and are line terminated. The non-leaf nodes have a collection of pipes, each of which have their own parser and interpretation of the returned data. Specifically, if the subnode is a leaf, it will be configured to have a parser for that systems IDSs loaded for processing that message pipe. If the message pipe is standard input, the parser is a line terminated Scheme read and the interpreter is Scheme '`eval`', resulting in a message pipe that simply evaluates the code passed to it. If the subnode is not a leaf node, then the parser is line based and examines the first whitespace separated symbol on the line to determine the syntax and semantics of the remainder of the line. This allows each type of internal message to use its own syntax and semantics if necessary. The output to the user, which is the output on standard output of the root node, is formatted for human readability.

One advantage of using standard input and standard output for the node communication is that is particularly easy for any function to generate an alarm. This can be used for reporting by resource limited structures when their finite resources limits have been reached. Another advantage is that it is easier to implement than adding another socket. The Chez Scheme '`process`' function generates the socket pair and after the fork, reopens standard input and standard output to be one end of the socket pair. The output module used with Snort only needed a minor modification to work with standard output rather than a file.

Furthermore, by using standard input and standard output, the remote process control can be accomplished by sending the commands and receiving the output through ssh, which is particularly easy for remote procedure calls and network transmission. The alternatives are substantially more involved and should invoke some secure method of network transmission like the secure socket layer (SSL). These also produce synchronization issues. Specifically, one machine must begin listing on a port then tell the other machine the port, which involves either threads or even worse, multiple processes.

Ssh has other advantages for the network connection. It solves the authentication problem for remote process invocation. It is a reasonably well tested piece of software, hence is unlikely to introduce many more security holes. It has been ported to many types of hardware and software, so its use is not specific to the OS in use.

The core loop of the interior nodes uses polling I/O over all of the connected subprocesses, the superprocess or user, and the run queue to determine if there is a new message to process. Blocking I/O with threads would be more efficient, but more difficult to implement. To implement it with blocking I/O, several approaches could be taken. The easiest approach is to use semaphores to count the messages and mutexs to give all of the messages to process to one thread. This approach limits the data that is accessed concurrently to the data necessary to read messages, manage memory, and place the messages in a queue for use by a thread to process them sequentially. Other approaches could do more processing in each thread, but at significantly increased complexity as the processing would happen concurrently. The approach taken is to avoid the concurrency issue by using polling I/O. This lightly loads the CPU when no data is being generated, but this load should be relatively small due to the waiting using '`usleep`' (micro-second sleep). For the demonstration of the effectiveness of this system, polling I/O is sufficient.

Message passing was also used internally as an encapsulation method. Rather than define a data structure and have separate accessor procedures, the definition of a data structure encapsulated the accessor procedures. For example, to get the head of a resource limited list `l`, one uses `(l 'head)` rather than `(limited-list-get-head l)`. This encapsulation method is highly effective but has a drawback when used for data structures which are transmitted across processes. For the transmission across processes, it is necessary to print the state of all of the variables that are encapsulated. For restricted definitions involving only one level of scope, a special form could be used which would bind `var-names` to a list of the local variables of the expression. This was not done due to the effort involved versus simply writing out the few instances in the constructed code.

The rules for the interior nodes were stored in a database. The rule could be entered into the database as applying either to a list of alerts or applying as a default to all of the alerts. It turned out that this rule database was not as helpful as originally envisioned. Combining the rules for one alert together to form a meaningful action to do is the weak point of this approach. The solution taken in the implementation was to simply do the first rule returned by the database. Since the database returns the local rules first, this in effect produced a substitution of the local rules for the generic rules whenever a local rule had been added.

An approach where every rule was performed for the alert was attempted. This did not produce desirable results. Specifically, some rules were designed to group together alerts. When these were intended to be used, the generic rules were used also which defeated some of the purpose of the special rules. Whereas the system that was constructed had only a few rules, the loss of orthogonality was not severe. There are several alternatives for treatment of this problem, but as sufficient numbers of meta-level alerts will need to be written to effectively test the proposed solutions, this problem is left for further research.

For ease of prototype implementation, the lower level computational framework was left to be an exposed version of Scheme. To implement it properly, an interpreted or translated version of Scheme restricted to the numerical and finite automata operations and declarations should be developed. Then when rules are declared for this level of the system, the language is restricted such that the computational claims hold. This was not done because the only user of the prototype system was the author of this thesis. Hence for prototyping, it was simpler to leave the entire Turing Machine language exposed and manually check that the code would be acceptable to the restricted language. For a production version, the language should be restricted so that the computational claims are enforced.

4.2 Booting and System Configuration

The trade-offs of allowing non-tree based configurations have been discussed in chapter 3. This section discusses the run time system control, configuration language, and the configuration itself.

As noted in the general system overview, each interior node has a message pipe from the parent. This message pipe calls the Scheme parser on lines, then uses the Scheme '`eval`' on the result. This is equivalent to interpreting the message as a

Scheme program. This mechanism allows the parent to tell the child to perform arbitrary actions. For example, to tear down the system, the user tells the root Scheme process “`(*clean-up*)`”, this function is already loaded by the nodes. It tells each non-terminal child node to “`(*clean-up*)`”, then it performs local exiting code, then flushes and closes the open sockets and exits. If the child node is an IDS, it tells the IDS to exit, often via `SIGTERM`. Since each node has its hostname bound to a specific variable, it is possible for the user to give commands to a node or collection of nodes via “`(broadcast '(if (equal?  hostname "fire") (do-function)))`”.

This mechanism almost, but not quite, allows dynamic reconfiguration. The problem with dynamic reconfiguration is that the Snort terminal nodes do not support dynamic reconfiguration. Although Snort will respond to `SIGHUP` by rereading its configuration file, the code actually just closes everything and re-execs Snort. Thus the data that is currently accepted but not processed will be lost. Even though this is a reconfiguration, since it is equivalent to tearing everything down and restarting, it does not suffice for effective dynamic reconfiguration. If the Scheme control message pipe mechanism were to be used for reconfiguration of the network transmissions paths for alerts, a mechanism for accepting connections to an existing node would need to be added. It is not difficult to add such a mechanism, it simply falls outside of the goals of this thesis.

Each leaf node needs a configuration file for the IDS running at that leaf. This system supports different configurations for each leaf node. This may be desirable if a set of rules significantly slows down the detection or if a set of rules will never match due to the placement of the IDS behind a firewall. In these cases, it may be desirable to have multiple IDS configurations for one type of IDS. Another case is when the rules have been compiled from another IDS or are the inputs to a processing module for some other IDS. In these cases also, the generation of these alerts is not desirable on every leaf IDS, thus the desire to implement different configuration files for different nodes. Without modification, Snort cannot read its configuration file from standard input, so files must be generated. The interior nodes of the system read their configuration from standard input by the same mechanism used to tear down the system.

Another reason to write a configuration file for every host node IDS is to deal with the possibilities of source feedback loops. If the IDS is set to generate an alert on every packet or packet within a suspicious connection, the IDS can get stuck in a loop reporting alerts for every time it sees its own output. This gives an inadvertent self DOS. One way to fix this is to add a pass rule in the node IDS configuration file specific to the connection which the alerts are transmitted out on. If more than the one connection is added, it may exclude more data from monitoring than is desirable, hence doing this correctly involves writing the configuration after the connection has been established. This has been handled in the constructed system by ignoring it.

The configuration language had two components. The first component specified the types of nodes that were to be configured. The second component specified how the nodes were to be arranged in a tree.

The elements of the first component took several possible forms. The first form was a specific IDS system and a function to generate the configuration file for the IDS system. In effect, this is a function to generate a running target IDS. As implemented, the function took a list of rules and had a mechanism to read existing rules from a configuration file for the target IDS. This allowed the user of this system to add rules not present in the configuration file or to remove undesirable. The reading of the target IDS configuration file facilitates backward compatibility with existing target IDS installations.

The parsing of the target IDS configuration file adds some complexity to the configuration code. The file is in the configuration language specific to the target IDS. For Snort, this is line terminated file with comments and the keywords are the first word on the line. During parsing file inclusion via the “`include`” keyword must be handled and the filename lookup must be by the Snort semantics, which are slightly different than the standard UNIX semantics. Variable definition and substitution must also be handled to use the rules as they are written.

Since Snort allows variable substitution, this must be simulated according to the Snort semantics. Snort treats variables as textual substitution and allows variables substitution to include other variable substitution in a non-recursive fashion. A recursive substitution would lead to an infinite loop because of a lack of control statements . The exact substitution model is not defined in the manual. In particular, if a variable is defined multiple times, it is not defined which definition has precedence or if this is an error. Some tests were constructed to determine how the variable substitution actually occurs. These indicated that a variable may be defined multiple times and the substitution of the most recently defined value occurs during each line of parsing. Substitution needs to occur during parsing to handle substitutions within filenames to include at that point in the input. A test was constructed to determine if all of the substitution happens recursively when the variable is substituted like "=" in the standard UNIX make utility or if the substitution within variables happens during the definition of the variable. Since the substitution happens during the definition of the variable, it is simulated this way. It is hoped that the variable substitution methods in Snort do not change. Since the details of the method have not been documented, there is little assurance that they will not change. In practice, judging by the complicated example configuration on the snort.org website, these fine points of variable substitution are not used. Each variable is defined once and is defined before it appears anywhere else.

The requirement that variable substitution happens where the variable appears in the file limits the use of generic variables in the parsed version of Snort. If the substitution is simulated truthfully, then the parsed Snort rules from the configuration file will have the variables already substitutions into. This is not desirable from a perspective of attempting to generate signature rules that are generic with respect to the underlying system. Variable substitution is useful especially for host and network addresses. One way around this is to break with the Snort substitution model by delaying the substitution of Snort variables except for "`include`" statements. Then the variables can be accessed according to operators in the generic language in the system. These variables can be set to values determined by the system, rather than by the Snort file. This is how Snort variable substitution occurs in the constructed system.

Another complication involved in reading Snort configuration files is that some slight transformations had to be performed upon the rules. These transformations encoded additional information in the message field of the rules. Since this field was printed when an alert was generated, this was parsed to associate the additional information with the alert. This solved a shortcoming of the call parameters of the output module used for Snort which was passed the string to print, but not the Snort identification number (SID) which corresponded to the rule. Hence the message to print field was transformed to include an identifier and the SID.

Sadly, despite only modifying a very small number of lines of an output module for Snort, several bugs were detected and corrected. These included a string that was not terminated properly in case of overflow and an improper string to check an output argument against. This was not a confidence inspiring experience for the security and robustness of Snort.

The second form of specifying the types of nodes was a specification of the interior nodes. This specification was a database of rules to run. The individual rules were specified in Scheme according with either the restriction to numerical finite automata or no restriction, hence a Turing Machine. As noted earlier, the restriction to the numerical finite automata language was left to the user in the system that was built.

Having a separate language for numerical finite automata facilitates a possible optimization. Some IDS systems including Snort have a module interface. Using this interface, C code can be developed to do arbitrary computations within Snort. In particular, code to interpret the numerical finite automata could be developed, hence the processing of this level of rules would occur inside of Snort which would be considerably more efficient due to a decrease in I/O for many types of alerts. In particular, the alerts used as input to the numerical finite automata set of rules for the generation of SOM inputs tend to generate a lot of alerts that are grouped together by the numerical finite automata. If these were handled in Snort instead, it should become nearly as efficient as the TCPTRACE module used with INBOUNDS. This optimization can also be performed without C code in IDS languages that are Turing Machines. For example, a numerical finite automata interpreter could be written for N-Code.

This optimization technique is not limited to numerical finite automata. Using a module interface, it is possible to embed a Scheme interpreter for the Turing Machine levels of this framework within Snort. Multiple inputs will cause problems, but otherwise, if the Turing Machine level only uses the output of the one node, it should be able to run within Snort. This disadvantages of putting this framework with Snort are that an entire Scheme interpreter is significantly more complex than a numerical finite automata interpreter and that Turing Machine code does not have a guaranteed upper bound on running time or space.

Once each rule was defined, they were inserted into a rule database. These rule databases constituted a configuration type for the interior nodes. Since each interior node is in a separate process space, there is no interaction between instances of the various databases. As noted in the section on general structure, these databases were not as helpful as anticipated for the organization of rules. Although they did present a uniform and systematic way of registering rules, the problem of combining rules together to form a configuration is more involved than just listing the rules to be run. Some rules need to be deactivated when a rule than applies to a more specific class of alerts is added and some rules should not be deactivated in this same case. The rule databases were helpful, but not sufficient by themselves to solve the organizational problems of grouping rules into node configurations.

The second component of the configuration language was a list that specified how the nodes were to be arranged into a tree. The assumption that the nodes can be arranged into a tree implies that the nodes can be written in a collection of nested lists such that each node appears once and appears as a list that is placed within the supernodes list of subnodes. For example, `(n1 (n2) (sn1 (sn2) (sn3)))` would correspond to node `n1` having subnodes `n2` and `sn1`, node `sn1` having subnodes `sn2` and `sn3`, and nodes `n2, sn2, sn3` having no subnodes. Since the type of node must also be specified, the syntax used in the list is `(hostname type subnodes...)`. For the common case of one IDS configuration and one interior node configuration, a special syntax without the types specified can be used by calling a different function. This is then syntax transformed into the standard syntax.

An early version of the system used a symbol based syntax instead. Each node would define a symbol that would be used for representing it when it was a subnode. Then this system of symbols was transformed into a tree for booting. This had the advantage that non-tree based configurations could be declared, but not used. It had the disadvantages of being more complicated and less clear, thus was abandoned.

One implementation issue is checking dependencies beyond the node configuration. Some interior rules should also have data dependencies which should be checked at compile time. For example, the SOM interior node module requires the addition of rules to the IDS subnode to generate the input for processing. If these rules are not added, then the SOM module will not function. As the system is currently implemented, only booting of the subnode is checked, not the configuration of the subnode. This could be fixed by adding an option for specification of dependency requirements to the declaration of interior rules. Then the dependencies of the rules could also be checked before booting the system.

4.3 Treatment of Time

The difficulties involved with the treatment of time have been noted in chapter 3. This section details which design decisions were taken. It discusses the internal structure of the representation of time, a time based internal queue and the external interface of the time representation.

Internally, time is represented as the number of microseconds since the Epoch (00:00:00 UTC, January 1, 1970) according to the standard POSIX method. Equivalently, this is the number of microseconds returned by the `gettimeofday` UNIX system call. Clock skew problems imply that the machine on which the time was recorded should also be recorded; this was not performed. Furthermore, it is convenient that the clocks obtain a modest level of synchronization before booting of the system to avoid possible clock warp issues. Most IDSs report the time at which a packet generating an alert was detected. This time was reported to the system which used this time rather than the time at which the system saw the alert.

The interface to this internal representation is correspondingly primitive. A proper interface would hide the internal representation in favor of a representation that is easier for the programmer to understand. Because of the simplicity of having one fundamental unit - the microsecond, the system simply lets the programmer use microseconds for everything. This design decision forces the programmer to use microseconds even if the programmer would desire different units. It often requires the programmer to carefully count to ensure the correct number of zeros in time arguments. An advantage of this design decision is that it forces the rounding problems and inexactness problems onto the user of the system. By requiring the number of microseconds to be an integer, rounding issues are now the problem of the programmer. Furthermore, the integer number of microseconds is an equivalence relation. Some support for approximate time operations has been provided to the programmer, of which get the current time is particularly useful.

A time based priority queue is part of the running structure of this system. This is useful for rules that give output at some delayed time. For example, the proposed real time extensions to the input module for the SOM report the status of a connection every 60 seconds. To simulate this, an alarm is needed every 60 seconds to generate the output. The time based priority queue can also be used for data collectors that flush the total number of alerts recently seen if there has not been an alert in some configurable number of seconds.

The implementation of a priority queue keyed on time is straightforward. It has features to add events at a given offset from the current time in addition to at an absolute time. The integration of the priority queue is taken to be a special case of a message pipe. Although it could be taken to be a normal message pipe, when used with polling I/O, the time until the next message from this is known, so the time for the `usleep` for the core interior node loop is the minimum of this time and the standard `usleep` value is taken instead. This reduces the latency for elements of the priority queue due to the sleep. For blocking I/O with threads, it is unnecessary to treat the priority queue as a special case.

This priority queue is mainly intended for use with the Turing Machine layers of this framework. If this priority queue is used for numerical finite automata based layers of this framework, the interface to the queue must be a delay action that causes the current finite automata to be suspended and resumed after the given delay. If instead simply registering a separate function with the priority queue were allowed (and multiple registrations by one computational element were allowed), the priority queue could be used for recursion, thus violating the restrictions on computational finite automata.

4.4 Data Collection

Data collectors were constructed to improve the frameworks handling of DOS attacks. A data collector runs on an interior node and simply takes the incoming alerts and sends on summaries with an exponential back-off. These summaries are collected by the higher level data collectors and the sub-results resummarized for reporting. This section discusses how the data collectors were built, the computational requirements for data collectors, the testing of the implemented data collectors and the trade offs for other design choices for data collectors.

The data collectors were constructed by one generic numeric finite automata with some additions. The collectors are registered in the rule database as applying as a default rule. So they will be called on every non-control alert that does not have a more specific rule. The collector then examines the trigger symbol, which is parsed by the parser particular to the message pipe type. The trigger symbol will differentiate output from another collector and a regular alert. If the output is from another collector at a subnode, then the number of rules counted is increased by the count given by the other collector. Likewise, other data structures like resource limited lists representing the alert types or the IP address to which they are connecting are also updated using the data supplied by the other collector. If the output is from a regular alert, then it is counted and the additional structures are updated. If the count exceeds an exponential threshold, implemented as twice the last threshold, then an alert is generated.

As implemented, only generating an alert after detecting the exceedince of the threshold uses an operation banned from use in numerical finite automata. Specifically, to do this an examination of the value for control switching purposes must be made. This was banned from the numerical finite automata because of the computational problems introduced by allowing this to happen.

To rectify the alert generation problem two approaches can be taken. The first approach is use data collectors in the Turing Machine layer instead. This is reasonable when the future work of integrating data collectors into dynamic sub-level reconfiguration is considered. Data collectors that involve multiple inputs may already be running at the Turing Machine layers of the system. If multiple inputs are involved, the node is unlikely to be integrated into the IDS for optimization reasons.

The second approach to rectifying this problem is to attempt to make the operations available to the numerical finite automata even more powerful without destroying the linear time bound. The alert generation option is where this will attempt to be added. It is required that node cannot send alerts to itself. This requirement is to prevent using the alert mechanism to produce recursion. In the Turing Machine layer, sending alerts to the node on which they originate is still allowed. It can also be required that a node which receives one alert cannot call the generate alert function more than once in the finite automata code. Encapsulated alert generation, from resource limited structures for example, can be given an exception as long as they generate at most an additional bounded number of alerts.

Now that the requirements to extend the numerical finite automata have been stated, the way of proceeding is to allow non-looping control statements that can examine the numerical operations within the alert generation function. The banning of control numerical operations was intended to prevent looping, the allowance of control numerical operations will allow looping here. The requirements placed upon the entrance of this code imply that it can only do one pass through a loop for every alert that it sees. Assuming fixed precision numbers, this preserves the linear running time requirement. As it does not allocate additional space, it does not impact the space requirement. Computationally, there is now an additional subtlety involved. There must be a distinction drawn between functions that can now be computed with one alert and functions that require multiple passes, hence multiple alerts.

The implemented data collectors were tested for performance in a DOS attack. The target IDS used for the tests was Snort. The stable Snort rule configuration given on the www.snort.org website was used for the Snort nodes. This configuration includes rules that cover ICMP and localhost traffic. Two of the rules were used for generating alerts. One rule detected ICMP pings that had 127.0.0.x as the originating IP address. Pings with these addresses would crash pre-1998 unpatched Solaris machines. The second rule was a restriction on traffic intended for the machine on which it is originating on.

10^6 pings that triggered this alert were generated using `ping -l 1000000`, which sends the pings as fast as they can be generated. This causes the monitoring IDS to send out alerts as fast as it can. With this rate of alerts, many of the ping packets are dropped by the IDS. The net result is that the IDS attempts to generate several times the amount of traffic that the pings generate for an interval of time starting with the beginning of the attack and ending once the attack has finished and the IDS has finished processing and outputting all of the data that it has not dropped. If this output is sent over the network, it makes the DOS problem considerably worse. Furthermore, if it is grouped together at one node, the node will need to deal with a multiplier of this amount of traffic, which makes it more likely that the system will be DOS to the users. It also makes it the case that the DOS may spread to machines and links beyond the initial path of the DOS via DOS through alert processing rather than the ICMP packets.

Using the data collectors, the output from the node under the DOS was only ten alerts, each of which gave a count of the number of alerts seen by the node. This prevented a significant increase in network traffic and processing by the higher levels of the system. Even though the target IDS was still dropping data at approximately the same rate, it was not extending the DOS past that node.

This was then tested with two nodes both receiving 10^6 pings. The collectors on each node outputted no more than ten alerts each. These were fed into a higher level collector on the root node, which output no more than eleven alerts. This demonstrated the ability to collect the data from the sublevels without losing the counting information.

Addressing the DOS at the target IDS is beyond the scope of this thesis. It is assumed that if the target IDS is DOS, then the data collectors only prevent the target IDS from generating many alerts. Except for the difference is speed of reporting, which should be a large difference if the data collector is compiled to run within the IDS, nothing is done to address the DOS of the target IDS.

Only tracking the number of alerts seen is of limited use. Since there are at a fixed finite number of possible alerts, the alerts that were seen were also tracked. This information was included in the output and state of the alert collectors. The collectors which have other collectors as input include the list of types of alerts seen by the collectors lower in the tree.

Tracking offending IP addresses is also useful, but must be done in a limited fashion. An attacker capable of transmitting packets with source addresses not originating on their subnet can cycle through a large number of source IP address. Tracking all of the source IP address could potentially involve 2^{32} addresses for IPv4, which is too many for the system to track. The solution taken for the implemented system was to use a resource limited list. If the number of address to track exceeded some bound, the additional address would not be tracked. Resource limited structures are discussed in the next section.

There are several options for the data collectors whose additions are debatable. These options are the reporting of overflow of resource limited structures, resetting the count and other data structures after no activity in some given time period and additional reporting when information is added to some structures.

The resource limited structures often have the configuration option to generate an alert when the first time additions are attempted past the fixed limit. The generation of this alert was not configured because the additional information gained seemed of little use for real time analysis. Instead, whether or not additions past the fixed limit had been attempted was included in the output of some versions of the data collectors.

Collecting the alerts before generating an alert to the parent node introduces a latency problem. Under the current implementation, the following somewhat undesirable behavior happens. If the collector sees 10^6 alerts then nothing for several days, it will not report on even 10^5 new alerts unless the system is reset. Similarly, during an ongoing DOS attack, the rate of alert generation continues to decrease by an exponential rate even at very low alert generation rates.

One way of fixing this is to flush the total and reset the counter if it has not seen any new alerts in some given number of seconds. Another similar approach is to exponentially decrease the threshold if no alerts has been generated in some given amount of time. This second approach should produce an alert at a relatively constant time interval for systems under a continuous DOS attack. Since there are trade-offs to each approach, it is not particularly important which one is chosen, only that one of them is chosen. In the implementation, the latency problem was not addressed.

4.5 Resource Limited Structures

Resource limited structures were introduced to increase the operations that could be performed using limited resources. They are used at two levels. With numerical finite automata, if the printing and handling of the contents is restricted to the alert generation function, then resource limited structures can be allowed with the impact on running time corresponding to the running time of the operations on the structures. This gives a substantial improvement in the information that can be returned by the numerical finite automata. For example, a resource limited hash table of IP addresses can be tracked by the numerical finite automata and printed upon generation of an alert. With the Turing Machine levels, resource limited structures are only a convenience to the programmer.

For example, ordinary list operations allow or give the allusion of allowing arbitrary length lists, whose usage is undesirable for resource limited computation. Arbitrary length lists can clearly exceed the reasonable bounds on space. Still, the usage of these operations is more convenient than treating everything as a preallocated fixed length array. Furthermore, if things are treated as a preallocated array, counters of what has been used must be kept, which add a substantial number of states to the finite state machine if the counters must be treated as separate states. Adding this number of states may push the number of states over the reasonable limits. If the counters are treated as numbers to do numerical operations upon for the numerical finite automata, it would also allow a number to be used as an index into memory, which probably increases the computational ability of the numerical finite automata. It is not necessarily undesirable, only more difficult to analyze. If numbers are to be used as indexices into memory, memory access must be range checked because the checking of bounds cannot be done by the numerical finite automata without examining the number. Instead of allowing the indexing of memory by numbers, it is proposed to give access to a list interface that handles the bound checking problem itself. This also simplifies the code for the numerical finite automatas.

A resource limited list was implemented. It had the option of configuring whether or not to generate an alert the first time that the finite configurable bound was exceeded. It was used by one version of the alert collector to store the list of which alerts were generated. Operations like union and intersection were defined on the limited list structure.

4.6 SOM Input Generation

The inputs to the SOM for INBOUNDS were generated as an extended numerical finite automata to demonstrate the practical sufficiency of this framework. The original input module for the SOM for INBOUNDS is written as a C-module for real time TCPTRACE, which is a network connection analysis tool that reads its data from libpcap, a tcpdump file or other network dump file formats. The demonstration of the input module for SOM as an extended numerical finite automata shows several things: first, it is an example of the sufficiency of basing higher computations on signature IDS alerts; second, it shows that the extended numerical finite automata are powerful enough to compute some useful output; third, since extended numerical finite automata are straightforward to optimize into IDSs that have a module interface, it shows that this method can be made close to as fast as the original C-module for real time TCPTRACE with a considerable gain in programmer efficiency.

The inputs needed for the SOM are six values per connection. The SOM is trained to recognize anomalies in these values when they are reported at the close of a connection. These values are detailed in [50, 51, 9, 43]. Their computation can be done on a per connection basis by storing the time of initiation of the connection along with the sum so far of the unnormalized variables and the direction of the last nonzero length transmission. At the close of the connection, these un-normalized values are normalized against the duration of the connection or the number of question-answer response cycles.

If the incoming packets are separated by connection, then a separate numerical finite automata for each connection can handle the generation of the inputs for the SOM. Since this is not explicitly allowed by the system, there are at least two ways of integrating it into the system. The first way is to recognize the importance of TCP based connections and allow special code to run to sort and deal with individual connections. Many IDSs do this and it seems to be a reasonable approach to deal with the TCP reconstruction problem. The second way of allowing these operations in the system is to allow the extended numerical finite automata to use a resource limited hash table. Then the connection data can be stored in a limited hash table and no special TCP code needs to be written. The slight drawback of this approach is that access to a hash table can be slower than constant time. Since the size of the hash table is limited, if the hash table is constructed using trees, the maximum lookup time will be logarithmic in the limited on the number of entries. Thus giving the extended finite automata access to hash tables solves this problem without excessively destroying the computational time bounds. This second approach was used by the implementation.

The inputs to the module for the inputs for the SOM were alerts generated by the IDS. The IDS used in the implementation was Snort. The different alerts were generated for TCP packets with different TCP flags combinations. One feature particular to Snort was used, when multiple rules match, Snort chooses the rule with the lowest Snort ID (SID). All of the rules that fed into the SOM were given larger SIDs than the other signature rules so that the remaining signature rules would still match if other misbehavior was detected. The rules for the generation of the input for the SOM input module had a slight order dependence. It was preferred to generate alerts for the reset TCP-flag over the FIN TCP flag if both were set. This is consistent with the interpretation of the reset TCP flag by TCP stacks [61].

The trained SOM itself was not embedded within the implemented system as it falls outside of the scope of this project. Only the inputs to be fed into the trained SOM were generated. Training the SOM is also outside of the scope of this project. It is not argued that the SOM cannot be embedded within this framework.

The development effort for this module measured in lines of code was considerably less than for the module running in real time TCPTRACE. The C code module is approximately 1000 lines of code, the Scheme module for this system is approximately 200 to 250 lines of code. Some of the reduction in lines of code is to be expected due to the language change before compensating for the changed interface. In general, the code necessary for memory allocation and deallocation in C and explicit bounds checking for every table access tend to consume a considerable proportion of the overall effort. Some of the reduction in lines of code may be due to a reduction in bit-bashing because the bit fields are handled in the IDS rules which feed into the module.

It was not needed to reconstruct TCP to deal with the generation of the inputs to the SOM. The implementation did not discard TCP session that had silently timed out. This can lead to a cluttering of the connection monitoring hash table if there are many missed FIN packets. One way of solving this is to introduce timers. The way to introduce timers without discarding the restrictions placed on the extended numerical finite automata is to treat the time as a number to be placed in a time based priority queue. When a new alert is being processed, a constant fixed number of these can be examined and removed if they are sufficiently old. This further expands the restriction on conditional number testing to include one constant length section per incoming alert.

Examples involving highly skewed test data were run to verify that the developed module produced accurate numbers. This test data was generated with standard UNIX utilities and various network services. For example, a test was constructed using the echo port and a script that generated one character per second. This was then run through the system to verify that the module produced the expected output.

4.7 Translation of Snort to N-Code

As noted in Chapter 3, the software has two usable stages. In the first stage, low level IDS components are referred to by symbols and need to have a corresponding instance for the target IDS. In the second stage, one generic low level representation is sufficient. Once the generic low level language is added, this system will form a complete language for IDS work. The complete second stage of the software was not implemented. Instead, an extended prototype intended to explore the possibilities of this lower level language was built.

Unlike existing work on intrusion signature modeling and translation, this work uses the higher layers of the system to fuse together lower level output that may not be directly translatable. LAMBDA only outputs rules for the signature based system, so any fundamental incompatibility cannot be dealt with [14]. As a prototype, this is intended as a translator between existing systems, rather than a translator from a distinct modeling language that does not correspond directly to any IDS.

This compiler was implemented as a translator from Snort to N-Code. N-Code is the language for the Network Intrusion Detection System from NFR Security, Incorporated. Prior to 1997, versions of this system were available for free. Internet postings indicate that in 1997, NFR Security stopped distributing the free version but stated that they would license it free for research usage. Unfortunately, currently it is not free for academic research usage.

There are neither multiple input nor multiple output modules for this compiler, so the complexity of translation to and from a system independent representation needed for retargetable compilation was saved. Furthermore, as N-Code is Turing Complete and allows access to the complete data stream, the output of the translation of Snort code could be almost completely in N-Code without the need for the generation of additional patch code running at the Turing Machine layer of the system to patch together the output of the N-Code alerts into the Snort alerts. The exception, which is the resolution of Snort alerts when multiple alerts can be generated for one packet, could be handled within N-Code at a substantial increase in complexity and running time.

The reverse translation from N-Code to Snort was not implemented. To do this translation requires interpreting the N-Code computations within either a Snort module or within the Turing Machine layers of this system. Building a N-Code interpreter is not particularly desirable because of the shortcomings of the N-Code language.

The compiler was implemented for a set of rules likely to be representative of Snort rules used in practice. The manual often differs from what is allowed in practice, so to do the compiler correctly, the actual behavior must be determined. In many cases, these language options are a superset of the options described in the manual. In some cases, the description in the manual is for a set of options that are not likely to be used in practice.

The stable rules collection from the www.snort.org website were used as a standard from which to gauge what and how options were used in practice. There were 1994 rules in the stable rules collection on the website. These rules were used to gauge the coverage of the compiler and to check for how the Snort options were used in practice. Of these rules, 1785 were able to be compiled. The RPC option accounted for 95 rules which were not compiled. The byte_test and byte_jump options accounted for 105 rules. The remaining nine rules that were not compiled had parsing problems.

The byte_test and byte_jump options were purposefully not implemented. They are straightforward to implement, but complicate the content checking code. Since the content checking code was already reasonably complicated, it was decided to delay the implementation of the byte_test and byte_jump options until after the content checking code was debugged in actual tests. The byte_jump option differs from the byte_test option only by specifying an optional positive displacement to resume matching at.

The RPC option decodes remote procedure call (RPC) traffic. It is straightforward to implement, but was not done because of implementor time considerations. In practice, there were only two distinct configurations of this Snort option: ‘‘rpc:100009,*,*’’ and ‘‘rpc:100000,*,*’’. The number represents the RPC application.

The nine rules that had parse errors were not deemed a sufficient number to warrant the effort involved in changing the parser. On page 16 of the Snort manual [54], it declares that the semicolon is used to separate the options. But in the documentation for the content rule, it declares that the semicolon can be used in strings provided that it is escaped by a backslash. To parse this properly, each option must be completely parsed before parsing the next option. One shortcut would be to just recognize strings or to use a parser generation tool. Since this system was implemented in Scheme, it was easiest to simply split the option field at the semicolon that the manual declared was the field separator. In practice, this failed for the nine rules that had escaped semicolons.

Snort process each rule by calling a registered function that then parses the arguments for that rule. This means that the Snort language is actually spread through the modules registered for each option. This explains the varied and unsystematic set of operations and arguments allowed for the different functions. Even worse, suboptions like the `nocase` option are placed in the fields like full options, yet apply to the last `content` or `uricontent` option. This is a major shortcoming of the Snort language.

Despite each option calling its own parser and having its own language, Snort is progressing toward some standard option field arguments. As the table 4.1 indicates, some fields have common languages. The Snort documentation describes each of these fields separately, [54] but the usage within these fields of more general language features often occurs. Specifically, the Snort manual may declare that only exact numeric values may appear as an argument, but some of the stable rules will specify a greater than or equal numeric comparison. It has been chosen to be liberal in what the compiler accepts rather than following the documentation. In every case, this allowed for accepting both the language as specified in the documentation and additional language features. The common languages for the Snort option fields were string arguments, bit field arguments, numeric test arguments, exact numeric arguments, and the null argument. The remainder of the Snort option fields were handled with special parsers. Often these performed a translation of a string into a numeric value for equality testing.

Table 4.1: Snort Options for Compilation

Snort Option	Superoption	Language	Comments
`ack`		Numerical test	TCP ack number
`byte_jump`		Special	Not implemented
`byte_test`		Special	Not implemented
`content-list`		Special	Not used in stable rules.
`content`		String	

Table 4.1: Continued

Snort Option	Superoption	Language	Comments
depth	content	Fixed integer	Maximum search depth.
dest. addr		Network addr.	Mandatory
dest. port		Port	Mandatory
direction		Special	Mandatory
distance	content	Fixed integer	Minimum distance between two content matches
dsize		Numerical test	
flags		Bit field	TCP flags
flow		Special	Direction of flow
fragbits		Bit field	IP fragmentation bits
fragoffset		Numerical test	Not used in stable rules.
icmp_id		Numerical test	ICMP echo ID
icmp_seq		Numerical test	ICMP echo sequence number
icode		Numerical test	ICMP code field
id		Numerical test	Fragment ID
ipoption		Special	Record route, etc.
ip_proto		Numerical test	Names are not implemented in Snort
itype		Numerical test	ICMP type
msg		String	Message printed on alert
nocase	content, uricontent	None	
offset	content	Fixed integer	
protocol		Special	Mandatory

Table 4.1: Continued

Snort Option	Superoption	Language	Comments
`rawbytes`		None	Preprocessor option, Not implemented
`react`			Not used in stable rules.
`regex`		Special	Not used in stable rules.
`resp`			Not used in stable rules.
`rpc`		Special	Not implemented
`sameip`		Numerical test	src_ip == dest_ip
`seq`		Numerical test	TCP sequence
`session`			Not used in stable rules.
`source IP`		Network addr.	Mandatory
`source port`		Port	Mandatory
`stateless`		None	Not used in stable rules.
`tag`			Not used in stable rules.
`tos`		Numerical test	Not used in stable rules.
`ttl`		Numerical test	
`uricontent`		String	Only matches in URI field of HTTP request
`within`	`content`	Fixed integer	Maximum distance between two content matches

String arguments were specified in a slightly different way in Snort and N-Code. Snort has a byte code escape character |, which is not present in N-Code. In N-Code each byte of the byte code sequence must be specified by \xff, where ff is the hexadecimal representation of the byte. It is straightforward to translate the strings including the byte-code sequences.

Bit field arguments had bits that were particular to the option field to match. It was implemented as a generic bit field compiler that took in a function to lookup the bit specified by the character in question. All of the bit fields took +, $*$ or no additional argument to specify whether to match on all of the flags, any of the flags or exactly the flags specified. An optional bit-mask field was provided that used the same translation of characters into bits as the bit field. For example, for the TCP bit field, "SR*" would match if either the syn flag or the reset flag were set in the packet. The bit flags field also took an optional beginning ! which inverted the match.

Numeric test arguments tested the given numeric value against the value specified by the field. One compiler that took in how to specify the value of the field was created. It handled the Snort not "!" option and the comparisons $<$ and $>$.

Some Snort option fields were actually sub-options of other options. For example, the `within` keyword was a sub-option of the `content` option. The semantics of `within` imply that it can only take an exact numeric value. It was handled by grouping via tree transformation all of the sub-options of `content` and `uricontent` into their respective super-option fields before compiling. Then the `within` keyword was handled within the compilation of the `content` option.

The `content` option is the most important and complex Snort option. It matches string patterns in the payload or TCP stream. When multiple `content` options are specified in one Snort rule, it is interpreted that each `content` rule must match on the payload or stream in turn separated by either an arbitrary number of characters or a number of characters bounded by `within` and `distance`. It is used by 1828 of the 1994 stable rules. The compilation of the `content` option requires state to handle stream based matches. These matches take place over many packets, so the partial matches must be stored along with a relative offset in the stream for the `distance` and `within` sub-options. A list of the partial matches suffices for state.

Snort rules have six mandatory fields. These fields specify the protocol, source IP address or netmask, destination IP address or netmask, source port, destination port, and direction of traffic. The protocol field specifies the stream type to watch: TCP, UDP, ICMP or IP. This field must be extracted to set some global flags for the compiler. The language that specifies the IP addresses and netmasks allows for non-nested lists and negation. The language that specifies the ports allows for possibly open ended ranges, but not lists of ports. For example, to watch ports 80 and 8080, it must be declared to watch all of the ports from 80 to 8080, rather than just the two ports. This will probably be fixed in upcoming versions of Snort.

Regular expressions are not handled by Snort except when they are encoded as a collection of content options. The Snort manual declares a `regexp` option, but states on page 31 that it "should not be used in production rule sets. As such, it will trigger an error condition if alerts are set using it" [54]. This is preferable to the approach taken by N-Code, which uses the operating system regular expression library with the OS syntax, rather than a system independent syntax.

Reconstructing TCP requires a considerable amount of state. If an attacker were to purposefully overflow the advertised buffer and the IDS did not monitor the send window advertisement or assumed that some acknowledgments were lost, then the IDS could be force to hold an arbitrary amount of data from a theoretical standpoint. Some engineering decisions must be made to deal with this worst case. The implemented TCP reconstruction assumes that all of the segments will be seen, but not necessarily in the correct order.

The implemented TCP reconstruction code does not run on an extended numerical finite automata. If the TCP reconstruction code sees a missing segment, it will produce output that may be many times as long as its input. Several fixes to this may be attempted. The first fix would be to join segments together as they are inserted into the pending segment queue. Then each segment would be processed in only about twice the join time. If the join is actually copying the segments, then this may still produce an arbitrary slowdown, so the solution is to just keep pointers. If only pointers are kept, the string matching code must be modified to handle this representation of streams.

After repeated emails, NFR Security indicated to the author of this thesis after about two months that they were no longer distributing a version of NFR to academics for free. Thus the compiled code could not be tested directly in N-Code. Although a N-Code interpreter could be built inside NML with only a moderate effort, this would not be an effective test of the code. The bugs in the code are likely to be due to manual ambiguity and interpretation problems, which would be faithfully modeled in a N-Code interpreter. Hence it would not test the major category of errors. Furthermore, there are a number of facets of the N-Code interpretation that are not specified by the manual, so would need to be explored using the actual product.

N-Code has several other limitations that make it undesirable as a general IDS programming language. The parser was implemented as easily as possible and lacks features desired in real programming languages. For example, the N-Code manual states on page 4-1 that N-Code does not have operator precedence. It even gives an example that in N-Code "$1 + 2 * 3 + 4$ evaluates as 13, not 11" [42]. As a source for the output of a compiled language, this precedence shortcoming is not important. N-Code has a list type, but does not allow the user to take a sublist. This seriously weakens the usefulness of N-Code lists. Since N-Code support associative arrays, these were used instead of lists.

Except for the rules containing parse problems or unimplemented Snort options, the remainder of the rules compile to N-Code. This is 1785 rules that compiled out of 1994 total rules. Each option was manually checked by itself against the N-Code documentation for bugs.

A slight inequivalence is introduced due to the resolution of ambiguous alerts in Snort. If one packet can generate two alerts, Snort appears to generate the alert with the lower Snort ID. In the compiled code in N-Code, each rule is in its own environment and generates alerts independently of the other rules. If the order to run the compiled rules could be specified, then this could be solved by a global variable. As it is, this was left as an implementation bug.

5. Conclusion

5.1 Conclusions

This thesis has shown a feasible higher level language system for distributed intrusion detection. The implementation results indicate that the system has the proposed properties of sufficiency of the language, efficacy of the language, usefulness for the administration of distributed systems, and attention to performance under load. Furthermore, the implementation results show that the system can be built. A useful first version has been outlined that gives many of the advantages of the system without the engineering complexity of constructing the full system.

This system obtains a measure of independence from the underlying IDS systems involved. This implies that configurations developed for this system can be unchanged despite optimizations or variations on the compiled configuration. This leads to a significant area to perform efficiency research and reap the benefits without impacting the usability.

Attention has been paid to ensure that the language features used for the low levels of the system are computationally reasonable for the indented purpose. This adds a degree of robustness despite the possibly dumb configurations added by users. It prevents whole classes of bugs that could hang or severely slow the lower levels of the system.

5.2 Future Work

This work lays the basis for many areas of future work. The most direct areas are the integration with resource management software, the expansion of the single language to include more run time topologies, more specifications, more target IDSs, and the optimizations for Snort or other module based IDS system.

The integration of this system with resource management software is a reasonably straightforward piece of research. It becomes interesting when the dynamic features are used to reconfigure the system on the fly. Many of the research questions involved in integrating the IDS systems with the resource management software are not particular to this system.

The expansion of the single language is an area of future research that is particular to this system. If it were expanded to include more primitive specifications, then it could be optimized further for the various IDS system. For example, N-Code does not have a primitive for matching on a pattern. The result is that each rule writer writes their own version, usually in the obvious but slow way of checking each string for a string match via `strcmp`. There are quicker ways of checking for string matching, but they are more effort than an individual rule write would like to expend. Now that each rule writer has used their own way of content matching, this cannot be automatically detected and the faster way substituted. Snort took the approach of using a content keyword, which probably initially used the slower method of matching, but now uses faster methods. Because the keyword was a distinct specification, they were able to substitute the functions.

Language features and run time mechanisms could be added to this system to support non-tree based run time configurations. This may have some usefulness when the network topology is known or when multiple output methods are desired, but at the present time it only adds to the engineering complexity.

The addition of more IDS systems for target configuration generation will further the usefulness of the system. This has two advantages. The first advantage is that the system can be used with more target IDSs. The second advantage is that by generating the output to more types of IDSs, aspects that are IDS dependent will become clearer.

Optimizations for Snort and other module based IDS systems are a useful area for further research. These optimizations would show significant speed ups for running the system for generating non-signature based analysis. By writing the interpreter for the lower level language, it would show that it is neither too complex nor too powerful.

Finally, using this system in practice would probably uncover more aspects that could use further examination. If the system proved to be highly useful, the language could eventually be made systematic and standardized.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Abelson, H., Sussman, G. J., and Sussman, J. *Structure and Interpretation of Computer Programs*, Second ed. MIT Press, Cambridge, MA, USA, 1996.

[2] Alessandri, D. Using Rule-Based Activity Descriptions to Evaluate Intrusion-Detection Systems. In *Recent Advances in Intrusion Detection: 3th International Workshop; proceedings (RAID 2000)* (Toulouse, France, October 2-4, 2000 2000), H. Debar, L. Me, and S. F. Wu, Eds., vol. 1907 of *Lecture Notes in Computer Science*, Springer, Berlin, pp. 183–196.

[3] Almgren, M., and Lindqvist, U. Application Integrated Data Collection for Security Monitoring. In *Recent Advances in Intrusion Detection: 4th International Symposium; proceedings (RAID 2001)* (Davis, CA, USA, October 10 - 12, 2001 2001), W. Lee, L. Me, and A. Wespi, Eds., vol. 2212 of *Lecture Notes in Computer Science*, Springer, Berlin, pp. 22–36.

[4] Anderson, J. Computer Security Threat Monitoring and Surveillance. Tech. rep., James P. Anderson Company, Fort Washington, Pennsylvania, USA, 1980.

[5] Asaka, M., Taguchi, A., and Goto, S. The Implementation of IDA: An Intrusion Detection Agent System. In *Proceedings of the 11th Annual FIRST Conference on Computer Security Incident Handling and Response (FIRST'99)* (1999).

[6] Axelsson, S. The Base-Rate Fallacy and Its Implications for the Difficulty of Intrusion Detection. In *ACM Conference on Computer and Communications Security* (1999), pp. 1–7.

[7] Axelsson, S. Intrusion Detection Systems: A Taxonomy and Survey. Tech. Rep. 99-15, Chalmers University of Technology, Goeteborg, Sweden, March 2000.

[8] Balasubramaniyan, J. S., Garcia-Fernandez, J. O., Isacoff, D., Spafford, E. H., and Zamboni, D. An Architecture for Intrusion Detection Using Autonomous Agents. Tech. Rep. TR 98-05, COAST Laboratory, Department of Computer Sciences, Purdue University, West Lafayette, Indiana, 1998.

[9] Balupari, R. Real-Time Network-Based Anomaly Intrusion Detection. Master's thesis, Ohio University, Athens, Ohio, June 2002.

[10] Balupari, R., Tjaden, B., Ostermann, S., Bykova, M., Tong, L., and Mitchell, A. Real-time Network-Based Anomaly Intrusion Detection. *Journal of Parallel and Distributed Computing Practices 4*, 2 (June 2001). Special Issue(Real Time Security).

[11] Brooks, F. P. *The Mythical Man-Month: Essays on Software Engineering*, anniversary ed. Addison-Wesley, 1995.

[12] Bykova, M., Ostermann, S., and Tjaden, B. Detecting Network Intrusions via a Statistical Analysis of Network Packet Characteristics. In *Proceedings of 33rd Southeastern Symposium on System Theory, SSST 2001* (Ohio University, Athens, Ohio, USA, March 2001).

[13] Cunningham, R. K., and Stevenson, C. S. Accurately Detecting Source Code of Attacks That Increase Privilege. In *Recent Advances in Intrusion Detection: 4th International Symposium; proceedings (RAID 2001)* (Davis, CA, USA, October 10 - 12, 2001 2001), W. Lee, L. Me, and A. Wespi, Eds., vol. 2212 of *Lecture Notes in Computer Science*, Springer, Berlin, pp. 104–116.

[14] Cuppens, F., and Ortalo, R. LAMBDA: A Language to Model a Database for Detection of Attacks. In *Recent Advances in Intrusion Detection: 3th International Workshop; proceedings (RAID 2000)* (Toulouse, France, October 2-4, 2000 2000), H. Debar, L. Me, and S. F. Wu, Eds., vol. 1907 of *Lecture Notes in Computer Science*, Springer, Berlin, pp. 197–216.

[15] Das, K. J. Attack Development for Intrusion Detection Evaluation. Master's thesis, Massachusetts Institute of Technology, June 2000.

[16] Debar, H., and Wespi, A. Aggregation and Correlation of Intrusion-Detection Alerts. In *Recent Advances in Intrusion Detection: 4th International Symposium; proceedings (RAID 2001)* (Davis, CA, USA, October 10 - 12, 2001 2001), W. Lee, L. Me, and A. Wespi, Eds., vol. 2212 of *Lecture Notes in Computer Science*, Springer, Berlin, pp. 85–103.

[17] Denning, D. E. *Cryptography and Data Security.* Addison-Wesley, Reading, Massachusetts, 1982.

[18] Denning, D. E. An Intrusion-Detection Model. Tech. Rep. SRI-CSL-87-5, Computer Science Laboratory, SRI International, Menlo Park, CA, June 1987.

[19] Dybvig, R. K. *The Scheme Programming Language*, 2nd ed. Prentice Hall, Upper Saddle River, NJ, 1996.

[20] Dybvig, R. K. *Chez Scheme User's Guide*. Cadence Research Systems, 1998.

[21] Fenet, S., and Hassas, S. A Distributed Intrusion Detection and Response System Based on Mobile Autonomous Agents Using Social Insects Communication Paradigm. In *Proceeding of the 1st International Workshop on Security of Mobile Multiagent Systems* (2001). Held at the Fifth International Conference on Autonomous Agents (Autonomous Agents'2001).

[22] Fyodor, Y. 'Snortnet' - A Distributed Intrusion Detection System. Kyrgyz Russian Slavic University, Bishkek, Kyrgyzstan. June, 2000.

[23] Goldman, R. P., and Geib, C. W. Plan Recognition in Intrusion Detection Systems. In *Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX)* (June 2001).

[24] Goldman, R. P., Heimerdinger, W., Harp, S. A., Geib, C., and Vicraj, T. Information Modeling for Intrusion Report Aggregation. In *Proceedings of the DARPA Information Survivability Conference and Exposition* (June 2001), IEEE Computer Society, IEEE, pp. 329–342.

[25] Gopalakrishna, R. A Framework for Distributed Intrusion Detection Using Interest Driven Cooperating Agents. Paper for qualifier II examination, Department of Computer Sciences, Purdue University, West Lafayette, Indiana, May 2001.

[26] Haines, J. W., Lippmann, R. P., Fried, D. J., Zissman, M. A., Tran, E., Boswell, S. B., and 62", G. 1999 DARPA Intrusion Detection Evaluation: Design and Procedures. Tech. Rep. 1062, Massachusetts Institute of Technology Lincoln Laboratory, Lexington, Massachusetts, February 2001.

[27] Howard, J. D. *An Analysis of Security Incidents on the Internet 1989-1995*. PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, April 1997.

[28] Jansen, W., Mell, P., Karygiannis, T., and Marks, D. Mobile Agents in Intrusion Detection and Response. In *12th Annual Canadian Information Technology Security Symposium* (Ottawa, Canada, 2000).

[29] Jou, Y., Gong, F., Sargor, C., Wu, X., Wu, S., Chang, H., and Wang, F. Design and Implementation of a Scalable Intrusion Detection System for the Protection of Network Infrastructure. In *DARPA Information Survivability Conference and Exposition 2000* (January 2000), vol. 2, pp. 69–83.

[30] Kendall, K. A Database of Computer Attacks for the Evaluation of Intrusion Detection Systems. Master's thesis, Massachusetts Institute of Technology, June 1999.

[31] Ko, C., Brutch, P., Rowe, J., Tsafnat, G., and Levitt, K. System Health and Intrusion Monitoring Using a Hierarchy of Constraints. In *Recent Advances in Intrusion Detection: 4th International Symposium; proceedings (RAID 2001)* (Davis, CA, USA, October 10 - 12, 2001 2001), W. Lee, L. Me, and A. Wespi, Eds., vol. 2212 of *Lecture Notes in Computer Science*, Springer, Berlin, pp. 190–203.

[32] Korba, J. Windows NT Attacks for the Evaluation of Intrusion Detection Systems. Master's thesis, Massachusetts Institute of Technology, June 2000.

[33] Kruegel, C., and Toth, T. Distributed Pattern Detection for Intrusion Detection. In *Network and Distributed System Security Symposium Conference Proceedings: 2002* (1775 Wiehle Ave., Suite 102, Reston, Virginia 20190, U.S.A., 2002), Internet Society.

[34] Kumar, S. *Classification and Detection of Computer Intrusions*. PhD thesis, Purdue University, West Lafayette, Indiana, 1995.

[35] Lippmann, R., Haines, J. W., Fried, D. J., Korba, J., and Das, K. Analysis and Results of the 1999 DARPA Off-Line Intrusion Detection Evaluation. In *Recent Advances in Intrusion Detection: 3th International Workshop; proceedings (RAID 2000)* (Toulouse, France, October 2-4, 2000 2000), H. Debar, L. Me, and S. F. Wu, Eds., vol. 1907 of *Lecture Notes in Computer Science*, Springer, Berlin, pp. 162–182.

[36] Martin, J. C. *Introduction To Languages and the Theory of Computation*, 2nd ed. McGraw-Hill, 1997.

[37] McHugh, J. The 1998 Lincoln Laboratory IDS Evaluation (A Critique). In *Recent Advances in Intrusion Detection: 3th International Workshop; proceedings (RAID 2000)* (Toulouse, France, October 2-4, 2000 2000), H. Debar, L. Me, and S. F. Wu, Eds., vol. 1907 of *Lecture Notes in Computer Science*, Springer, Berlin, pp. 145–161.

[38] Me, L., and Michel, C. Intrusion Detection: A Bibliography. Tech. Rep. SSIR-2001-01, Supelec, September 2001.

[39] Michel, C., and M, L. ADeLe: an Attack Description Language for Knowledge-based Intrusion Detection. In *Proceedings of the 16th International Conference on Information Security (IFIP/SEC 2001)* (June 2001), pp. 353–365.

[40] Mounji, A., Charlier, B. L., Zampunieris, D., and Habra, N. Distributed Audit Trail Analysis. In *Proceedings of the Internet Society Symposium on Network and Distributed System Security (ISOC '95)* (San Diego, California, February 1995), IEEE.

[41] Neumann, P. G., and Porras, P. A. Experience with EMERALD to Date. In *First USENIX Workshop on Intrusion Detection and Network Monitoring* (Santa Clara, California, apr 1999), pp. 73–80.

[42] NFR Security. *NFR Network Intrusion Detection System N-Code Guide.* Rockville, Maryland, 2001.

[43] Nguyen, B. V. Self Organizing Map (SOM) for Anomaly Detection. Spring 2002.

[44] Ning, P., Jajodia, S., and Wang, X. S. Abstraction-Based Intrusion Detection in Distributed Environments. *Information and System Security 4*, 4 (2001), 407–452.

[45] Northcutt, S., and Novak, J. *Network Intrusion Detection: An Analyst's Handbook*, 2nd ed. Prentice Hall, 2000.

[46] Perrin, D. *Handbook of Theortical Computer Science*, vol. B: Formal Models and Semantics. MIT Press, 1990, ch. 1: Finite Automata.

[47] Porras, P. A., Fong, M. W., and Valdes, A. A Mission-Impact-Based Approach to INFOSEC Alarm Correlation. In *Recent Advances in Intrusion Detection: 5th International Symposium, RAID 2002, Proceedings* (Zurich, Switzerland, October, 2002 2002), A. Wespi, G. Vigna, and L. Deri, Eds., vol. 2516 of *Lecture Notes in Computer Science*, Springer, Berlin, pp. 95–114.

[48] Porras, P. A., and Neumann, P. G. EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances. In *1997 National Information Systems Security Conference* (oct 1997).

[49] Pouzol, J.-P., and Ducasse, M. From Declarative Signatures to Misuse IDS. In *Recent Advances in Intrusion Detection: 4th International Symposium; proceedings (RAID 2001)* (Davis, CA, USA, October 10 - 12, 2001 2001), W. Lee, L. Me, and A. Wespi, Eds., vol. 2212 of *Lecture Notes in Computer Science*, Springer, Berlin, pp. 1–21.

[50] Ramadas, M. Detecting Anomalous Network Traffic With Self-Organizing Maps. Master's thesis, Ohio University, Athens, Ohio, November 2002.

[51] Ramadas, M., Ostermann, S., and Tjaden, B. Detecting Anomalous Network Traffic with Self-Organizing Maps. In *Proceedings of the 6th International Symposium on Recent Advances in Intrusion Detection, RAID 2003* (Pittsburgh, PA, USA, September 8-10 2003).

[52] Ranum, M. J. Experiences Benchmarking Intrusion Detection Systems. Tech. rep., NFR Security, 2001.

[53] Riordan, J., and Alessandri, D. Target Naming and Service Apoptosis. In *Recent Advances in Intrusion Detection: 3th International Workshop; proceedings (RAID 2000)* (Toulouse, France, October 2-4, 2000 2000), H. Debar, L. Me, and S. F. Wu, Eds., vol. 1907 of *Lecture Notes in Computer Science*, Springer, Berlin, pp. 217–226.

[54] Roesch, M., and Green, C. *Snort Users Manual*, July 2003.

[55] Russell, S., and Norvig, P. *Artificial Intelligence: A Modern Approach.* Prentice Hall, Upper Saddle River, NJ 07458, 1995.

[56] Sekar, R., Guang, Y., Verma, S., and Shanbhag, T. A High-Performance Network Intrusion Detection System. In *ACM Conference on Computer and Communications Security* (1999), pp. 8–17.

[57] Stallings, W. *Network Security Essentials.* Prentice Hall, Upper Saddle River, NJ, 2000.

[58] Staniford-Chen, S. Distributed Tracing of Intruders. Master's thesis, University of California at Davis, 1995.

[59] Staniford-Chen, S., Cheung, S., Crawford, R., Dilger, M., Frank, J., Hoagland, J., Levitt, K., Wee, C., Yip, R., and Zerkle, D. GrIDS – A Graph-based Intrusion Detection System for Large Networks. In *Proceedings of the 19th National Information Systems Security Conference* (1996).

[60] Sterne, D., Djahandari, K., Wilson, B., Babson, B., Schnackenberg, D., Holliday, H., and Reid, T. Autonomic Response to Distributed Denial of Service Attacks. In *Recent Advances in Intrusion Detection: 4th International Symposium; proceedings (RAID 2001)* (Davis, CA, USA, October 10 - 12, 2001 2001), W. Lee, L. Me, and A. Wespi, Eds., vol. 2212 of *Lecture Notes in Computer Science*, Springer, Berlin, pp. 134–149.

[61] Stevens, W. R. *TCP/IP Illustrated: the Protocols*, vol. 1. Addison Wesley Longman, 1994.

[62] Tjaden, B., Welch, L., Ostermann, S., Chelberg, D., Masters, M., Werme, P., Marlow, D., IV, P. I., Chapell, B., Balupari, R., Bykova, M., Mitchell, A., Lissitsyn, D., and Tong, L. INBOUNDS: The Integrated Network-Based Ohio University Network Detective Service. In *Proceedings of the 4th World Multiconference on Systemics, Cybernetics and Informatics, SCI 2000 and The 6th International Conference on Information Systems, Analysis and Synthesis, ISAS 2000* (Orlando, Florida, USA, July 2000).

[63] Uppuluri, P., and Sekar, R. Experiences with Specification-Based Intrusion Detection. In *Recent Advances in Intrusion Detection: 4th International Symposium; proceedings (RAID 2001)* (Davis, CA, USA, October 10 - 12, 2001 2001), W. Lee, L. Me, and A. Wespi, Eds., vol. 2212 of *Lecture Notes in Computer Science*, Springer, Berlin, pp. 172–189.

[64] Valdes, A., and Skinner, K. Adaptive, Model-based Monitoring for Cyber Attack Detection. In *Recent Advances in Intrusion Detection (RAID 2000)* (Toulouse, France, October 2000), H. Debar, L. Me, and F. Wu, Eds., no. 1907 in Lecture Notes in Computer Science, Springer-Verlag, pp. 80–92.

[65] Valdes, A., and Skinner, K. Probabilistic Alert Correlation. In *Recent Advances in Intrusion Detection: 4th International Symposium; proceedings (RAID 2001)* (Davis, CA, USA, October 10 - 12, 2001 2001), W. Lee, L. Me, and A. Wespi, Eds., vol. 2212 of *Lecture Notes in Computer Science*, Springer, Berlin, pp. 54–68.

[66] Vigna, G. Inspect: a Lightweight Distributed Approach to Automated Audit Trail Analysis.

[67] Vigna, G., Kemmerer, R. A., and Blix, P. Designing a Web of Highly-Configurable Intrusion Detection Sensors. In *Recent Advances in Intrusion Detection: 4th International Symposium; proceedings (RAID 2001)* (Davis, CA, USA, October 10 - 12, 2001 2001), W. Lee, L. Me, and A. Wespi, Eds., vol. 2212 of *Lecture Notes in Computer Science*, Springer, Berlin, pp. 69–84.

[68] Wang, F., Gong, F., Trivedi, K., and Coseva-Popstojanova, K. SITAR: A Scalable Intrusion-Tolerant Architecture for Distributed Services. In *2nd Annual IEEE SMC Information Assurance Workshop* (New York, 2001).

[69] Weber, D. A Taxonomy of Computer Intrusions. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, 1998.

[70] Welz, M., and Hutchison, A. Interfacing Trusted Applications with Intrusion Detection Systems. In *Recent Advances in Intrusion Detection: 4th International Symposium; proceedings (RAID 2001)* (Davis, CA, USA, October 10 - 12, 2001 2001), W. Lee, L. Me, and A. Wespi, Eds., vol. 2212 of *Lecture Notes in Computer Science*, Springer, Berlin, pp. 37–53.

[71] Williams, P. D., Anchor, K. P., Bebo, J. L., Gunsch, G. H., and Lamont, G. D. CDIS: Toward a Computer Immune System for Detecting Network Intrusions. In *Recent Advances in Intrusion Detection: 4th International Symposium; proceedings (RAID 2001)* (Davis, CA, USA, October 10 - 12, 2001 2001), W. Lee, L. Me, and A. Wespi, Eds., vol. 2212 of *Lecture Notes in Computer Science*, Springer, Berlin, pp. 117–133.

[72] Yang, J., Ning, P., Wang, X. S., and Jajodia, S. CARDS: A Distributed System for Detecting Coordinated Attacks. In *Proceedings of Sixteenth Annual Working Conference on Information Security (SEC 2000)* (August 2000), S. Qing and J. H. P. Elof, Eds., Kluwer Academic, pp. 171–180.

APPENDIX

A. Prototype Details

The appendices to this thesis give details on the prototype software, such that the prototype system could easily be reconstructed. When in doubt as to whether or not a prototype detail or prototype design decision detail is necessary for the construction of another prototype system, the detail has been included. Hence, these appendices lean toward the specification of the prototype in greater detail than is necessary for reconstruction.

Appendix A contains general information about the prototype, but does not include information on the Snort to NFR N-Code compiler. As may be noted from the title of this thesis “Compilation for Intrusion Detection Systems”, the emphasis in this thesis has been on the process of compilation, not on the language to compile. This emphasis on the semantic aspects of the language, rather than the syntax is both for software engineering reasons and as a direct response to the failure of the reverse approach of the IETF standardization effort. For software engineering reasons, it is necessary to specify the semantic aspects of the language before specification of the syntax. This thesis does not contribute the full semantics for the proposed language, so any syntax given for the language is premature. The thesis does contribute indications that such a language is possible. Indeed, the prototype is a demonstration of such a system. As this demonstration relates to appendix A, the whole system is detailed, as this is the proposed context for the language.

Appendix B contains the justification for the design choices in the compilation of the Snort rule language. It contains a table of example rules which cover most of the cases of the actual usage of Snort. This table is used as justification for the generation of a well defined Snort language used by the compiler. This Snort language differs from the language described in [54]. In any case, commonalities between option fields were exploited to simplify the task of compilation of Snort rules.

Appendix C contains details about the implemented prototype compiler from Snort to N-Code. It includes examples to demonstrate how various components work. The shortcomings of the prototype compiler are noted both in appendix C and in section 4.7.

The inclusion of the Scheme code in the figures is meant to complete the details of the descriptions. In choosing between space in this thesis and describing exactly how the prototype works, usually, but not always, describing exactly how the prototype works was preferred. This choice of preferring to describe implemented functionality is consistent with the goals of demonstrating the prototype and easing the task of future implementors. Since the code for the system was written in a powerful programming language, the code is not very long, especially considering the functionality. If the prototype system were written in a more traditional programming language like 'C' or 'C++', the code would be at least a factor of two longer than it is. These appendices do not include an introduction to the programming language Scheme. The reader looking for more information about Scheme is invited to consult Dybvig [19] or Abelson [1].

The constructed system was a prototype, not a refined product. The code was intended to construct sufficient functionality of the system so that the implementation based tests could function. Much of this prototype should not be used as a basis for a more extensive system. It is a common software engineering mistake to attempt to turn a prototype system into a full featured maintainable system. The goals in developing the two systems are different, specifically, for the prototype, every corner possible is cut to attempt to resolve the critical questions, whereas for a full featured maintainable system extensive corner cutting is counterproductive. The code that appears in the figures in these appendices should be treated as prototype code. The figures show one way of constructing the system with the goal of minimizing the initial software development time. Lest this code be criticized for aesthetic or software engineering reasons, the last point is repeated: the goal during the development of the prototype capable of demonstrating basic functionality of the system was to minimize software development time, not to produce a full featured maintainable system.

Appendix A is organized as follows: section A.1 discusses the general structure of the system with subsections on specific aspects of the system; section A.2 discusses the implementation of resource limited structures; section A.3 discusses the implemention of an alert collector for DOS attacks; section A.4 discusses the generation of inputs for the self organizing map intrusion detection module.

A.1 General Structure

Figure A.1 is the main loop for the run time system. It iterates through the data and control streams checking for new data or commands to process. When it finds new data or a new command, it dispatches the processing function associated with that data or command stream. In addition, it defines the tear-down procedure that is called by this node to tear-down this node and this node's sub-nodes of the system. As noted in section 4.1, this prototype was constructed using polling I/O.

Message passing is used in this system as an object oriented encapsulation technique. For example, in figure A.1, different types of message pipes will perform different operations for 'do-msg, but they are all called by the same mechanism. In addition, message passing enforces the encapsulation. For example, *proc-msg-pipes* can not access the internal data of a message pipe because only the function interface is exposed. Furthermore, functions are not mutable in Scheme.

A.1.1 Message Pipes

The message pipe abstraction encapsulates a data stream along with the corresponding processing mechanisms. Figure A.2 lists the general encapsulation. The data stream is *input-port*, which is further encapsulated by *make-line-buffer*. The

```
(define (proc-msg-pipes pipes) ; pipes could also stand to be global
  (set! *clean-up* (lambda ()
                     (map (lambda (p) (p 'close)) pipes)
                     (exit)))
  (let loop ((l pipes)
             (did-one #f))
    (cond
     [(null? l)
      (if (not did-one) (sleep-usec 20000))
      (loop pipes #f)]
     [((car l) 'msg-ready?)
      ((car l) 'do-msg)
      (loop (cdr l) #t)]
     [else (loop (cdr l) did-one)])))
```

Figure A.1: Main System Loop

processing mechanisms are *parser* and *eval-f*. The external interface only takes the commands 'msg-ready?, 'do-msg, and 'close. Figures A.3, A.4, and A.5 list the implemented types of message pipes. A message pipe type for NFR was not implemented, as an actual copy of NFR was not obtained, section 4.7 contains more details on the failure to obtain a copy of the NFR IDS.

As may be noted in figure A.4, this version of the system bases the configured system on an existing Snort configuration file. This Snort configuration file is parsed and transformed into a configuration file for Snort. The transformation of Snort rules is necessary to include additional information in the reporting of Snort alerts using the chosen output module. It is detailed in figure A.21. This version of the system does not support multiple distinct Snort configurations. Section 4.2 details reasons for supporting a distinct Snort configuration file for every Snort node. An earlier version of the system required the Snort rules to be given in a parsed form as an argument to *make-snort-msg-pipe*. Although allowing *make-snort-msg-pipe* to accept lists of rules is more useful than only accepting Snort configuration filenames,

```
(define make-msg-pipe
  (lambda (input-port parser eval-f close-f . l)
    (assert (input-port? input-port))    ; may want one reading
                                         ; from a local structure...
    (let ((l-buf (make-line-buffer input-port)))
      (if (and (not (null? l)) (procedure? (car l))) ; boot procedure
          ((car l) l-buf))
      (lambda (cmd)
        (case cmd
          [(msg-ready?) (l-buf 'line-ready?)]
          [(do-msg) (eval-f (parser (l-buf 'get-line)))]
          [(close)
           (close-f)
           (l-buf 'close)]
          [else (error 'msg-pipe "Unknown cmd ~s" cmd)])))))
```

Figure A.2: General Message Pipe

make-snort-msg-pipe was changed to accepting Snort configuration filenames instead because this is how the system was normally used.

Section 4.2 briefly discusses the method by which this system performs a teardown. Snort nodes are sent a `SIGKILL` signal by their parent nodes. Statistical nodes, which are all interior nodes, send their subnodes the (∗*clean-up*∗) command before closing their data streams and exiting. These functions can be observed as an argument to *make-msg-pipe* in figures A.4 and A.5. Any broadcast to all of the nodes could be done by a similar method. A general broadcast mechanism was not implemented because ∗*clean-up*∗ was the only command that was broadcast in this prototype.

A.1.2 Input Port Encapsulation

Figure A.6 shows the encapsulation of input streams into line buffered input. The encapsulation eases system development because the messages are line delimited. This is one way of handling this in a generic way for all of the input types. The interface commands are 'eof?, 'line-ready?, 'peek-line, 'get-line, and 'close. The

```
(define (make-stdin-msg-pipe)
  (make-msg-pipe (current-input-port)
                 parse-scheme-line
                 eval
                 (lambda () #t)))
```

Figure A.3: Scheme Standard Input Message Pipe

current implementation is not as fast as possible due to the unnecessary translations of string representations between a buffer of characters and a list of characters by *make-line-buffer* and other functions. Optimizations like avoiding string representation translations are not important to the proof of concept given by the prototype. Buffer overflows attacks against this system are not an issue due to the checking of memory bounds by the Scheme run time system. There are valid reasons to limit the input buffer size. If one message is greater than 10MB, it would seriously degrade the system to handle it. Code could be added to the *make-line-buffer* routine for a way to limit the inputs for every message type.

A.1.3 Parsing

The parsing of Scheme is the easiest of the data streams to parse, so that is treated first. Next, utilities for parsing other input methods are detailed, followed by the parsing of alerts generated by Snort with an example. Next, section 1 covers parsing of the Snort rules. Finally, in section 2 the parsing of Snort configuration files is treated.

Figure A.7 details the parsing of Scheme for the input to a node. Unsurprisingly, it simply calls the underlying Scheme mechanism for parsing Scheme input.

The parsing of other data streams was handled in a top down fashion with a simple utility routine. Strings were split based upon some token. This splitting on various tokens was powerful enough to handle the configuration files and all but nine of the rules for Snort. Section 4.7 details the rational behind this choice for parsing

```
(define make-snort-msg-pipe
  (lambda () ; no args yet, eventually hostname, config
    (let ((proc (process
                 (string-append
                  "exec "
                  snort-bin
                  " -i "
                  snort-ether-interface
                  " -c "
                  (existing-snort-conf->idsds snort-config ".fixme_host")
                  " 2> /dev/null"))))
      (and (list? proc) (eqv? 3 (length proc)) (<= 1 (caddr proc))
           (make-msg-pipe (car proc)
                          parse-csv
                          *ids-msg-eval*
                          (lambda ()
                            (kill 'SIGTERM (caddr proc))))))))
```

Figure A.4: Snort Message Pipe

```
(define (make-stat-msg-pipe)
  (let ((proc (process
               (string-append
                "exec "
                scheme-bin
                " "
                this-file
                " 2> /dev/null"))))
    (and (list? proc) (eqv? 3 (length proc)) (<= 1 (caddr proc))
         (make-msg-pipe (car proc)
                        parse-scheme-line
                        *stat-msg-eval*
                        (lambda () (fprintf (cadr proc) "~s~n"
                                            '(*clean-up*)))
                        (make-boot-stat (cadr proc) '(do-two-snort))))))
```

Figure A.5: Statistical Node Message Pipe

```
 ; line buffered IO
(define make-line-buffer
  (lambda (port)
    (let* ((l '())
          (eof? #f)
          (is-line?
           (lambda z
             (let ((x (if (null? z) l (car z))))
               (and (pair? x) (not (null? x)) (eqv? (car x) #\newline)))))
          (peek-line  ; no peeking at unfull lines
           (lambda ()
             (list->string (reverse (cdr l)))))
          (buffer-read-char
           (lambda ()
             (let ((c (read-char port)))
               (if (eof-object? c)
                  (set! eof? #t)
                  (set! l (cons c l))))))
          (line-ready?  ; returns false on EOF w/o a line
           (lambda ()
             (let loop ()
               (cond
                [(is-line?) #t]
                [eof? #f]
                [(char-ready? port)
                 (buffer-read-char)
                 (or (is-line?) (loop))]
                [else #f])))))

       (lambda (cmd)
         (case cmd
           [(eof?)
            (and (not (line-ready?))
                 eof?
                 ; (null? l) - drop unfinished lines
                 )]
           [(line-ready?)
            (line-ready?)]
```

Figure A.6: Line Buffered Input Encapsulation

```
            [(peek-line)
             (peek-line)]
            [(get-line)
             (assert (line-ready?))
             (let ((r (peek-line)))
                (set! l '())
                r)]
            [(close)
             (close-input-port port)]
            [else (printf "missed case")]
            )))))
```

Figure A.6: Continued

```
(define parse-scheme-line
  (lambda (s)
    (let* ((p (open-input-string s))
           (x (read p))) ; how does this handle blank lines?
      (close-input-port p) ; unnecc.
      x)))
```

Figure A.7: Scheme Input Parsing

the Snort rules. The splitting routine and a helper routine for the splitting routine are detailed in figure A.8. An inverse of the splitting routine, which is used at some points in the system, is detailed in figure A.9.

A.1.3.1 Parsing Snort Output

The parsing of Snort uses the splitting routine to parse alerts from a specific Snort output module. The output module used for Snort running in this system prints fields for the alert in a comma separated list. The output module is called ‘csv’ for comma separated values. A few minor modifications were made to the ‘C’ Snort code for the output module consisting of minor bug fixes, enabling the output module to write to standard output, and the addition of the option for the output of an additional field. The default output configuration string and the output configuration string

```
(define (split token str)
  (map list->string
        (letrec
            ((token-ref-loop
              (lambda (l ref)
                (cond
                 [(null? l) #f]
                 [(eqv? (car l) token) ref]
                 [else (token-ref-loop (cdr l) (+ 1 ref))])))
             (token-ref
              (lambda (l)
                (token-ref-loop l 0))))
          (let loop
              ((x (string->list str)))
            (let ((r (token-ref x)))
              (if r
                 (let ((y (split-at-ref x r)))
                   (cons (car y) (loop (cddr y)))) ;cddr deletes the token
                 (list x)))))))

 ; split-at-eq? would make it clearer
(define (split-at-ref l ref) ; slow implementation
  (cond
   [(< ref 0) (error 'split-at-ref "bad argument")]
   [(zero? ref) (cons '() l)]
   [(null? l) (error 'split-at-ref "reference longer than argument")]
   [else
    (let ((r (split-at-ref (cdr l) (- ref 1))))
      (cons (cons (car l) (car r)) (cdr r)))]))
```

Figure A.8: Token Splitting Routines

```
(define (unsplit token l) ; undoes split
  (cond
   [(null? l) ""]
   [(null? (cdr l)) (car l)]
   [else
    (string-append (car l)
                   (string token)
                   (unsplit token (cdr l)))]))
```

Figure A.9: Inverse of Splitting Routine

```
 ;tcpln is a bug and typo in Snort CSV output module
(define csv-default-output-string
  (string-append  ; broken into substrings for display in thesis
   "timestamp,msg,proto,src,srcport,dst,dstport,ethsrc,ethdst,ethlen,"
   "tcpflags,tcpseq,tcpack,tcpln,tcpwindow,ttl,tos,id,dgmlen,iplen,"
   "icmptype,icmpcode,icmpid,icmpseq"))

(define csv-output-string
  "msg,timestamp,usectime,src,srcport,dst,dstport,iplen,tcplen,dgmlen")
```

Figure A.10: Snort Output Configuration Strings

used by this system is shown in figure A.10. Example output from Snort configured with these output configuration strings is shown in figure A.11.

The comma separated values from Snort are further parsed by splitting the lines at the commas. Then the output field names are associated with the field values to form an association list. This code is shown in figure A.12 and an example is shown in figure A.13.

The parsed Snort output is accessed through functions that deal with extracting the relevant data when the relevant data does not directly correspond to an output field. For example, the size of a TCP segment is not given as an output field, so it is computed from the entire IP datagram size minus the IP header size minus the TCP header size. Figure A.14 shows details.

With *csv-default-output-string* configured:

```
12/01-10:05:24.746579 ,sid 528 BAD-TRAFFIC loopback traffic,ICMP,127.0.0.1,,127.0.0.1,,0:0:0:0:0:0,0:0:0:0:0:0,0x62,,,,,,64,0,0,84,20,8,0,,
```

With *csv-output-string* configured:

```
sid 528 BAD-TRAFFIC loopback traffic,12/01-10:16:37.949747 ,1070291797949747,127.0.0.1,,127.0.0.1,,20,,84
```

Figure A.11: Example Snort Outputs

```
(define (parse-csv csv-line)
  (apply map cons (map (lambda (l) (split #\, l))
                       (list csv-output-string csv-line))))
```

Figure A.12: Parsing Snort Output

With *csv-output-string* configured:

```
(("msg" . "sid 528 BAD-TRAFFIC loopback traffic")
 ("timestamp" . "12/01-10:16:37.949747 ")
 ("usectime" . "1070291797949747")
 ("src" . "127.0.0.1")
 ("srcport" . "")
 ("dst" . "127.0.0.1")
 ("dstport" . "")
 ("iplen" . "20")
 ("tcplen" . "")
 ("dgmlen" . "84"))
```

Figure A.13: Example of Snort Output Parsing

```
(define (get-alert-field field alert)
  (let ((x (assoc field alert)))
    (and x (cdr x)))) ; or if/then

(define (get-alert-sid alert)
  (let ((x (get-alert-field "msg" alert)))
    (and x (string->number
             (cadr (split #\space x)))))) ;eventually do something fancier.

(define (get-alert-tcpseg-size alert)
  (let ((f (lambda (x) (let ((y (get-alert-field x alert)))
                          (if y (string->number y) 0)))))
    (- (f "dgmlen")
       (+ (f "iplen") (f "tcplen")))))
```

Figure A.14: Accessing Snort Alert Data

A.1.3.2 Parsing Snort Configuration Files

Parsing Snort configuration files involves recognizing the Snort configuration file syntax and properly interpreting the Snort variables. Variable interpretation is necessary at the parsing level to deal with locating files that are included in the Snort configuration file. Snort configuration files are line based, with a minor exception of the specification of the output for a new output module, which has been ignored for the purpose of this system. A line starting with a '# ' character is considered to be a comment line. Otherwise, a non-blank line is required to start with a space delimited string that states the type of option to specify. For example, lines that include other files start with the `include` keyword.

According to the Snort manual [54], Snort rules start with the string "`alert`" followed by six mandatory space delimited fields followed by the option fields in parenthesis. The option fields are delimited by semicolons and use a single colon to separate the field name from the optional field argument. The parser shown in figure A.15 uses these assumptions to parse a snort rule. An example rule along with its parsed form is shown in figure A.16. As noted in section 4.7, for nine Snort rules this method of parsing did not work due to escaped semicolons. The nine rules for which this method of parsing did not work are listed in table A.1. The parsed Snort rules are encapsulated by an association list of fields and their arguments. For the preservation of the semantics of multiple `content` options, it is necessary that the order in which they appear in the `alert` Snort string be preserved. The parsed Snort rule encapsulation and related functions are detailed in figure A.17.

Snort has its own system of variables and substitution. Snort substitutions are textual substitution. Extensive documentation on programming language aspects of the use of these variables is not found in the manual [54]. All Snort variables are of global scope and are defined for the remainder of the configuration file. It is unspecified in [54] whether the same variable can be defined multiple times and if so, what value to take. Also unspecified in [54] is how to disambiguate multiple possible variable substitutions. For example, if a variable `A` is defined to be `D` and variable `AB` is defined to be `E`, then the value of `$AB` may be `DB` or `E`.

Figure A.18 details the encapsulation of Snort variables and substitutions in this system. One design for variable encapsulation would be to perform all of the substitutions during the reading of the Snort configuration file. This design choice has the disadvantage that the internal rules cannot change the values of the variables before writing the configuration file for the running system. Changing the values of the variables before writing the configuration file for the running system is a desirable property, especially as host and network information are commonly specified in variables. In the implemented prototype, the substitutions are delayed until either an attempt is made to compile the rule into an N-Code rule or a configuration file for Snort is written, at which point the variables and their values are written to the configuration file. For prototyping and debugging reasons, this implementation design choice of delaying substitution was ignored for variables whose value depends on the value of other variables. In the constructed prototype system, the substitutions

```
(define (parse-snort-rule s) ; s is a string
  (letrec
      ((split-line    ; space delimited for first 7 fields only
        (lambda (s)
          (let ((x (split-at-ref (split #\space s) 7)))
            (cons (car x) (unsplit #\space (cdr x))))))
       (parse-misc-field
        (lambda (s)
          (filter (lambda (x) (or (null? x) (eqv? (car x) "")))
                (map (lambda (f) (map remove-leading-spaces
                                      (split #\: f)))
                     (split #\;
                          (substring s 1 (− (string-length s) 1))))))) ; kill “()”
       (parse-line
        (lambda (s)
          (let ((l (split-line s)))
            (make-snort-rule (car l) (parse-misc-field (cdr l)))))))
    (parse-line s)))

(define (remove-leading-spaces s)
  (remove-leading (lambda (c) (eqv? c #\space)) s))

(define remove-leading ; inefficient.
  (lambda (r? s)
    (let loop ((x (string->list s)))
      (cond
       [(null? x) ""]
       [(r? (car x)) (loop (cdr x))]
       [else (list->string x)]))))
```

Figure A.15: Snort Rule Parsing

Table A.1: Snort Rules With Parsing Difficulties

SID	Problem	Snort Rule
326	\;	alert tcp any any -> 10.1.1.0/24 79 (msg:"sid 326 FINGER remote command \; execution attempt"; flow:to_server,established; content:"|3b|"; reference:cve,CVE-1999-0150; reference:bugtraq,974; reference:arachnids,379; classtype:attempted-user; sid:326; rev:5;)
975	\;	alert tcp any any -> 10.1.1.0/24 80 (msg:"sid 975 WEB-IIS .asp\; flow:to_server,established; uricontent:".asp|3a3a|$DATA"; nocase; reference:bugtraq,149; reference:url,support.microsoft.com/default.aspx?scid=kb\; EN-US\; q188806; reference:cve,CVE-1999-0278; reference:nessus,10362; classtype:web-application-attack; sid:975; rev:8;)
1321	Other	alert ip any any -> 10.1.1.0/24 any (msg:"sid 1321 BAD-TRAFFIC 0 ttl"; ttl:0; reference:url,www.isi.edu/in-notes/rfc1122.txt; reference:url,support.microsoft.com/default.aspx?scid=kb\; EN-US\; q138268; sid:1321; classtype:misc-activity; rev:6;)
1333	\;	alert tcp any any -> 10.1.1.0/24 80 (msg:"sid 1333 WEB-ATTACKS id command attempt"; flow:to_server,established; content:"\; id"; nocase; sid:1333; classtype:web-application-attack; rev:4;)
1565	\;	alert tcp any any -> 10.1.1.0/24 80 (msg:"sid 1565 WEB-CGI eshop.pl arbitrary commane execution attempt"; flow:to_server,established; uricontent:"/eshop.pl?seite=\; "; nocase; reference:cve,CAN-2001-1014; classtype:web-application-attack; sid:1565; rev:4;)
1815	\;	alert tcp any any -> 10.1.1.0/24 80 (msg:"sid 1815 WEB-PHP directory.php arbitrary command attempt"; flow:to_server,established; uricontent:"/directory.php"; content:"dir="; content:"\; "; reference:bugtraq,4278; reference:cve,CAN-2002-0434; classtype:misc-attack; sid:1815; rev:2;)

Table A.1: Continued

SID	Problem	Snort Rule
1865	\;	alert tcp any any -> 10.1.1.0/24 80 (msg:"sid 1865 WEB-CGI webdist.cgi arbitrary command attempt"; flow:to_server,established; uricontent:"/webdist.cgi"; nocase; content:"distloc=\; "; nocase; reference:bugtraq,374; reference:cve,CVE-1999-0039; reference:nessus,10299; classtype:web-application-attack; sid:1865; rev:1;)
1947	\;	alert tcp any any -> 10.1.1.0/24 8888 (msg:"sid 1947 WEB-MISC answerbook2 arbitrary command execution attempt"; flow:to_server,established; uricontent:"/ab2/"; content:"\; "; distance:1; classtype:web-application-attack; sid:1947; rev:2;)
2054	\;	alert tcp any any -> 10.1.1.0/24 80 (msg:"sid 2054 WEB-CGI enter_bug.cgi arbitrary command attempt"; flow:to_server,established; uricontent:"/enter_bug.cgi"; nocase; content:"who="; content:"\; "; distance:0; reference:cve,CAN-2002-0008; classtype:web-application-attack; sid:2054; rev:2;)

of variable references in variable value strings happen at the variable define time as may be seen in the code for *define-var!* in figure A.18.

Snort configuration files are line based. Each line can either be a comment, define an alert rule, define a variable, include another file, configure a preprocessor, configure an output module or define a class of alerts. A comment line is either completely whitespace or starts with the # character. The remaining lines start with the keyword for the type of operation. A listing of how this system handles the snort configuration files is provided in figure A.19. In the prototype code, recursive file includes are not checked against. The path to files with a relative path specification in their Snort `include` string is relative to the command line Snort configuration file, not the current working directory as is the default in UNIX. The code to fix this idiosyncrasy is listed in figure A.20.

A slight transform is preformed upon incoming Snort rules by the code in figure

Snort Rule:
alert tcp $HTTP_SERVERS $HTTP_PORTS -> $EXTERNAL_NET any (msg:"ATTACK-RESPONSES command completed"; content:"Command completed"; nocase; flow:from_server,established; classtype:bad-unknown; sid:494; rev:6;)"

Parsed Snort Rule:

```
(("alert" "alert")
 ("proto" "tcp")
 ("src-net" "$HTTP_SERVERS")
 ("src-port" "$HTTP_PORTS")
 ("dir" "->")
 ("dst-net" "$EXTERNAL_NET")
 ("dst-port" "any")
 ("msg" "\"ATTACK-RESPONSES command completed\"")
 ("content" "\"Command completed\"")
 ("nocase")
 ("flow" "from_server,established")
 ("classtype" "bad-unknown")
 ("sid" "494")
 ("rev" "6"))
```

Figure A.16: Example of Parsed Snort Rule

A.21. This transformation puts the SID into the message string so that the SID is printed when an alert is generated. The interface for the output module does not allow for the easy modification of the output module to facilitate printing the SID unless it is inside of the message field already. A regeneration of Snort files for use in this system along with the total transformation procedure for an existing Snort configuration file is shown in figure A.22.

A.1.4 Booting

In the first version of this prototype, each item necessary for booting was defined on a separate line. Then this list of lines was transformed into a tree for booting. Toward this end, the encapsulation detailed in figure A.23 was developed. Although this encapsulation has the advantage that non-tree based systems can have a configuration

```
(define snort-main-field-list
  '("alert" "proto" "src-net" "src-port" "dir" "dst-net" "dst-port"))

(define make-snort-rule
  (lambda (m o)
    (append (map list
                 snort-main-field-list
                 m)
            o)))

(define snort-rule-main
  (lambda (l)
    (map (lambda (k) (cadr (assoc k l))) snort-main-field-list)))

(define snort-rule-options
  (lambda (l)
    (filter (lambda (x) (member (car x) snort-main-field-list))
            l)))

(define (snort-rule->string r)
  (apply
   string-append
   (append (apply append
                  (map (lambda (s) (list s " "))
                       (snort-rule-main r)))
           '("(")
           (apply append
                  (map (lambda (l) (cons (car l)
                                         (if (null? (cdr l))
                                             '("; ")
                                             (list ":" (cadr l) "; "))))
                       (snort-rule-options r)))
           '(")"))))
```

Figure A.17: Snort Rule Encapsulation

```
(define (make-snort-var-env)
  (letrec
      ((v '())
       (get-var
        (lambda (x)
          (let ((y (assoc x v)))
            (if y (cdr y) #f))))
       (define-var!
         (lambda (var val)
           (let loop
               ((x val))
             (let ((y (snort-sub x))) ; snort manual claims all are resolvable.
               (if (equal? x y)
                   (if (assoc var v)
                       (set-cdr! (assoc var v) y)
                       (set! v (cons (cons var y) v)))
                   (loop y))))))
       (defined?
         (lambda (x) (assoc x v)))
       (var-strings
        (lambda ()
          (let loop
              ((l v)
               (r '()))
            (if (null? l)
                r
                (loop (cdr l)
                      (cons (string-append "var " (caar l) " " (cdar l)) r))))))
       (assoc-pred
        (lambda (p? l)
          (cond
           [(null? l) #f]
           [(p? (caar l)) (car l)]
           [else (assoc-pred p? (cdr l))]))))
```

Figure A.18: Snort Variable Encapsulation

```
    (superstring?
     (lambda (s x)
       (if (< (string-length s) (string-length x))
           #f
           (let ref-loop
                ((n (string-length x))
                 (i 0))
             (cond
              [(>= i n) #t]
              [(equal? (string-ref s i) (string-ref x i))
               (ref-loop n (+ i 1))]
              [else #f])))))
    (snort-var-sub
     (lambda (s)
       (let ((v (assoc-pred (lambda (key) (superstring? s key)) v)))
         (if v
             (string-append (cdr v) (substring s
                                               (string-length (car v))
                                               (string-length s)))
             (string-append "$" s))))) ; no variable found
    (snort-sub
     (lambda (s)
       (let ((x (split #\$ s)))
         (apply string-append
                (car x)
                (map snort-var-sub (cdr x)))))))
(lambda (cmd . l)
  (case cmd
    [(get get-var val) (apply get-var l)]
    [(define define-var! set-var! add-var!) (apply define-var! l)]
    [(defined?) (apply defined? l)]
    [(var-strings) (var-strings)]
    [(sub) (apply snort-sub l)])))))
```

Figure A.18: Continued

```
(define (parse-snort-rule-file f-in)
  (let ((env (make-snort-var-env)))
    (let file-loop
        ((f f-in))
      (let* ((in-p (open-input-file f))
             (in-buf (make-line-buffer in-p))
             (lines
              (let line-loop
                  ()
                (if (in-buf 'eof?)
                   '()
                   (let* ((l (in-buf 'get-line))
                          (s (split #\space l)))
                     (case-equal (car s)
                       [("alert")
                        (cons (snort-rule-transform l) (line-loop))]
                       [("var")
                        (env 'define-var! (cadr s) (caddr s))
                        (line-loop)]
                       [("#" "" " ") ; doesn't need to be "# "
                        (line-loop)]
                       [("include")
                        (append (cdr (assoc 'lines
                                                   (file-loop
                                                    (snort-filename-translate
                                                     (env 'sub (cadr s))))))
                                (line-loop))]
                       [("preprocessor" "output" "config")
                        (cons l (line-loop))]
                       [else
        ;(error 'snort-config-line-processor "Unknown line type ~a~n" (car s))
                             (line-loop)])))))))
        (list (cons 'env env) (cons 'lines lines))))))
```

Figure A.19: Snort Configuration File Parsing

```
(define (snort-filename-translate f) ; handle some ugly assumptions of Snort
  (if (member #\/ (string->list f))
      f
      (string-append snort-config-dir "/" f))) ; not necessarily working dir.
```

Figure A.20: Fixing Snort Relative Path Includes

```
(define (snort-rule-transform l)
  (let* ((x (parse-snort-rule l))
         (s (assoc "sid" (snort-rule-options x)))
         (m (assoc "msg" (snort-rule-options x))))
    (set-cdr! m (list (string-append
                       "\"sid "
                       (cadr s)
                       " "
                       (substring (cadr m) 1 (string-length (cadr m))))))
    (snort-rule->string x)))
```

Figure A.21: Snort Rule Transformation

specified, this representation is more difficult than necessary for an implementation restricted to trees. For the purpose of booting, the representation was transformed into a tree. Figure A.24 is an example of a system configuration using this initial implementation.

The second attempt at the language for system configuration simply specifies in a tree the hostnames as strings or as symbols whose values are the hostname as a string. If the node is a leaf node, then it is assumed to run the configured version of Snort on that node. If the node is an interior node, it is assumed to run the statistical node type with the **default-rules** rule database. The transformation for the tree is preformed by expanding the tree so that each node starts with a configuration type. Figure A.25 details the code to do the transformation along with an example. The example in figure A.24 is (`"fire" "fire" "agent"`) in the newer language. From a language development standpoint, further transformations may be useful. For example, it is

```
(define (print-snort-rule-file x f-out)
  (let* ((out-p (and f-out
                     (begin
                       (if (file-exists? f-out)
                           (delete-file f-out))
                       #t)
                     (open-output-file f-out)))
         (var-lines ((cdr (assoc 'env x)) 'var-strings))
         (l (append var-lines (cdr (assoc 'lines x)))))
    (fprintf out-p
             "~a"
             (apply string-append
                    (let loop
                        ((y l))
                      (if (null? y)
                          '()
                          (cons (car y)
                                (cons "\n"
                                      (loop (cdr y))))))))
    (close-output-port out-p)))

(define (existing-snort-conf->idsds f suffix)
  (let ((x (string-append f suffix)))
    (print-snort-rule-file (parse-snort-rule-file f) x)
    x))
```

Figure A.22: Writing Snort Configuration Files

common to run a signature based IDS on every interior node. It may be useful to specify a default host based IDS for the purpose of running by default on interior nodes.

As a shortcut for system development, the implemented prototype only used one type of configuration for statistical or signature nodes. For many smaller systems this implementation shortcoming is an irrelevant limitation. This shortcut saves the development of the aspects of the language dealing with specification of individual

```
(define get-node-name car)
(define get-node-type cadr)
(define get-node-host caddr)
(define get-node-depend cadddr)

(define (find-node-user l)
  (find-abstract (lambda (n) (eq? 'node-user (get-node-type n))) l))

(define (find-node-named name l)
  (find-abstract (lambda (n) (eq? name (get-node-name n))) l))
```

Figure A.23: Older System Configuration Encapsulation

```
((root node-user "fire" fire-stat)
 (fire-stat stat "fire" fire-sig agent-sig)
 (fire-sig snort "fire")
 (agent-sig snort "agent")))
```

Figure A.24: Example of Older Booting Configuration

signature or statistical based types. A complete version of the language needs a way of specifying different individual signature and statistical configurations.

For booting, Scheme nodes are loaded with the code for the entire system, then sent their configuration commands through standard input. Figure A.26 details the boot procedure for statistical nodes. The full current booting procedure is not as refined as possible and should be cleaned up by those wishing to advance this system further.

A.1.5 Configuring Statistical Rules

As noted in section 4.1, a database object was constructed to manage the organization of rules for interior nodes. This way of interior rule management did not prove as useful as expected. Although it provided a way of systematically organizing and retrieving rules, additional work is suggested so that operations other than simply applying the first rule retrieved or applying all of the rules retrieved can be accom-

```
(define (ids-config-short->long x) ; expands syntatic sugar
  (cond
   [(list? x)
    (cons 'stat
          (cons (car x)
                (map ids-config-short->long (cdr x))))]
   [(or (string? x) (symbol? x)) ; these symbols should evaluate to strings
    (list 'snort x)]
   [else (error 'ids-config-snort->long)]))

> (ids-config-short->long '(h1 h1 (sn1 sn2 sn3 sn4)))

(stat h1
      (snort h1)
      (stat sn1 (snort sn2) (snort sn3) (snort sn4)))
```

Figure A.25: Newer Booting Configuration

plished while maintaining reasonable complexity of rule management. This problem is significant for systems with many interior rules and is not adequately solved in the prototype system. Figure A.27 lists the rule database encapsulation. In this encapsulation, rules can either be global and apply to all of the alerts or they can be specific to a single alert type. Figure A.28 shows an example usage of the rule database. Figure A.29 shows how the database is used by the default Snort and Statistical nodes of the system.

A.1.6 Treatment of Time

As noted in section 4.3, time synchronization issues were avoided for the construction of the prototype. Internally, time is represented as the number of microseconds returned by the `gettimeofday` UNIX system call. For both performance and ease of system development reasons, this time representation was stored as a 64 bit integer rather than as a string representing the formatted time. Figure A.30 details the C system call code. Figure A.31 details the Scheme code for interfacing with the C helper functions. Other Scheme UNIX system library calls are detailed in figure A.32.

```
(define make-boot-stat
  (lambda (out-port cmd)
    (lambda (l-buf)
      (fprintf out-port "~s~n" '(set! *human-reader* #f))
      (grab-n-lines 3 l-buf)
      (printf "Sending scheme subprocess the command ~s~n" cmd)
      (fprintf out-port "~s~n" cmd))))

(define (grab-n-lines n x)
  (if (<= n 0)
      '()
      (let loop ()   ; TODO: line buffers should also do blocking IO
           (if (x 'line-ready?)
               (cons (x 'get-line) (grab-n-lines (- n 1) x))
               (loop)))))

 ;proc-msg-pipes is the main loop
(define (snort-node hostname)
  (proc-msg-pipes (list (make-stdin-msg-pipe) (make-snort-msg-pipe hostname))))

(define (config-stat-node sub-nodes)
  (proc-msg-pipes (cons (make-stdin-msg-pipe)
                        (map ids-config
                             sub-nodes))))
```

Figure A.26: Booting Scheme Based Nodes

A.1.7 Event Queues

Event queues were created as part of the prototype. The event queue was intended to allow for the computation of time averaging components. As noted in section 4.3, real time INBOUNDS can display the vector of SOM input values every 60 seconds during the duration of a connection. To model this displaying, it would be easy to put a callback in a time based event queue for 60 seconds later every time the vector of SOM input values is printed or the connection is opened. With a possibly large latency, the 60 second interval updated values for the vector of SOM input values can be computed upon arrival of a new packet. Unfortunately for this method of

```
(define (make-stat-rule-db)
  (letrec ((singles '())
           (globals '())
           (rule-put
            (lambda (rule l)
              (cond
               [(null? l) ]
               [(list? l) (map (lambda (x) (rule-put rule x)) l)]
               [(number? l)
                (let ((x (assoc l singles)))
                  (if x
                      (set-cdr! x (cons rule (cdr x)))
                      (let ((y (cons (cons l (list rule)) singles)))
                        (set! singles y))))]
               [(eq? l 'global)
                (set! globals (cons rule globals))]
               [else (error 'stat-rule-db "Unknown put command ~s" l)])))
           (rule-get
            (lambda (n)
              ; ordering will matter - this may change
              (append (let ((x (assoc n singles)))
                        (if x (cdr x) '()))
                      globals))))
    (lambda (cmd . l)
      (case cmd
        [(put)
         (apply rule-put l)]
        [(put-local)
         (apply rule-put l)]
        [(put-global)
         (rule-put (car l) 'global)]
        [(get)
         (apply rule-get l)]
        [(empty)
         (set! singles '())
         (set! globals '())]
        [else
         (error 'stat-rule-db "Unknown command")]))))
```

Figure A.27: Statistical Rule Database Object

```
(define *default-rules* (make-stat-rule-db))

(*default-rules* 'put-global (make-collector))

(define pass-rule
  (lambda (trigger arg)
    (if *human-reader*
        (printf "Pass-rule – ~d ~s~n" trigger arg)
        (printf "~s~n" (list 'stat-alert 'pass trigger arg)))))

(*default-rules* 'put-local
                 pass-rule
                 '(2000001 2000002 2000003 2000004 2000005))

> (*default-rules* 'get 123)

(#<procedure>)

> (*default-rules* 'get 2000001)

(#<procedure pass-rule> #<procedure>)
```

Figure A.28: Example Usage of Statistical Rule Database

computing these values without an event queue, the timeout of a TCP session must still be handled without inspecting new traffic. Hence, for at least the handling of TCP timeouts, a time based event queue is useful.

As the prototype was constructed, the time based event queue was not integrated into the lower levels of the system. It was determined that the current SOM module does not use the 60 second updates. Anomalies on partial connection information is an area for future work on the INBOUNDS project. Because the event queue allows for adding events to be executed in the same time period, it allows recursion, which would destroy the computational claims for the lower levels of the system. For these two reasons, the time based event queue was not integrated into the prototype. Figure A.33 details the event queue encapsulation. Figure A.34 details how to process the

```
(define make-ids-msg-eval
  (lambda (rule-db get-trigger)
    (lambda (msg)
      (let* ((trigger (get-trigger msg))
             (rules (and trigger (rule-db 'get trigger))))
        (if rules
 ; (begin (map (lambda (r) (r trigger msg)) rules))))))) ; do all the rules
              ((car rules) trigger msg)) rules)))) ; just do the first rule

(define make-snort-msg-eval
  (lambda (rule-db)
    (make-ids-msg-eval rule-db get-alert-sid)))

(define make-stat-msg-eval
  (lambda (rule-db)
    (make-ids-msg-eval rule-db
                       (lambda (stat-msg)
                         (and (pair? stat-msg)
                              (eq? (car stat-msg) 'stat-alert)
                              (cadr stat-msg))))))

(define *stat-msg-eval* (make-stat-msg-eval *default-rules*))
(define *ids-msg-eval* (make-snort-msg-eval *default-rules*))
```

Figure A.29: Usage of Rule Database in the System

event queue indefinitely. It is straightforward to change the process event queue code so that it has a message pipe interface, hence could be used by the nodes.

A.2 Resource Limited Structures

As noted in section 4.5, a resource limited list was implemented. A version of the implemented resource limited list is detailed in figure A.35. This version of the implementation quietly ignores attempts to add past the end of the list. An earlier version would print out an alert by the resource limited list encapsulation itself if the resource bounds were exceeded. It was intended to be useful to the user to know that the bounds were exceeded. Although the exceedence of bounds can be useful information, generating a separate alert for exceeding the bounds was later thought

```
#include <sys/time.h>
#include <errno.h>
#include <stdio.h>
#include <assert.h>


/* Rather than figure out how to return structs in Scheme,
 * use local state */
int gettimeofdayhelper (int n)
{
  static struct timeval x;

  assert(4 == sizeof(long));
  switch (n) {
  case 0:
    if (gettimeofday(&x, NULL) < 0) {
      perror(__FILE__);
      exit(-1);
    }
    return x.tv_sec;
    break;
  case 1:
    return x.tv_usec;
    break;
  default:
    exit(-1);
  }
}
```

Figure A.30: UNIX System Call Helper

```
 ; make C_lib_helper.so by gcc -fPIC -shared
(load-shared-object "/lib/libc.so.6") ; linux specific
(load-shared-object "./C_lib_helper.so")

(define C-gettimeofdayhelper
  (foreign-procedure "gettimeofdayhelper" (integer-32) integer-32))

(define (get-time-of-day) ; returns usec
  (let* ((sec (C-gettimeofdayhelper 0))
         (usec (C-gettimeofdayhelper 1)))
    (+ (* 1000000 sec) usec)))
```

Figure A.31: Scheme System Time

```
(define C-usleep
  (foreign-procedure "usleep" (unsigned-32) void)) ; may be different arg
                                                   ; length on sparc.

(define (sleep-usec x)
  (if (< 0 x)
      (C-usleep x)))

 ;The name collision doesn't seem to be a problem.
(define (kill signal pid)
  (define sys-signals
    '((SIGHUP 1)
      (SIGKILL 9)
      (SIGTERM 15)))
  (let* ((x (assoc signal sys-signals))
         (r (and x (C-kill pid (cadr x)))))
    (if (not x) ; ignore other errors
        (error 'kill "bad signal %s%n" signal))))

(define C-kill
  (foreign-procedure "kill" (unsigned-32 integer-32) integer-32))
```

Figure A.32: Assorted Scheme System Library Calls

```
(define make-event-queue
  (lambda ()
    (let ((q '(*head*)))
      (define get-time car)
      (define get-events cdr)
      (define set-events! set-cdr!)
      (define make-time-events cons)
      (define (get-first-chain)
        (cadr q))
      (define (remove-first-element!)
        (if (null? (cdr (get-events (get-first-chain))))
           (set-cdr! q (cddr q))
           (set-events! (get-first-chain)
                       (cdr (get-events (get-first-chain))))))
      (define (add-element! abs-time element)
        (let loop
            ((l q))
          (if (null? (cdr l))
             (set-cdr! l (list (make-time-events abs-time (list element))))
             (let* ((y (cadr l)) ; next chain
                   (x (get-time y)))
               (cond
                [(< abs-time x)
                 (set-cdr! l (cons (make-time-events abs-time (list element))
                                 (cdr l)))]
                [(equal? abs-time x)
                 (set-events! y (cons element (get-events y)))]
                [else (loop (cdr l))])))))
      (lambda (cmd . l)
        (case cmd
          [(get-first-element) ;peek
           (car (get-events (get-first-chain)))]
          [(get-first-time)
           (get-time (get-first-chain))]
          [(remove-first-element!) (remove-first-element!)]
          [(empty?) (null? (cdr q))]
```

Figure A.33: Event Queue

```
[(add-element!)
 (add-element!
  (case (car l)
     [(now) (get-time-of-day)]
     [(delta-usec) (+ (get-time-of-day) (cadr l))]
     [(abs) (cadr l)])
  (if (eqv? (car l) 'now)
       (cadr l)
       (caddr l)))]
[(dump) ; for debugging
 (pretty-print q)]
[else (error 'event-queue "Unknown operation ~s~n" cmd)]))))))
```

Figure A.33: Continued

```
(define process-event-queue ; assume events are functions w/ zero args
  (lambda (q)
    (if (not (q 'empty?))
        (let* ((x (q 'get-first-time)) ; order matters
                (y (get-time-of-day)))
          (if (<= x y)
              (let ((e (q 'get-first-element)))
                (q 'remove-first-element!)
                (e)) ; e can add elements
              (sleep-usec (− x y)))
          (process-event-queue q)))))
```

Figure A.34: Indefinitely Processing an Event Queue

to not be the best way to handle it. Instead, a predicate 'limit-ever-exceeded? with a corresponding count 'times-exceeded was added to the encapsulation. Using this predicate and count, the alert generation routines in the rule which uses the resource limited structure can handle the reporting of resource bounds exceedence. An example usage is shown in figure A.36.

A.3 Data Collection

A version of the data collector mentioned in section 4.4 is detailed in figure A.37. This collector is written to take output either from Snort processes or another collector as a subnode. When fed information from a collector as a subnode, this collector will use the counts seen by the subnode to increase the count of alerts seen. The code has been written directly in Scheme instead of the proposed computationally limited language. This code constitutes a test for verification that the semantics of the proposed language are sufficiently powerful. The output routine *print-number-list* detailed in figure A.38 will take time proportional to the length of the list, so to implement this with proper bounds on time and space, it is necessary to either have a reasonable number of alert types or to use a resource limited list for *which-rules*. As the current number of alerts is bounded by 3,000, the enforcement of a tighter bound on running time by further resource limiting of the *which-rules* list was not performed. An example output of this data collector from a ping flood is shown in figure A.39. If the ping flood were used to mask an attack, the new alert would show up as an additional alert type reported in the printing of *which-rules*. As the code is currently written, an event including a new alert type does not automatically produce an alert.

A.4 Self Organizing Map Inputs

To compute the inputs for the SOM, special Snort rules that alarmed on the various types of normal TCP traffic depending on the TCP flags were added to a Snort

```
 ; A resource limited list. To be fully proper, should not expose underlying
 ; list, but to be practical, it is useful to do so.
(define (make-limited-list limit . opts)
  (let* ((opt-l (apply a-list opts))
         (l (assoc-default 'initial-list opt-l '()))
         (n (assoc-default 'times-exceeded opt-l 0))
         (len (assoc-default 'internal-len opt-l (length l))) ; only use this internally
         (check-limit
          (lambda ()
            (if (< limit len)
                (let ((x (- len limit)))
                  (set! l (cdr-N x l))
                  (set! n (+ n x))
                  (set! len limit)))))
         (copy-limited-list  ; underlying list is not copied
          (lambda ()
            (make-limited-list limit 'initial-list l 'times-exceeded n 'internal-len len))))
    (check-limit)
    (lambda (cmd . args)
      (case cmd
        [(head list) l]
        [(length) len]
        [(times-exceeded) n]
        [(limit-ever-exceeded?) (not (zero? n))]
        [(limit) limit]
        [(set-list!)
         (set! l (car args))
         (set! len (length l))
         (check-limit)]
        [(set-limit!)
         (set! limit (car args))
         (check-limit)]
        [(add-front!)
         (set! l (append args l))
         (set! len (+ len (length args)))
         (check-limit)]
```

Figure A.35: Resource Limited List

```
[(pop-front!)
 (if (zero? len)
     (error 'limited-list "tried to pop on an empty list")
     (let ((x (car l)))
        (set! l (cdr l))
        (set! len (- len 1))
        x))]
[(add-front)  ; non-destructive
 (let ((x (copy-limited-list)))
   (apply x 'add-front! args)
   x)]
[(peek-front)
 (if (zero? len)
     (error 'limited-list "tried to peek at the front of an empty list")
     (car l))]
[else (error 'limited-list
              "Unknown command issued to a limited list: ~s~n"
              cmd)]))))
```

Figure A.35: Continued

configuration. Then the alarms from these rules were correlated at the statistical layer of the system to compute the input values for the SOM.

A separate FSM object to compute the SOM value was attached to each TCP connection. This FSM object is detailed in figure A.40. It is assumed that the direction of the TCP connection can be specified as an examinable input to the FSM, while the numerical values should not be examined by a FSM in this system. As an implementation shortcut, the criteria of not examining numerical values was not explicitly enforced. As there is very little explicit state in this numerical FSM, a diagram has not been provided. FSMs have many ways of being specified. The prototype used explicit FSM programming in Scheme because that was easiest FSM specification to implement.

Section 4.6 discusses the trade-offs in different strategies for dealing with TCP reconstruction. Non-integrated TCP reconstruction can be difficult to do while main-

```
(define x (make-limited-list 3))

(define print-info
  (lambda (x)
    (map (lambda (cmd) (printf "~s ~s~n" cmd (x cmd)))
         (list 'head 'length 'limit 'limit-ever-exceeded? 'times-exceeded))))

(define try
  (lambda (x . l)
    (let ((y (apply x l)))
      (print-info x)
      y)))

> (try x 'add-front! 5)

length 1
limit-ever-exceeded? #f
times-exceeded 0
limit 3
head (5)

> (try x 'add-front! 3 4 5)

length 3
limit-ever-exceeded? #t
times-exceeded 1
limit 3
head (4 5 5)

> (try x 'set-limit! 2)

length 2
limit-ever-exceeded? #t
times-exceeded 2
limit 2
head (5 5)
```

Figure A.36: Example Usage of Resource Limited List

```
> (try x ’pop-front!)

length 1
limit-ever-exceeded? #t
times-exceeded 2
limit 2
head (5)
5

> (define y (try x ’add-front 6))

length 1
limit-ever-exceeded? #t
times-exceeded 2
limit 2
head (5)

> (try y ’peek-front)

length 2
limit-ever-exceeded? #t
times-exceeded 2
limit 2
head (6 5)
6
```

Figure A.36: Continued

taining the bounds on running time of the FSM doing the reconstruction. Specifically, the TCP reconstruction code needs helper functions that handle the cases where the connection has not be created yet and where the connection sees a TCP FIN. For these cases, either the FSM or the helper function must examine the state stored in the connection hash table. In theory, this gives the FSM access to at least as many bits for computational storage as the size of the hash table. This would increase the bounds on running time by $2^{\text{size(hash table)}}$ which is unacceptable. There may be solutions to the problem of reconstructing TCP while preserving running time which involve preventing the examination of other parts of the data. Because an acceptable

```
 ; collector should be identity for first occurrence
(define (make-collector) ; all rules only act by side effects (print)
  (let ((x 0)
        (y 0)
        ;(full-rules? #f)
        (which-rules '()) ; or use a resource limited list
        ;(max-num-rules 3)
        ;(full-nodes? #f)
        ;(which-nodes? '())
        )
    (lambda (trigger arg)
      (if (eq? (car arg) 'stat-alert) ; or rewrite collector properly
         (set! arg (cddr arg)))
      (cond
       [(eq? trigger 'collector) ; collector is input
        (set! x (+ x (car arg)))
        (set! which-rules (union which-rules (cadr arg)))]
       [else
        (set! x (+ x 1))
        (if (not (memv trigger which-rules))
            (set! which-rules (cons trigger which-rules)))])
      (if (> x (* 2 y)) ; exp back off
         (begin
           (set! y x)
           (if *human-reader*
               (begin
                 (printf "Collected ~d alarms including rules " x)
                 (print-number-list which-rules)
                 (printf "~n"))
               (printf "~s~n" (list 'stat-alert 'collector x which-rules))))))))
```

Figure A.37: Alert Collector

```
(define (print-number-list l)
  (cond
   [(null? l) ]
   [(number? (car l))
    (printf " ~d" (car l))
    (print-number-list (cdr l))]
   [else (error 'print-number-list "Not a number ~s~n" (car l))]))
```

Figure A.38: Alert Collector Output Subroutine

```
> (do-one-snort)

Collected 1 alarms including rules  528
Collected 3 alarms including rules  528
Collected 7 alarms including rules  528
Collected 15 alarms including rules  528
Collected 31 alarms including rules  528
Collected 63 alarms including rules  528
Collected 127 alarms including rules  528
Collected 255 alarms including rules  528
Collected 511 alarms including rules  528
Collected 1023 alarms including rules  528
Collected 2047 alarms including rules  528

(*clean-up*)

[root@trotsky idsds]#
```

Figure A.39: Alert Collector During Ping Flood

solution can be obtained by integrating custom TCP reconstruction, these other possible solutions were not explored. The code listed in figure A.41 is how the prototype deals with TCP reconstruction. The TCP reconstruction code in the prototype could use additional work to make it into a reliable resource bounded portion of the system. Specifically, connection timeouts and mechanisms to deal with lost data could be added. These additions were not vital to the computation of SOM inputs, so were not done for the prototype.

```
 ; Another way to compute this (on the fly) would be to attach a rule
 ; processor to every rule instance, then have it do the partial
 ; compilation followed by returning the new rule to attach (or
 ; modifying the state of the rule). Need a new stat object for every
 ; connection.
(define (make-connection-state)
  (let
      ((one-fin #f)
       (size-src-dst 0)
       (size-dst-src 0)
       (n-src-dst 0) ; actually ”questions”
       (n-dst-src 0) ; ditto
       (idle-src-dst 0)
       (idle-dst-src 0)
       (dir 'begin)
       (ts-start #f)
       (p-ts #f))
    (lambda (cmd . l)
      (let ((do-packet
             (lambda (q-dir q-seg-len q-ts)
               (if (not ts-start)
                   (set! ts-start q-ts))
               (if (and (not (zero? q-seg-len))
                        (not (eq? dir q-dir))) ; dir change
                   (begin
                     (set! dir q-dir)
                     (if (eq? dir 'src-dst)
                         (begin
                           (set! n-src-dst (+ n-src-dst 1))
                           (set! idle-src-dst (+ idle-src-dst (- q-ts p-ts))))
                         (begin
                           (set! n-dst-src (+ n-dst-src 1))
                           (set! idle-dst-src (+ idle-dst-src (- q-ts p-ts)))))))
               (cond ; handle 'begin also
                [(eq? dir 'src-dst)
                 (set! size-src-dst (+ size-src-dst q-seg-len))]
                [(eq? dir 'dst-src)
                 (set! size-dst-src (+ size-dst-src q-seg-len))])
               (set! p-ts q-ts))))
```

Figure A.40: Computing SOM Input Values for a TCP Connection

```
(case cmd
  [(add-packet) (apply do-packet l)]
  [(compute-stats)
   (if (or (eq? dir 'begin) (not ts-start) (not p-ts))
       '(0 0 0 0 0 0)
       (let ((time (/ (− p-ts ts-start) 1000000.0)))
         (if (zero? time)
             '(0 0 0 0 0 0)
             (list
              (/ n-src-dst time) ; Inter
              (if (zero? n-src-dst) 0 (/ size-src-dst 1.0 n-src-dst))
              (if (zero? n-dst-src) 0 (/ size-dst-src 1.0 n-dst-src))
              (/ idle-src-dst 1000000.0 time)
              (/ idle-dst-src 1000000.0 time)
              time))))]
  [(seen-one-fin?)
   one-fin]
  [(seen-one-fin!) (set! one-fin #t)]))))))
```

Figure A.40: Continued

```
 ; Don't use this concurrently, locking is not implemented
(define (make-tcp-stats-obj conn-table)
  (letrec
      ((get-connection
        (lambda (alert)
          (map (lambda (f)
                  (get-alert-field f alert))
                '("src" "srcport" "dst" "dstport"))))
       (reverse-connection
        (lambda (x)
          (append (cddr x) (list (car x) (cadr x)))))
       (action-rule ; load this only for the special rules.
        (lambda (trigger alert)
          (let*
 ;standard TCP processing...
                ((c (get-connection alert))
                 (c-r (reverse-connection c))
                 (seg-length (get-alert-tcpseg-size alert))
                 (ts (string->number (get-alert-field "usectime" alert)))
                 (trigger-type
                  (cdr (assoc trigger '((2000001 . syn)
                                        (2000002 . syn-ack)
                                        (2000003 . rst)
                                        (2000004 . fin)
                                        (2000005 . no-flags)))))
                 (t1
                  (let ((x (conn-table 'get c)))
                    (if x
                       (cons 'src-dst x)
                       (let ((x (conn-table 'get c-r)))
                         (if x
                            (cons 'dst-src x)
                            #f)))))
                 (conn-dir (and t1 (car t1)))
                 (conn-info (and t1 (cdr t1)))
```

Figure A.41: TCP Connection Demultiplexing

```
             (finish-conn
              (lambda ()
                (let ((r (conn-info 'compute-stats)))
                  (conn-table 'remove c)
                  (if *human-reader*
                      (printf "SOM input ~s~n" r)
                      (printf "~s~n" (list 'som-input r)))))))
          (case trigger-type
            [(syn) #t] ; ignore or just count.
            [(syn-ack)
             (set! conn-info (make-connection-state))
             (conn-table 'add c conn-info)
             (conn-info 'add-packet 'src-dst seg-length ts)]
            [(rst fin no-flags)
             (if conn-info
                 (begin
                   (conn-info 'add-packet conn-dir seg-length ts)
                   (cond
                    [(eq? trigger-type 'rst)
                     (finish-conn)]
                    [(eq? trigger-type 'fin)
                     (if (conn-info 'seen-one-fin?)
                         (finish-conn)
                         (conn-info 'seen-one-fin!))])))]))))
(lambda (cmd)
  (case cmd
    [(rule) action-rule]))))
```

Figure A.41: Continued

B. Snort Language

Table B.1 is relevant to this work for multiple reasons. Table B.1 is an examination of how Snort is used in practice. The specification for the Snort language determined by these examples differs for some options from the specification of the Snort language as defined in the Snort manual [54]. Furthermore, the specification of the Snort language given in [54] is an ad-hoc approach. It would involve considerable effort to build a compiler from this ad-hoc language specification. In any case, the examples generated by this table were the justification for the choice of specifications for the Snort language interpreted by the compiler. Specifically, the determination of general language options shown in table 4.1 was determined by an exhaustive examination of common examples similar to table B.1. Table B.1 gives a rather comprehensive view of the usage of one network based IDS at a large reduction in data to comprehend.

Table B.1 lists almost all of the ways that the Snort options were used in the 1985 parseable stable rules for Snort. Technically, when an option is used multiple times in a single rule, only the first way of using the option was examined for table B.1 is counted. Examining only the first way of using the option rather than all of the ways saved both space in the table and time to create the table.

Snort options denoted with an asterisk have had their examples truncated to only the first example to save space in table B.1. This was done to the Snort options `content`, `uricontent`, `icode`, `itype`, and `offset`. For example, the `content` Snort option only shows one example, yet content appears with 1060 different arguments in the Snort rules. Similarly, `uricontent` appears with 664 different arguments in the stable Snort rules. `Offset` appears with 26 different integer arguments ranging from

0 to 300. `Icode` appears with all integer values from 0 to 15 inclusive. `Itype` appears with all integer values from 0 to 40. The examples for `itype` are more interesting than those for `icode`, but are still not included due to space considerations.

The following Snort options do not appear as elements in table B.1: `classtype`, `msg`, `rev`, `reference`, and `sid`. These options are intended for human rather than machine usage. In particular, they do not specify semantic objects that require compilation. A table of the base URL values referenced in the `reference` option is provided in table B.2.

The six mandatory Snort fields do not appear as elements in table B.1. These mandatory fields are `protocol`, `source address`, `source port`, `direction`, `destination address` and `destination port`. The description in the Snort manual [54] is accurate except for stating the `/etc/services` style strings are allowed for port specification. In any case, many of these fields are site configuration dependent.

Table B.1: Snort Options Used in Stable Rules Collection

Snort Option	Ways Used	Example Rule
ack	0	alert tcp $EXTERNAL_NET 10101 -> $HOME_NET any (msg:"SCAN myscan"; ttl:>220; ack:0; flags:S; reference:arachnids,439; classtype:attempted-recon; sid:613; rev:1;)
	101058054	alert tcp $EXTERNAL_NET 80 -> $HOME_NET 1054 (msg:"BACKDOOR ACKcmdC trojan scan"; seq:101058054; ack:101058054; flags:A,12; reference:arachnids,445; sid:106; classtype:misc-activity; rev:4;)
byte_jump	4,20,relative,align	alert tcp $EXTERNAL_NET any -> $HOME_NET 32771:34000 (msg:"RPC kcms_server directory traversal attempt"; flow:to_server,established; content:"|00 00 00 00|"; offset:8; depth:4; content:"|00 01 87 7D|"; offset:16; depth:4; byte_jump:4,20,relative,align; byte_jump:4,4,relative,align; content:"/../"; distance:0; reference:cve,CAN-2003-0027; reference:url,www.kb.cert.org/vuls/id/850785; classtype:misc-attack; sid:2007; rev:5;)
	4,4,relative,align	alert tcp $EXTERNAL_NET any -> $HOME_NET 111 (msg:"RPC portmap tooltalk request TCP"; flow:to_server,established; content:"|00 00 00 00|"; offset:8; depth:4; content:"|00 01 86 A0|"; offset:16; depth:4; content:"|00 00 00 03|"; distance:4; within:4; byte_jump:4,4,relative,align; byte_jump:4,4,relative,align; content:"|00 01 86 F3|"; within:4; reference:cve,CAN-2001-0717; reference:cve,CVE-1999-0003; reference:cve,CVE-1999-0687; reference:cve,CAN-1999-1075; reference:url,www.cert.org/advisories/CA-2001-05.html; classtype:rpc-portmap-decode; sid:1298; rev:10;)

Table B.1: Continued

<table>
<tr><th>Snort Option</th><th>Ways Used</th><th>Example Rule</th></tr>
<tr><td>byte_test</td><td>1,>,0,0,relative,string</td><td>alert tcp $EXTERNAL_NET any -> $HOME_NET 110 (msg:"POP3 DELE negative arguement attempt"; content:"DELE"; depth:4; nocase; content:"-"; distance:1; byte_test:1,>,0,0,relative,string; classtype:misc-attack; reference:bugtraq,7445; reference:bugtraq,6053; sid:2121; rev:1;)</td></tr>
<tr><td></td><td>1,>,6,2</td><td>alert udp $EXTERNAL_NET any -> $HOME_NET 67 (msg:"MISC bootp hardware address length overflow"; content:"|01|"; offset:0; depth:1; byte_test:1,>,6,2; reference:cve,CAN-1999-0798; classtype:misc-activity; sid:1939; rev:2;)</td></tr>
<tr><td></td><td>1,>,7,1</td><td>alert udp $EXTERNAL_NET any -> $HOME_NET 67 (msg:"MISC bootp invalid hardware type"; content:"|01|"; offset:0; depth:1; byte_test:1,>,7,1; reference:cve,CAN-1999-0798; classtype:misc-activity; sid:1940; rev:1;)</td></tr>
<tr><td></td><td>2,>,1024,0,relative,little</td><td>alert tcp $EXTERNAL_NET any -> $HOME_NET 139 (msg:"NETBIOS SMB trans2open buffer overflow attempt"; flow:to_server,established; content:"|00|"; offset:0; depth:1; content:"|ff|SMB|32|"; offset:4; depth:5; content:"|00 14|"; offset:60; depth:2; byte_test:2,>,1024,0,relative,little; reference:cve,CAN-2003-0201; reference:url,www.digitaldefense.net/labs/advisories/DDI-1013.txt; classtype:attempted-admin; sid:2103; rev:4;)</td></tr>
<tr><td></td><td>2,>,4000,0</td><td>alert tcp $EXTERNAL_NET any -> $HOME_NET 873 (msg:"MISC rsyncd overflow attempt"; flow:to_server; byte_test:2,>,4000,0; content:"|00 00|"; offset:2; depth:2; classtype:misc-activity; sid:2048; rev:1;)</td></tr>
<tr><td></td><td>4,>,100,0,relative</td><td>alert udp $EXTERNAL_NET any -> $HOME_NET any (msg:"RPC STATD UDP stat mon_name format string exploit attempt"; content:"|00 00 00 00|"; offset:4; depth:4; content:"|00 01 86 B8|"; offset:12; depth:4; content:"|00 00 00 01|"; distance:4; within:4; byte_jump:4,4,relative,align; byte_jump:4,4,relative,align; byte_test:4,>,100,0,relative; reference:cve,CVE-2000-0666; reference:bugtraq,1480; classtype:attempted-admin; sid:1913; rev:7;)</td></tr>
</table>

Table B.1: Continued

Snort Option	Ways Used	Example Rule
	4,>,1000,28,relative	alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"RPC CMSD TCP CMSD_INSERT buffer overflow attempt"; flow:to_server,established; content:"|00 00 00 00|"; offset:8; depth:4; content:"|00 01 86 E4|"; offset:16; depth:4; content:"|00 00 00 06|"; distance:4; within:4; byte_jump:4,4,relative,align; byte_jump:4,4,relative,align; byte_jump:4,0,relative,align; byte_test:4,>,1000,28,relative; reference:cve,CVE-1999-0696; reference:url,www.cert.org/advisories/CA-99-08-cmsd.html; classtype:misc-attack; sid:1909; rev:6;)
	4,>,1024,0,relative	alert udp $EXTERNAL_NET any -> $HOME_NET any (msg:"RPC CMSD UDP CMSD_CREATE buffer overflow attempt"; content:"|00 00 00 00|"; offset:4; depth:4; content:"|00 01 86 E4|"; offset:12; depth:4; content:"|00 00 00 15|"; distance:4; within:4; byte_jump:4,4,relative,align; byte_jump:4,4,relative,align; byte_test:4,>,1024,0,relative; reference:cve,CVE-1999-0696; reference:bugtraq,524; classtype:attempted-admin; sid:1907; rev:7;)
	4,>,1024,20,relative	alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"RPC snmpXdmi overflow attempt TCP"; flow:to_server,established; content:"|00 00 00 00|"; offset:8; depth:4; content:"|00 01 87 99|"; offset:16; depth:4; content:"|00 00 01 01|"; distance:4; within:4; byte_jump:4,4,relative,align; byte_jump:4,4,relative,align; byte_test:4,>,1024,20,relative; reference:bugtraq,2417; reference:cve,CAN-2001-0236; reference:url,www.cert.org/advisories/CA-2001-05.html; classtype:attempted-admin; sid:569; rev:9;)

Table B.1: Continued

Snort Option	Ways Used	Example Rule
	4,>,128,0,relative	alert udp $EXTERNAL_NET any -> $HOME_NET any (msg:"RPC RQUOTA getquota overflow attempt UDP"; content:"|00 00 00 00|"; offset:4; depth:4; content:"|00 01 86 AB|"; offset:12; depth:4; content:"|00 00 00 01|"; distance:4; within:4; byte_jump:4,4,relative,align; byte_jump:4,4,relative,align; byte_test:4,>,128,0,relative; reference:cve,CVE-1999-0974; reference:bugtraq,864; classtype:misc-attack; sid:1963; rev:6;)
	4,>,2048,12,relative	alert tcp $EXTERNAL_NET any -> $HOME_NET 111 (msg:"RPC portmap proxy integer overflow attempt TCP"; flow:to_server,established; content:"|00 00 00 00|"; offset:8; depth:4; content:"|00 01 86 A0 00|"; offset:16; depth:5; content:"|00 00 00 05|"; distance:3; within:4; byte_jump:4,4,relative,align; byte_jump:4,4,relative,align; byte_test:4,>,2048,12,relative; reference:cve,CAN-2003-0028; reference:bugtraq,7123; classtype:rpc-portmap-decode; sid:2093; rev:2;)
	4,>,512,0,relative	alert udp $EXTERNAL_NET any -> $HOME_NET 500: (msg:"RPC AMD UDP amqproc_mount plog overflow attempt"; content:"|00 00 00 00|"; offset:4; depth:4; content:"|00 04 93 F3|"; offset:12; depth:4; content:"|00 00 00 07|"; distance:4; within:4; byte_jump:4,4,relative,align; byte_jump:4,4,relative,align; byte_test:4,>,512,0,relative; reference:cve,CVE-1999-0704; reference:bugtraq,614; classtype:misc-attack; sid:1905; rev:5;)
	4,>,512,4,relative	alert udp $EXTERNAL_NET any -> $HOME_NET any (msg:"RPC sadmind UDP NETMGT_PROC_SERVICE CLIENT_DOMAIN overflow attempt"; content:"|00 00 00 00|"; offset:4; depth:4; content:"|00 01 87 88|"; offset:12; depth:4; content:"|00 00 00 01|"; distance:4; within:4; byte_jump:4,4,relative,align; byte_jump:4,4,relative,align; byte_jump:4,124,relative,align; byte_jump:4,20,relative,align; byte_test:4,>,512,4,relative; reference:cve,CVE-1999-0977; reference:bugtraq,866; classtype:attempted-admin; sid:1911; rev:6;)

Table B.1: Continued

<table>
<tr><th>Snort Option</th><th>Ways Used</th><th>Example Rule</th></tr>
<tr><td></td><td>4,>,64,0,relative</td><td>alert udp $EXTERNAL_NET any -> $HOME_NET any (msg:"RPC yppasswd old password overflow attempt UDP"; content:"|00 00 00 00|"; offset:4; depth:4; content:"|00 01 86 A9|"; offset:12; depth:4; content:"|00 00 00 01|"; distance:4; within:4; byte_jump:4,4,relative,align; byte_jump:4,4,relative,align; byte_test:4,>,64,0,relative; classtype:rpc-portmap-decode; sid:2027; rev:3;)</td></tr>
<tr><td></td><td>5,<,65537,0,relative,string</td><td>alert ip $HOME_NET any -> $EXTERNAL_NET any (msg:"ATTACK-RESPONSES id check returned userid"; content:"uid="; byte_test:5,<,65537,0,relative,string; content:"gid="; distance:1; within:15; byte_test:5,<,65537,0,relative,string; classtype:bad-unknown; sid:1882; rev:7;)</td></tr>
<tr><td></td><td>5,>,256,0,string,dec,relative</td><td>alert tcp $EXTERNAL_NET any -> $HOME_NET 143 (msg:"IMAP login literal buffer overflow attempt"; flow:established,to_server; content:" LOGIN "; content:" {"; distance:0; nocase; byte_test:5,>,256,0,string,dec,relative; reference:bugtraq,6298; classtype:misc-attack; sid:1993; rev:3;)</td></tr>
<tr><td>content*</td><td>" –use-compress-program"</td><td>alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:"FTP tar parameters"; flow:to_server,established; content:" –use-compress-program" ; nocase; reference:bugtraq,2240; reference:arachnids,134; reference:cve,CVE-1999-0202; classtype:bad-unknown; sid:362; rev:7;)</td></tr>
<tr><td>depth</td><td>1</td><td>alert tcp $EXTERNAL_NET 31790 -> $HOME_NET 31789 (msg:"BACKDOOR hack-a-tack attempt"; content:"A"; depth:1; reference:arachnids,314; flags:A+; classtype:attempted-recon; sid:614; rev:2;)</td></tr>
<tr><td></td><td>2</td><td>alert udp $EXTERNAL_NET any -> $HOME_NET 2140 (msg:"BACKDOOR DeepThroat 3.1 Connection attempt"; content:"00"; depth:2; classtype:misc-activity; sid:1980; rev:1;)</td></tr>
</table>

Table B.1: Continued

Snort Option	Ways Used	Example Rule
	3	alert tcp $EXTERNAL_NET any -> $HOME_NET 33270 (msg:"BACKDOOR trinity connection attempt"; flow:to_server,established; content:"|21 40 23|"; offset:0; depth:3; reference:nessus,10501; reference:cve,CAN-2000-0138; classtype:attempted-admin; sid:1843; rev:3;)
	4	alert tcp $HOME_NET any <> $EXTERNAL_NET 1863 (msg:"CHAT MSN message"; flow:established; content:"MSG "; depth:4; content:"Content-Type\; content:"text/plain"; distance:1; classtype:misc-activity; sid:540; rev:8;)
	5	alert tcp $EXTERNAL_NET any -> $HOME_NET 873 (msg:"MISC rsyncd module list access"; flow:to_server,established; content:"|23|list"; offset:0; depth:5; classtype:misc-activity; sid:2047; rev:1;)
	6	alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"OTHER-IDS SecureNetPro traffic"; content:"|00 67 00 01 00 03|"; offset:0; depth:6; flow:established; classtype:bad-unknown; sid:1629; rev:3;)
	7	alert tcp $EXTERNAL_NET any -> $HOME_NET 22 (msg:"EXPLOIT ssh CRC32 overflow"; flow:to_server,established; content:"|00 01 57 00 00 00 18|"; offset:0; depth:7; content:"|FF FF FF FF 00 00|"; offset:8; depth:14; reference:bugtraq,2347; reference:cve,CVE-2001-0144; classtype:shellcode-detect; sid:1327; rev:3;)
	8	alert tcp $HOME_NET 749 -> $EXTERNAL_NET any (msg:"ATTACK-RESPONSES successful kadmind buffer overflow attempt"; flow:established,from_server; content:"*GOBBLE*"; depth:8; reference:cve,CAN-2002-1235; reference:url,www.kb.cert.org/vuls/id/875073; classtype:successful-admin; sid:1900; rev:3;)

Table B.1: Continued

Snort Option	Ways Used	Example Rule
	9	alert udp $EXTERNAL_NET any -> $HOME_NET 1900 (msg:"SCAN UPnP service discover attempt"; content:"M-SEARCH "; offset:0; depth:9; content:"ssdp\; classtype:network-scan; sid:1917; rev:4;)
	10	alert tcp $EXTERNAL_NET any -> $HOME_NET 79 (msg:"FINGER cybercop query"; content:"|0A| "; flow:to_server,established; depth:10; reference:arachnids,132; reference:cve,CVE-1999-0612; classtype:attempted-recon; sid:331; rev:6;)
	11	alert tcp $EXTERNAL_NET any -> $HOME_NET 3389 (msg:"MISC MS Terminal server request (RDP)"; content:"|03 00 00 0b 06 E0 00 00 00 00 00|"; offset:0; depth:11; flow:to_server,established; reference:cve,CAN-2001-0540; classtype:protocol-command-decode; sid:1447; rev:4;)
	12	alert tcp $HTTP_SERVERS $HTTP_PORTS -> $EXTERNAL_NET any (msg:"ATTACK-RESPONSES 403 Forbidden"; flow:from_server,established; content:"HTTP/1.1 403"; depth:12; classtype:attempted-recon; sid:1201; rev:7;)
	13	alert tcp $HOME_NET 5631 -> $EXTERNAL_NET any (msg:"MISC Invalid PCAnywhere Login"; flow:from_server,established; content:"Invalid login"; offset:5; depth:13; classtype:unsuccessful-user; sid:511; rev:4;)
	14	alert udp $EXTERNAL_NET any -> $HOME_NET 35555 (msg:"BACKDOOR win-trin00 connection attempt"; content:"png []..Ks l44"; offset:0; depth:14; reference:cve,CAN-2000-0138; reference:nessus,10307; classtype:attempted-admin; sid:1853; rev:3;)
	15	alert tcp $EXTERNAL_NET any -> $HOME_NET 79 (msg:"BACKDOOR CDK"; flow:to_server,established; content:"ypi0ca"; nocase; depth:15; reference:arachnids,263; classtype:misc-activity; sid:185; rev:4;)

Table B.1: Continued

Snort Option	Ways Used	Example Rule
	16	alert tcp $EXTERNAL_NET 1024: -> $HOME_NET 2589 (msg:"BACKDOOR - Dagger_1.4.0_client_connect"; flow:to_server,established; content:"|0b 00 00 00 07 00 00 00|Connect"; depth:16; reference:url,www.tlsecurity.net/backdoor/Dagger.1.4.html; reference:arachnids,483; sid:104; classtype:misc-activity; rev:5;)
	17	alert tcp $HOME_NET 512 -> $EXTERNAL_NET any (msg:"ATTACK-RESPONSES rexec username too long response"; flow:from_server,established; content:"username too long"; offset:0; depth:17; classtype:unsuccessful-user; sid:2104; rev:2;)
	18	alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"WEB-MISC bad HTTP/1.1 request, Potentially worm attack"; flow:to_server,established; content:"GET / HTTP/1.1|0d 0a 0d 0a|"; offset:0; depth:18; reference:url,securityresponse.symantec.com/avcenter/security/Content/2002.09.13.html; classtype:web-application-activity; sid:1881; rev:4;)
	22	alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"BACKDOOR SubSeven 2.1 Gold server connection response"; flow:from_server,established; content:"connected. time/date\; depth:22; content:"version\; distance:1; classtype:misc-activity; sid:2100; rev:1;)
	32	alert tcp $HOME_NET 6789 -> $EXTERNAL_NET any (msg:"BACKDOOR Doly 2.0 access"; flow:established,from_server; content:"Wtzup Use"; depth:32; reference:arachnids,312; sid:119; classtype:misc-activity; rev:4;)
	36	alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"WEB-MISC PCCS mysql database admin tool access"; flow:to_server,established; content:"pccsmysqladm/incs/dbconnect.inc"; nocase; depth:36; reference:arachnids,300; classtype:web-application-attack; sid:509; rev:5;)

Table B.1: Continued

<table>
<tr><th>Snort Option</th><th>Ways Used</th><th>Example Rule</th></tr>
<tr><td></td><td>40</td><td>alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"INFO Outbound GNUTella client request"; flow:established; content:"GNUTELLA OK"; depth:40; classtype:misc-activity; sid:558; rev:5;)</td></tr>
<tr><td></td><td>50</td><td>alert udp $EXTERNAL_NET any -> $HOME_NET 9 (msg:"DOS Ascend Route"; content:"|4e 41 4d 45 4e 41 4d 45|"; offset:25; depth:50; reference:bugtraq,714; reference:cve,CVE-1999-0060; reference:arachnids,262; classtype:attempted-dos; sid:281; rev:2;)</td></tr>
<tr><td></td><td>70</td><td>alert tcp $HOME_NET 902 -> $EXTERNAL_NET any (msg:"OTHER-IDS ISS RealSecure 6 event collector connection attempt"; flow:from_server,established; content:"6ISS ECNRA Built-In Provider, Strong Encryption"; nocase; offset:30; depth:70; classtype:successful-recon-limited; sid:1760; rev:2;)</td></tr>
<tr><td></td><td>100</td><td>alert icmp $EXTERNAL_NET any -> $HOME_NET any (msg:"ICMP PING speedera"; content:"|3839 3a3b 3c3d 3e3f|"; depth:100; itype:8; sid:480; classtype:misc-activity; rev:2;)</td></tr>
<tr><td></td><td>128</td><td>alert ip $EXTERNAL_NET any -> $HOME_NET $SHELLCODE_PORTS (msg:"SHELLCODE x86 NOOP"; content:"|90 90 90 90 90 90 90 90 90 90 90 90 90 90|"; depth:128; reference:arachnids,181; classtype:shellcode-detect; sid:648; rev:5;)</td></tr>
<tr><td></td><td>750</td><td>alert tcp any 110 -> any any (msg:"Virus - Possible PrettyPark Trojan"; content:"\\CoolProgs\\"; offset:300; depth:750; reference:MCAFEE,10175; sid:772; classtype:misc-activity; rev:4;)</td></tr>
<tr><td>distance</td><td>0</td><td>alert tcp $HOME_NET any <> $EXTERNAL_NET 1863 (msg:"CHAT MSN file transfer request"; flow:established; content:"MSG "; depth:4; content:"Content-Type\; nocase; distance:0; content:"text/x-msmsgsinvite"; nocase; distance:0; content:"Application-Name\; content:"File Transfer"; nocase; distance:0; classtype:policy-violation; sid:1986; rev:1;)</td></tr>
</table>

Table B.1: Continued

Snort Option	Ways Used	Example Rule
	1	alert ip $HOME_NET any -> $EXTERNAL_NET any (msg:"ATTACK-RESPONSES id check returned userid"; content:"uid="; byte_test:5,<,65537,0,relative,string; content:"gid="; distance:1; within:15; byte_test:5,<,65537,0,relative,string; classtype:bad-unknown; sid:1882; rev:7;)
	3	alert tcp $EXTERNAL_NET any -> $HOME_NET 111 (msg:"RPC portmap proxy integer overflow attempt TCP"; flow:to_server,established; content:"|00 00 00 00|"; offset:8; depth:4; content:"|00 01 86 A0 00|"; offset:16; depth:5; content:"|00 00 00 05|"; distance:3; within:4; byte_jump:4,4,relative,align; byte_jump:4,4,relative,align; byte_test:4,>,2048,12,relative; reference:cve,CAN-2003-0028; reference:bugtraq,7123; classtype:rpc-portmap-decode; sid:2093; rev:2;)
	4	alert tcp $EXTERNAL_NET any -> $HOME_NET 111 (msg:"RPC portmap tooltalk request TCP"; flow:to_server,established; content:"|00 00 00 00|"; offset:8; depth:4; content:"|00 01 86 A0|"; offset:16; depth:4; content:"|00 00 00 03|"; distance:4; within:4; byte_jump:4,4,relative,align; byte_jump:4,4,relative,align; content:"|00 01 86 F3|"; within:4; reference:cve,CAN-2001-0717; reference:cve,CVE-1999-0003; reference:cve,CVE-1999-0687; reference:cve,CAN-1999-1075; reference:url,www.cert.org/advisories/CA-2001-05.html; classtype:rpc-portmap-decode; sid:1298; rev:10;)

Table B.1: Continued

<table>
<tr><th>Snort Option</th><th>Ways Used</th><th>Example Rule</th></tr>
<tr><td></td><td>13</td><td>alert udp $HOME_NET 500 -> $EXTERNAL_NET 500 (msg:"MISC isakmp login failed"; content:"|10 05|"; offset:17; depth:2; content:"|00 00 00 01 01 00 00 18|"; distance:13; within:8; classtype:misc-activity; sid:2043; rev:1;)</td></tr>
<tr><td></td><td>33</td><td>alert tcp $EXTERNAL_NET any -> $HOME_NET 512 (msg:"RSERVICES rexec password overflow attempt"; content:"|00|"; content:"|00|"; distance:33; content:"|00|"; distance:0; classtype:attempted-admin; sid:2114; rev:2;)</td></tr>
<tr><td></td><td>240</td><td>alert udp $EXTERNAL_NET any -> $HOME_NET 67 (msg:"MISC bootp hostname format string attempt"; content:"|01|"; offset:0; depth:1; content:"|0C|"; distance:240; content:"%"; distance:0; content:"%"; distance:1; within:8; content:"%"; distance:1; within:8; reference:bugtraq,4701; classtype:misc-attack; sid:2039; rev:1;)</td></tr>
<tr><td>dsize</td><td>0</td><td>alert udp $EXTERNAL_NET any -> $HOME_NET 161 (msg:"DOS Bay/Nortel Nautica Marlin"; dsize:0; reference:bugtraq,1009; reference:cve,CVE-2000-0221; classtype:attempted-dos; sid:279; rev:2;)</td></tr>
<tr><td></td><td>1</td><td>alert tcp $EXTERNAL_NET any -> $HOME_NET 6789:6790 (msg:"DOS DB2 dos attempt"; flow:to_server,established; dsize:1; classtype:denial-of-service; sid:1641; rev:4;)</td></tr>
<tr><td></td><td>10</td><td>alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:"FTP large PWD command"; flow:to_server,established; content:"PWD"; nocase; dsize:10; classtype:protocol-command-decode; sid:1624; rev:3;)</td></tr>
<tr><td></td><td>11</td><td>alert tcp $EXTERNAL_NET any -> $HOME_NET 79 (msg:"FINGER cybercop redirection"; flow:to_server,established; content:"@localhost|0A|"; dsize:11; reference:arachnids,11; classtype:attempted-recon; sid:329; rev:6;)</td></tr>
</table>

Table B.1: Continued

Snort Option	Ways Used	Example Rule
	20	alert icmp $EXTERNAL_NET any -> $HOME_NET any (msg:"ICMP Nemesis v1.1 Echo"; dsize:20; itype:8; icmp_id:0; icmp_seq:0; content:"|0000000000000000000000000000000000000000|"; reference:arachnids,449; classtype:attempted-recon; sid:467; rev:1;)
	4	alert icmp $EXTERNAL_NET any -> $HOME_NET any (msg:"ICMP Broadscan Smurf Scanner"; itype:8; icmp_id:0; icmp_seq:0; dsize:4; classtype:attempted-recon; sid:478; rev:1;)
	408	alert ip $EXTERNAL_NET any -> $HOME_NET any (msg:"DOS Jolt attack"; fragbits:M; dsize:408; reference:cve,CAN-1999-0345; classtype:attempted-dos; sid:268; rev:2;)
	8	alert icmp $EXTERNAL_NET any -> $HOME_NET any (msg:"ICMP PING LINUX/*BSD"; dsize:8; itype:8; id:13170; reference:arachnids,447; sid:375; classtype:misc-activity; rev:4;)
	<25	alert ip $EXTERNAL_NET any -> $HOME_NET any (msg:"MISC Tiny Fragments"; fragbits:M; dsize:< 25; classtype:bad-unknown; sid:522; rev:1;)
	<5	alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"WEB-MISC whisker tab splice attack"; dsize:<5; flow:to_server,established; content:"|09|"; reference:arachnids,415; classtype:attempted-recon; reference:url,www.wiretrip.net/rfp/pages/whitepapers/whiskerids.html; sid:1087; rev:6;)
	>1	alert icmp 255.255.255.0/24 any -> $HOME_NET any (msg:"BACKDOOR SIGNATURE - Q ICMP"; itype:0; dsize:>1; reference:arachnids,202; sid:183; classtype:misc-activity; rev:3;)

Table B.1: Continued

Snort Option	Ways Used	Example Rule
	>100	alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:"FTP command overflow attempt"; flow:to_server,established,no_stream; dsize:>100; reference:bugtraq,4638; classtype:protocol-command-decode; sid:1748; rev:4;)
	>1000	alert tcp $EXTERNAL_NET any -> $HOME_NET 8080 (msg:"EXPLOIT delegate proxy overflow"; flow:to_server,established; content:"whois|3a|//"; nocase; dsize:>1000; reference:arachnids,267; classtype:attempted-admin; sid:305; reference:bugtraq,808; reference:cve,CVE-2000-0165; rev:5;)
	>1023	alert tcp $EXTERNAL_NET any -> $HOME_NET 3372 (msg:"DOS MS-DTC attempt"; flow:to_server,established; dsize:>1023; reference:bugtraq,4006; classtype:attempted-dos; sid:1408; rev:5;)
	>1092	alert tcp $EXTERNAL_NET any -> $HOME_NET 143 (msg:"IMAP EXPLOIT partial body overflow attempt"; flow:to_server,established; content:" x PARTIAL 1 BODY["; dsize:>1092; reference:bugtraq,4713; classtype:misc-attack; sid:1780; rev:5;)
	>120	alert tcp $EXTERNAL_NET any -> $SMTP_SERVERS 25 (msg:"VIRUS Klez Incoming"; flow:to_server,established; dsize:>120; content:"MIME"; content:"VGhpcyBwcm9"; classtype:misc-activity; sid:1800; rev:3;)
	>128	alert udp $EXTERNAL_NET any -> $HOME_NET 123 (msg:"EXPLOIT ntpdx overflow attempt"; dsize:>128; reference:arachnids,492; reference:bugtraq,2540; classtype:attempted-admin; sid:312; rev:2;)
	>1445	alert tcp $EXTERNAL_NET any -> $HOME_NET 617 (msg:"DOS arkiea backup"; flow:to_server,established; dsize:>1445; reference:bugtraq,662; reference:cve,CVE-1999-0788; reference:arachnids,261; classtype:attempted-dos; sid:282; rev:4;)

Table B.1: Continued

Snort Option	Ways Used	Example Rule
	>200	alert tcp $EXTERNAL_NET any -> $TELNET_SERVERS 23 (msg:"TELNET bsd exploit client finishing"; flow:to_client,established; dsize:>200; content:"|FF F6 FF F6 FF FB 08 FF F6|"; offset:200; depth:50; classtype:successful-admin; sid:1253; reference:bugtraq,3064; reference:cve,CAN-2001-0554; rev:7;)
	>258	alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"WEB-FRONTPAGE rad overflow attempt"; uricontent:"/fp30reg.dll"; nocase; dsize:>258; flow:to_server,established; classtype:web-application-attack; reference:arachnids,555; reference:bugtraq,2906; reference:cve,CAN-2001-0341; reference:url,www.microsoft.com/technet/security/bulletin/MS01-035.asp; sid:1246; rev:8;)
	>259	alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"WEB-FRONTPAGE rad overflow attempt"; uricontent:"/fp4areg.dll"; nocase; dsize:>259; flow:to_server,established; reference:cve,CAN-2001-0341; reference:bugtraq,2906; classtype:web-application-attack; sid:1247; rev:7;)
	>500	alert tcp $EXTERNAL_NET any -> $HOME_NET 32000 (msg:"MISC Xtramail Username overflow attempt"; flow:to_server,established; dsize:>500; content:"Username\; nocase; reference:cve,CAN-1999-1511; reference:bugtraq,791; classtype:attempted-admin; sid:1636; rev:3;)
	>512	alert tcp $EXTERNAL_NET any -> $HOME_NET 119 (msg:"NNTP Cassandra Overflow"; flow:to_server,established; content:"AUTHINFO USER"; nocase; dsize:>512; depth:16; reference:cve,CAN-2000-0341; reference:arachnids,274; classtype:attempted-user; sid:291; rev:6;)

Table B.1: Continued

Snort Option	Ways Used	Example Rule
	>6	alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"BAD-TRAFFIC data in TCP SYN packet"; flags:S,12; dsize:>6; reference:url,www.cert.org/incident_notes/IN-99-07.html; sid:526; classtype:misc-activity; rev:6;)
	>720	alert tcp $EXTERNAL_NET any -> $HOME_NET 32772:34000 (msg:"EXPLOIT cachefsd buffer overflow attempt"; flow:to_server,established; dsize:>720; content:"|00 01 87 86 00 00 00 01 00 00 00 05|"; classtype:misc-attack; reference:cve,CAN-2002-0084; reference:bugtraq,4631; sid:1751; rev:3;)
	>800	alert icmp $EXTERNAL_NET any -> $HOME_NET any (msg:"ICMP Large ICMP Packet"; dsize:>800; reference:arachnids,246; classtype:bad-unknown; sid:499; rev:3;)
	>999	alert tcp $EXTERNAL_NET any -> $HOME_NET 32771:34000 (msg:"RPC EXPLOIT ttdbserv solaris overflow"; content:"|C0 22 3F FC A2 02 20 09 C0 2C 7F FF E2 22 3F F4|"; flow:to_server,established; dsize:>999; reference:url,www.cert.org/advisories/CA-2001-27.html; reference:bugtraq,122; reference:cve,CVE-1999-0003; reference:arachnids,242; classtype:attempted-admin; sid:570; rev:6;)
flags	0	alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"SCAN NULL"; flags:0; seq:0; ack:0; reference:arachnids,4; classtype:attempted-recon; sid:623; rev:1;)
	A	alert tcp any any -> any 139 (msg:"Virus - Possible QAZ Worm Infection"; flags:A; content:"|71 61 7a 77 73 78 2e 68 73 71|"; reference:MCAFEE,98775; sid:732; classtype:misc-activity; rev:3;)

Table B.1: Continued

Snort Option	Ways Used	Example Rule
	A+	alert tcp 255.255.255.0/24 any -> $HOME_NET any (msg:"BACKDOOR Q access"; flags:A+; dsize:>1; reference:arachnids,203; sid:184; classtype:misc-activity; rev:3;)
	A,12	alert tcp $EXTERNAL_NET 80 -> $HOME_NET 1054 (msg:"BACKDOOR ACKcmdC trojan scan"; seq:101058054; ack:101058054; flags:A,12; reference:arachnids,445; sid:106; classtype:misc-activity; rev:4;)
	F,12	alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"SCAN FIN"; flags:F,12; reference:arachnids,27; classtype:attempted-recon; sid:621; rev:2;)
	FPU,12	alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"SCAN nmap XMAS"; flags:FPU,12; reference:arachnids,30; classtype:attempted-recon; sid:1228; rev:2;)
	PA	alert tcp any any -> any 25 (msg:"Virus - Successful eurocalculator execution"; flags:PA; content:"funguscrack@hotmail.com"; nocase; sid:736; classtype:misc-activity; rev:4;)
	PA12	alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"SCAN cybercop os PA12 attempt"; content:"AAAAAAAAAAAAAAAA"; depth:16; flags:PA12; reference:arachnids,149; classtype:attempted-recon; sid:626; rev:2;)
	S	alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"DOS Land attack"; id:3868; seq:3868; flags:S; reference:cve,CVE-1999-0016; classtype:attempted-dos; sid:269; rev:3;)
	S+	alert tcp any any -> [232.0.0.0/8,233.0.0.0/8,239.0.0.0/8] any (msg:"BAD-TRAFFIC syn to multicast address"; flags:S+; classtype:bad-unknown; sid:1431; rev:5;)

Table B.1: Continued

Snort Option	Ways Used	Example Rule
	S,12	alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"BAD-TRAFFIC data in TCP SYN packet"; flags:S,12; dsize:>6; reference:url,www.cert.org/incident_notes/IN-99-07.html; sid:526; classtype:misc-activity; rev:6;)
	SA,12	alert tcp $HOME_NET 5714 -> $EXTERNAL_NET any (msg:"BACKDOOR WinCrash 1.0 Server Active" ; flags:SA,12; content:"|B4 B4|"; reference:arachnids,36; sid:163; classtype:misc-activity; rev:4;)
	SF	alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"SCAN synscan portscan"; id:39426; flags:SF; reference:arachnids,441; classtype:attempted-recon; sid:630; rev:1;)
	SF,12	alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"SCAN SYN FIN"; flags:SF,12; reference:arachnids,198; classtype:attempted-recon; sid:624; rev:2;)
	SF12	alert tcp $EXTERNAL_NET any -> $HOME_NET 80 (msg:"SCAN cybercop os probe"; flags:SF12; dsize:0; reference:arachnids,146; classtype:attempted-recon; sid:619; rev:1;)
	SFP	alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"SCAN cybercop os probe"; content:"AAAAAAAAAAAAAAAA"; flags:SFP; ack:0; depth:16; reference:arachnids,145; classtype:attempted-recon; sid:1133; rev:6;)
	SFPU	alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"SCAN nmap fingerprint attempt"; flags:SFPU; reference:arachnids,05; classtype:attempted-recon; sid:629; rev:1;)
	SFU12	alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"SCAN cybercop os SFU12 probe"; content:"AAAAAAAAAAAAAAAA"; depth:16; flags:SFU12; ack:0; reference:arachnids,150; classtype:attempted-recon; sid:627; rev:2;)

Table B.1: Continued

Snort Option	Ways Used	Example Rule
	SRAFPU,12	alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"SCAN XMAS"; flags:SRAFPU,12; reference:arachnids,144; classtype:attempted-recon; sid:625; rev:2;)
	U+	alert tcp $EXTERNAL_NET any -> $HOME_NET 135:139 (msg:"DOS Winnuke attack"; flags:U+; reference:bugtraq,2010; reference:cve,CVE-1999-0153; classtype:attempted-dos; sid:1257; rev:4;)
flow	established	alert tcp $HOME_NET any <> $EXTERNAL_NET 1863 (msg:"CHAT MSN message"; flow:established; content:"MSG "; depth:4; content:"Content-Type\; content:"text/plain"; distance:1; classtype:misc-activity; sid:540; rev:8;)
	established,from_server	alert tcp $HOME_NET 749 -> $EXTERNAL_NET any (msg:"ATTACK-RESPONSES successful kadmind buffer overflow attempt"; flow:established,from_server; content:"*GOBBLE*"; depth:8; reference:cve,CAN-2002-1235; reference:url,www.kb.cert.org/vuls/id/875073; classtype:successful-admin; sid:1900; rev:3;)
	established,to_client	alert tcp $EXTERNAL_NET 80 -> $HOME_NET any (msg:"CHAT ICQ forced user addition"; flow:established,to_client; content:"Content-Type\; content:"[ICQ User]"; reference:bugtraq,3226; reference:cve,CAN-2001-1305; classtype:misc-activity; sid:1832; rev:3;)
	established,to_server	alert tcp $EXTERNAL_NET any -> $HOME_NET 27665 (msg:"DDOS Trin00\; flow:established,to_server; content:"betaalmostdone"; reference:arachnids,197; classtype:attempted-dos; sid:233; rev:2;)
	from_client,established	alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:"INFO FTP No Password"; content:"PASS"; nocase; offset:0; depth:4; content:"|0a|"; within:3; reference:arachnids,322; flow:from_client,established; classtype:unknown; sid:489; rev:5;)

Table B.1: Continued

Snort Option	Ways Used	Example Rule
	from_server,established	alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"ATTACK-RESPONSES directory listing"; content:"Volume Serial Number"; flow:from_server,established; classtype:bad-unknown; sid:1292; rev:7;)
	to_Server,established	alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:"POLICY FTP 'MKD ' possible warez site"; flow:to_Server,established; content:"MKD "; nocase; depth:5; classtype:misc-activity; sid:547; rev:5;)
	to_client	alert tcp $AIM_SERVERS any -> $HOME_NET any (msg:"CHAT AIM receive message"; flow:to_client; content:"|2a 02|"; offset:0; depth:2; content:"|00 04 00 07|"; offset:6; depth:4; classtype:policy-violation; sid:1633; rev:4;)
	to_client,established	alert tcp $EXTERNAL_NET 6666:7000 -> $HOME_NET any (msg:"CHAT IRC dns response"; flow:to_client,established; content:"\; offset:0; content:" 302 "; content:"=+"; classtype:misc-activity; sid:1790; rev:2;)
	to_server	alert tcp $EXTERNAL_NET any -> $HOME_NET 873 (msg:"MISC rsyncd overflow attempt"; flow:to_server; byte_test:2,>,4000,0; content:"|00 00|"; offset:2; depth:2; classtype:misc-activity; sid:2048; rev:1;)
	to_server,established	alert tcp $EXTERNAL_NET 27374 -> $HOME_NET any (msg:"BACKDOOR subseven 22"; flow:to_server,established; content:"|0d0a5b52504c5d3030320d0a|"; reference:arachnids,485; reference:url,www.hackfix.org/subseven/; classtype:misc-activity; sid:103; rev:5;)
	to_server,established,no_stream	alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:"FTP USER overflow attempt"; flow:to_server,established,no_stream; content:"USER "; nocase; content:!"|0a|"; within:100; reference:bugtraq,4638; reference:cve,CAN-2000-0479; reference:cve,CAN-2000-0656; reference:cve,CAN-2000-1035; reference:cve,CAN-2000-1194; reference:cve,CAN-2001-0794; reference:cve,CAN-2001-0826; reference:cve,CAN-2002-0126; reference:cve,CVE-2000-0943; classtype:attempted-admin; sid:1734; rev:7;)

Table B.1: Continued

Snort Option	Ways Used	Example Rule
fragbits	M	alert ip $EXTERNAL_NET any -> $HOME_NET any (msg:"DOS Jolt attack"; fragbits:M; dsize:408; reference:cve,CAN-1999-0345; classtype:attempted-dos; sid:268; rev:2;)
	M+	alert ip $EXTERNAL_NET any -> $HOME_NET any (msg:"DOS IGMP dos attack"; content:"|02 00|"; depth:2; ip_proto:2; fragbits:M+; reference:cve,CVE-1999-0918; classtype:attempted-dos; sid:272; rev:2;)
	MD	alert ip $EXTERNAL_NET any -> $HOME_NET any (msg:"BAD-TRAFFIC bad frag bits"; fragbits:MD; sid:1322; classtype:misc-activity; rev:5;)
	R	alert ip $EXTERNAL_NET any -> $HOME_NET any (msg:"BAD-TRAFFIC ip reserved bit set"; fragbits:R; sid:523; classtype:misc-activity; rev:4;)
icmp_id	0	alert icmp $EXTERNAL_NET any -> $HOME_NET any (msg:"DDOS tfn2k icmp possible communication"; itype:0; icmp_id:0; content:"AAAAAAAAAA"; reference:arachnids,425; classtype:attempted-dos; sid:222; rev:1;)
	123	alert icmp $HOME_NET any -> $EXTERNAL_NET any (msg:"DDOS TFN server response"; itype:0; icmp_id:123; icmp_seq:0; content:"shell bound to port"; reference:arachnids,182; classtype:attempted-dos; sid:238; rev:4;)
	456	alert icmp $EXTERNAL_NET any -> $HOME_NET any (msg:"DDOS TFN client command BE"; itype:0; icmp_id:456; icmp_seq:0; reference:arachnids,184; classtype:attempted-dos; sid:228; rev:1;)
	666	alert icmp 3.3.3.3/32 any -> $EXTERNAL_NET any (msg:"DDOS Stacheldraht server spoof"; itype:0; icmp_id:666; reference:arachnids,193; classtype:attempted-dos; sid:224; rev:2;)
	667	alert icmp $HOME_NET any -> $EXTERNAL_NET any (msg:"DDOS Stacheldraht server response"; content:"ficken"; itype:0; icmp_id:667; reference:arachnids,191; classtype:attempted-dos; sid:226; rev:3;)

Table B.1: Continued

Snort Option	Ways Used	Example Rule
	668	alert icmp $EXTERNAL_NET any -> $HOME_NET any (msg:"DDOS Stacheldraht client check gag"; content:"gesundheit!"; itype:0; icmp_id:668; reference:arachnids,194; classtype:attempted-dos; sid:236; rev:3;)
	669	alert icmp $HOME_NET any -> $EXTERNAL_NET any (msg:"DDOS Stacheldraht gag server response"; content:"sicken"; itype:0; icmp_id:669; reference:arachnids,195; classtype:attempted-dos; sid:225; rev:3;)
	1000	alert icmp $EXTERNAL_NET any -> $HOME_NET any (msg:"DDOS Stacheldraht client spoofworks"; content:"spoofworks"; itype:0; icmp_id:1000; reference:arachnids,192; classtype:attempted-dos; sid:227; rev:3;)
	6666	alert icmp $EXTERNAL_NET any <> $HOME_NET any (msg:"DDOS Stacheldraht agent->handler (skillz)"; content:"skillz"; itype:0; icmp_id:6666; reference:url,staff.washington.edu/dittrich/misc/stacheldraht.analysis; classtype:attempted-dos; sid:1855; rev:2;)
	6667	alert icmp $EXTERNAL_NET any <> $HOME_NET any (msg:"DDOS Stacheldraht handler->agent (ficken)"; content:"ficken"; itype:0; icmp_id:6667; reference:url,staff.washington.edu/dittrich/misc/stacheldraht.analysis; classtype:attempted-dos; sid:1856; rev:2;)
	9015	alert icmp $EXTERNAL_NET any <> $HOME_NET any (msg:"DDOS Stacheldraht handler->agent (niggahbitch)"; content:"niggahbitch"; itype:0; icmp_id:9015; reference:url,staff.washington.edu/dittrich/misc/stacheldraht.analysis; classtype:attempted-dos; sid:1854; rev:2;)

Table B.1: Continued

Snort Option	Ways Used	Example Rule
	51201	alert icmp $EXTERNAL_NET any -> $HOME_NET any (msg:"DDOS - TFN client command LE"; itype:0; icmp_id:51201; icmp_seq:0; reference:arachnids,183; classtype:attempted-dos; sid:251; rev:1;)
icmp_seq	0	alert icmp $EXTERNAL_NET any -> $HOME_NET any (msg:"DDOS TFN client command BE"; itype:0; icmp_id:456; icmp_seq:0; reference:arachnids,184; classtype:attempted-dos; sid:228; rev:1;)
icode*	0	alert icmp $EXTERNAL_NET any -> $HOME_NET any (msg:"ICMP PING"; itype:8; icode:0; sid:384; classtype:misc-activity; rev:4;)
id	242	alert udp $EXTERNAL_NET any -> $HOME_NET any (msg:"DOS Teardrop attack"; id:242; fragbits:M; reference:cve,CAN-1999-0015; reference:url,www.cert.org/advisories/CA-1997-28.html; reference:bugtraq,124; classtype:attempted-dos; sid:270; rev:2;)
	413	alert tcp $EXTERNAL_NET any <> $HOME_NET any (msg:"DOS NAPTHA"; flags:S; seq:6060842; id:413; reference:cve,CAN-2000-1039; reference:url,www.microsoft.com/technet/security/bulletin/MS00-091.asp; reference:url,www.cert.org/advisories/CA-2000-21.html; reference:url,razor.bindview.com/publish/advisories/adv_NAPTHA.html; reference:bugtraq,2022; classtype:attempted-dos; sid:275; rev:4;)
	666	alert icmp $EXTERNAL_NET any -> $HOME_NET any (msg:"ICMP icmpenum v1.1.1"; id:666; dsize:0; itype:8; icmp_id:666 ; icmp_seq:0; reference:arachnids,450; classtype:attempted-recon; sid:471; rev:1;)
	678	alert icmp $EXTERNAL_NET any -> $HOME_NET any (msg:"DDOS TFN Probe"; id:678; itype:8; content:"1234"; reference:arachnids,443; classtype:attempted-recon; sid:221; rev:1;)

Table B.1: Continued

Snort Option	Ways Used	Example Rule
	3868	alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"DOS Land attack"; id:3868; seq:3868; flags:S; reference:cve,CVE-1999-0016; classtype:attempted-dos; sid:269; rev:3;)
	13170	alert icmp $EXTERNAL_NET any -> $HOME_NET any (msg:"ICMP PING LINUX/*BSD"; dsize:8; itype:8; id:13170; reference:arachnids,447; sid:375; classtype:misc-activity; rev:4;)
	39426	alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"SCAN synscan portscan"; id:39426; flags:SF; reference:arachnids,441; classtype:attempted-recon; sid:630; rev:1;)
ipopts	lsrr	alert ip $EXTERNAL_NET any -> $HOME_NET any (msg:"MISC source route lssr"; ipopts:lsrr; reference:bugtraq,646; reference:cve,CVE-1999-0909; reference:arachnids,418; classtype:bad-unknown; sid:500; rev:2;)
	lsrre	alert ip $EXTERNAL_NET any -> $HOME_NET any (msg:"MISC source route lssre"; ipopts:lsrre; reference:bugtraq,646; reference:cve,CVE-1999-0909; reference:arachnids,420; classtype:bad-unknown; sid:501; rev:2;)
	rr	alert icmp $EXTERNAL_NET any -> $HOME_NET any (msg:"ICMP Traceroute ipopts"; ipopts:rr; itype:0; reference:arachnids,238; sid:455; classtype:misc-activity; rev:5;)
	ssrr	alert ip $EXTERNAL_NET any -> $HOME_NET any (msg:"MISC source route ssrr"; ipopts:ssrr ; reference:arachnids,422; classtype:bad-unknown; sid:502; rev:1;)
ip_proto	!1	alert ip $EXTERNAL_NET any -> $HOME_NET any (msg:"BAD TRAFFIC Non-Standard IP protocol"; ip_proto:!1; ip_proto:!2; ip_proto:!6; ip_proto:!47; ip_proto:!50; ip_proto:!51; ip_proto:!89; classtype:non-standard-protocol; sid:1620; rev:3;)

Table B.1: Continued

Snort Option	Ways Used	Example Rule
	2	alert ip $EXTERNAL_NET any -> $HOME_NET any (msg:"DOS IGMP dos attack"; content:"|02 00|"; depth:2; ip_proto:2; fragbits:M+; reference:cve,CVE-1999-0918; classtype:attempted-dos; sid:272; rev:2;)
	>134	alert ip $EXTERNAL_NET any -> $HOME_NET any (msg:"BAD-TRAFFIC Unassigned/Reserved IP protocol"; ip_proto:>134; reference:url,www.iana.org/assignments/protocol-numbers; classtype:non-standard-protocol; sid:1627; rev:3;)
itype*	0	alert icmp 255.255.255.0/24 any -> $HOME_NET any (msg:"BACKDOOR SIGNATURE - Q ICMP"; itype:0; dsize:>1; reference:arachnids,202; sid:183; classtype:misc-activity; rev:3;)
nocase		alert tcp $HTTP_SERVERS $HTTP_PORTS -> $EXTERNAL_NET any (msg:"ATTACK-RESPONSES command completed"; content:"Command completed"; nocase; flow:from_server,established; classtype:bad-unknown; sid:494; rev:6;)
offset*	0	alert tcp $HOME_NET 512 -> $EXTERNAL_NET any (msg:"ATTACK-RESPONSES rexec username too long response"; flow:from_server,established; content:"username too long"; offset:0; depth:17; classtype:unsuccessful-user; sid:2104; rev:2;)
rawbytes		alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"WEB-MISC ///cgi-bin access"; flow:to_server,established; uricontent:"///cgi-bin"; nocase; rawbytes; classtype:attempted-recon; sid:1143; rev:5;)

Table B.1: Continued

Snort Option	Ways Used	Example Rule
rpc	100000,*,*	alert tcp $EXTERNAL_NET any -> $HOME_NET 111 (msg:"RPC portmap listing"; flow:to_server,established; rpc:100000,*,*; reference:arachnids,429; classtype:rpc-portmap-decode; sid:596; rev:5;)
	100009,*,*	alert udp $EXTERNAL_NET any -> $HOME_NET 111 (msg:"RPC portmap request yppasswdd"; rpc:100009,*,*; reference:bugtraq,2763; classtype:rpc-portmap-decode; sid:1296; rev:4;)
sameip		alert ip any any -> any any (msg:"BAD-TRAFFIC same SRC/DST"; sameip; reference:cve,CVE-1999-0016; reference:url,www.cert.org/advisories/CA-1997-28.html; classtype:bad-unknown; sid:527; rev:4;)
seq	0	alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"SCAN NULL"; flags:0; seq:0; ack:0; reference:arachnids,4; classtype:attempted-recon; sid:623; rev:1;)
	3868	alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"DOS Land attack"; id:3868; seq:3868; flags:S; reference:cve,CVE-1999-0016; classtype:attempted-dos; sid:269; rev:3;)
	6060842	alert tcp $EXTERNAL_NET any <> $HOME_NET any (msg:"DOS NAPTHA"; flags:S; seq:6060842; id:413; reference:cve,CAN-2000-1039; reference:url,www.microsoft.com/technet/security/bulletin/MS00-091.asp; reference:url,www.cert.org/advisories/CA-2000-21.html; reference:url,razor.bindview.com/publish/advisories/adv_NAPTHA.html; reference:bugtraq,2022; classtype:attempted-dos; sid:275; rev:4;)

Table B.1: Continued

Snort Option	Ways Used	Example Rule
	101058054	alert tcp $EXTERNAL_NET 80 -> $HOME_NET 1054 (msg:"BACKDOOR ACKcmdC trojan scan"; seq:101058054; ack:101058054; flags:A,12; reference:arachnids,445; sid:106; classtype:misc-activity; rev:4;)
	674711609	alert tcp $HOME_NET any <> $EXTERNAL_NET any (msg:"DDOS shaft synflood"; flags:S,12; seq:674711609; reference:arachnids,253; classtype:attempted-dos; sid:241; rev:3;)
	1958810375	alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"SCAN ipEye SYN scan"; flags:S; seq:1958810375; reference:arachnids,236; classtype:attempted-recon; sid:622; rev:2;)
ttl	0	alert ip $EXTERNAL_NET any -> $HOME_NET any (msg:"BAD-TRAFFIC 0 ttl"; ttl:0; reference:url,www.isi.edu/in-notes/rfc1122.txt; reference:url,support.microsoft.com/default.aspx?scid=kb\; EN-US\; q138268; sid:1321; classtype:misc-activity; rev:6;)
	1	alert icmp $EXTERNAL_NET any -> $HOME_NET any (msg:"ICMP traceroute"; ttl:1; itype:8; reference:arachnids,118; classtype:attempted-recon; sid:385; rev:3;)
	>220	alert tcp $EXTERNAL_NET 10101 -> $HOME_NET any (msg:"SCAN myscan"; ttl:>220; ack:0; flags:S; reference:arachnids,439; classtype:attempted-recon; sid:613; rev:1;)
uricontent*	"#filename=*.asp"	alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"WEB-IIS asp-srch attempt"; flow:to_server,established; uricontent:"#filename=*.asp"; nocase; classtype:web-application-attack; sid:998; rev:5;)

Table B.1: Continued

Snort Option	Ways Used	Example Rule
within	1	alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"WEB-MISC Lotus Notes .csp script source download attempt"; flow:to_server,established; uricontent:".csp"; content:".csp"; content:"."; within:1; classtype:web-application-attack; sid:2064; rev:2;)
	2	alert tcp $EXTERNAL_NET 80 -> $HOME_NET any (msg:"MULTIMEDIA Windows Media audio download"; flow:from_server,established; content:"Content-type\; content:"|0a|"; within:2; classtype:policy-violation; sid:1437; rev:3;)
	3	alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:"INFO FTP No Password"; content:"PASS"; nocase; offset:0; depth:4; content:"|0a|"; within:3; reference:arachnids,322; flow:from_client,established; classtype:unknown; sid:489; rev:5;)
	4	alert tcp $EXTERNAL_NET any -> $HOME_NET 111 (msg:"RPC portmap tooltalk request TCP"; flow:to_server,established; content:"|00 00 00 00|"; offset:8; depth:4; content:"|00 01 86 A0|"; offset:16; depth:4; content:"|00 00 00 03|"; distance:4; within:4; byte_jump:4,4,relative,align; byte_jump:4,4,relative,align; content:"|00 01 86 F3|"; within:4; reference:cve,CAN-2001-0717; reference:cve,CVE-1999-0003; reference:cve,CVE-1999-0687; reference:cve,CAN-1999-1075; reference:url,www.cert.org/advisories/CA-2001-05.html; classtype:rpc-portmap-decode; sid:1298; rev:10;)

Table B.1: Continued

Snort Option	Ways Used	Example Rule
	8	alert udp $EXTERNAL_NET any -> $HOME_NET 67 (msg:"MISC bootp hostname format string attempt"; content:"|01|"; offset:0; depth:1; content:"|0C|"; distance:240; content:"%"; distance:0; content:"%"; distance:1; within:8; content:"%"; distance:1; within:8; reference:bugtraq,4701; classtype:misc-attack; sid:2039; rev:1;)
	10	alert tcp $EXTERNAL_NET any -> $HOME_NET 110 (msg:"POP3 CAPA overflow attempt"; flow:to_server,established; content:"CAPA"; nocase; content:!"|0a|"; within:10; classtype:attempted-admin; sid:2108; rev:1;)
	15	alert ip $HOME_NET any -> $EXTERNAL_NET any (msg:"ATTACK-RESPONSES id check returned userid"; content:"uid="; byte_test:5,<,65537,0,relative,string; content:"gid="; distance:1; within:15; byte_test:5,<,65537,0,relative,string; classtype:bad-unknown; sid:1882; rev:7;)
	30	alert tcp $SMTP_SERVERS any -> $EXTERNAL_NET 25 (msg:"VIRUS OUTBOUND .pif file attachment"; flow:to_server,established; content:"Content-Disposition|3a|"; content:"filename=|22|"; distance:0; within:30; content:".pif|22|"; distance:0; within:30; nocase; classtype:suspicious-filename-detect; sid:721; rev:4;)
	50	alert tcp $EXTERNAL_NET any -> $HOME_NET 110 (msg:"POP3 USER overflow attempt"; flow:to_server,established; content:"USER"; nocase; content:!"|0a|"; within:50; reference:bugtraq,789; reference:cve,CVE-1999-0494; reference:nessus,10311; classtype:attempted-admin; sid:1866; rev:5;)
	64	alert tcp $EXTERNAL_NET 119 -> $HOME_NET any (msg:"NNTP return code buffer overflow attempt"; flow:to_server,established,no_stream; content:"200 "; offset:0; depth:4; content:!"|0a|"; within:64; reference:bugtraq,4900; reference:cve,CAN-2002-0909; classtype:protocol-command-decode; sid:1792; rev:5;)

Table B.1: Continued

Snort Option	Ways Used	Example Rule
	100	alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:"FTP CEL overflow attempt"; flow:to_server,established; content:"CEL "; nocase; content:!"|0a|"; within:100; reference:bugtraq,679; reference:cve,CVE-1999-0789; reference:arachnids,257; classtype:attempted-admin; sid:337; rev:5;)
	128	alert udp $EXTERNAL_NET any -> $HOME_NET 1900 (msg:"MISC UPnP Location overflow"; content:"|0d|Location|3a|"; nocase; content:!"|0a|"; within:128; classtype:misc-attack; reference:cve,CAN-2001-0876; sid:1388; rev:4;)
	150	alert tcp any any -> any 6666:7000 (msg:"EXPLOIT CHAT IRC Ettercap parse overflow attempt"; flow:to_server,established; content:"PRIVMSG nickserv IDENTIFY"; nocase; offset:0; content:!"|0a|"; within:150; reference:url,www.bugtraq.org/dev/GOBBLES-12.txt; classtype:misc-attack; sid:1382; rev:7;)
	255	alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"WEB-IIS WEBDAV nessus safe scan attempt"; flow:to_server,established; content:"SEARCH / HTTP/1.1|0d0a|Host|3a|"; content:"|0d0a0d0a|"; within:255; reference:cve,CAN-2003-0109; reference:bugtraq,7116; reference:nessus,11412; classtype:attempted-admin; sid:2091; rev:2;)
	256	alert tcp $EXTERNAL_NET any -> $HOME_NET 109 (msg:"POP2 FOLD overflow attempt"; flow:to_server,established; content:"FOLD "; content:!"|0A|"; within:256; reference:bugtraq,283; reference:cve,CVE-1999-0920; classtype:attempted-admin; sid:1934; rev:1;)

Table B.1: Continued

Snort Option	Ways Used	Example Rule
	500	alert tcp $EXTERNAL_NET any -> $HOME_NET 119 (msg:"NNTP AUTHINFO USER overflow attempt"; flow:to_server,established; content:"AUTHINFO USER "; nocase; depth:14; content:!"|0a|"; within:500; reference:cve,CAN-2000-0341; reference:arachnids,274; classtype:attempted-admin; sid:1538; rev:5;)
	512	alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"WEB-MISC long basic authorization string"; flow:to_server,established; content:"Authorization\; nocase; content:!"|0A|"; within:512; classtype:attempted-dos; reference:bugtraq,3230; sid:1260; rev:6;)
	600	alert tcp $EXTERNAL_NET 22 -> $HOME_NET any (msg:"EXPLOIT SSH server banner overflow"; flow:established,from_server; content:"SSH-"; offset:0; depth:4; content:!"|0a|"; within:600; reference:bugtraq,5287; classtype:misc-attack; sid:1838; rev:4;)
	1024	alert tcp $EXTERNAL_NET any -> $HOME_NET 143 (msg:"IMAP rename overflow attempt"; flow:established,to_server; content:" RENAME "; nocase; content:!"|0a|"; within:1024; reference:nessus,10374; reference:cve,CAN-2000-0284; classtype:misc-attack; sid:1903; rev:3;)

Table B.2: References in Snort Rules

Reference	URL Base
arachNIDS	http://www.whitehats.com/info/IDS
bugtraq	http://www.securityfocus.com/bid/
cve	http://cve.mitre.org/cgi-bin/cvename.cgi?name=
McAfee	http://vil.nai.com/vil/content/v_
nessus	http://cgi.nessus.org/plugins/dump.php3?id=
url	http://

C. Compiler

This appendix discusses the implementation of the prototype compiler from Snort IDS rules to NFR N-Code functions. For the reader totally unfamiliar with this compiler, section 4.7 presents the purposes and an overview of the compiler in a less technical manner. Furthermore, the results of the prototype implementation which appear in section 4.7 are not repeated in this appendix. It is generally assumed that the reader has examined section 4.7 before examining this appendix. Snort language issues are covered in section 4.7 with extensive justification in appendix B. The reasoning behind the presentation of prototype code is stated in the introduction to appendix A.

Appendix C is organized as follows: first, section C.1 gives an overview of the prototype including the outputs with an example; second, section C.2 discusses the main compilation loop; third, section C.3 discusses the compilation of mandatory Snort fields; forth, section C.4 discusses the compilation of Snort options with common language features; fifth, section C.5 discusses the compilation of Snort options with special language features; and sixth, section C.6 discusses the compilation of the `content` Snort option.

C.1 Overview

The compiler from Snort to NFR N-Code discussed in section 4.7 was implemented as a translator. Since there was only one output module, it was unnecessary to define a meta-level language representing an arbitrary signature based IDS. As the prototype was implemented, the parsed Snort rules were used for the tree based compilation. The Snort rules were transformed in the compiler so that rather than being an association list, suboptions of the options were grouped with the option.

Figure C.1 shows an example Snort rule, its parsed form, its regrouped form and its compiled form. The compiled form of this rule has a long listing because of the length of the code needed to handle the `content` rules.

The option regrouping is detailed in figures C.2 and C.3. During the option regrouping, the `stream-assemble` internal compiler option is added to the rule if any of the options require an examination of the reassembled stream. As the compiler is written, the compilation of `stream-assemble` outputs code for TCP re-assembly or sets the same variables without assembly if it is called on UDP or IP layer signatures. Figure C.4 shows how the re-assembly code is generated.

The output of the prototype compiler is a N-Code function which will alarm under the same conditions as the Snort rule. This N-Code function to output is represented as a string. For ease of internal use, N-Code code is represented as a tree of strings which are later appended. Figure C.5 shows this process. The string representation was chosen because it sped the development of a limited compiler. Another possibility for an internal N-Code representation is an abstract syntax tree (AST) structure. The AST representation was not used for the prototype because it involves more software development to reach a point where the same functionality is present. For the functionality used in the prototype, lists of strings as an N-Code representation are sufficiently powerful to do an acceptable job.

The interface between the N-Code function and the NFR IDS is not output by this compiler. This would be a useful function for the compiler to do, but as a version of the NFR IDS was not available, the generated interface would not have been tested, hence was not developed.

C.2 Main Compilation Procedure

The main compilation loop is shown in figure C.6. For space reasons, not every field that is compiled will be remarked upon. As the fields in a Snort rule are considered to form a logical ‘and’ statement, the compilation loop usually just outputs

Snort Rule

```
alert ip any any -> any any (content:"abc"; nocase; content:"xyz"; within:23; )
```

Parsed Form

```
(("alert" "alert")
 ("proto" "ip")
 ("src-net" "any")
 ("src-port" "any")
 ("dir" "->")
 ("dst-net" "any")
 ("dst-port" "any")
 ("content" "\"abc\"")
 ("nocase")
 ("content" "\"xyz\"")
 ("within" "23"))
```

Regrouped Parsed Form

```
(("stream-assemble")
 ("alert" "alert")
 ("proto" "ip")
 ("src-net" "any")
 ("src-port" "any")
 ("dir" "->")
 ("dst-net" "any")
 ("dst-port" "any")
 ("content-group"
   ((("content" "\"abc\"") ("nocase"))
    (("content" "\"xyz\"") ("within" "23")))))
```

Compiled Form

```
$t1[0] = -1;
$t1[1] = 0;
if ($pm == null)
  $pm[-1] = $t1;
$t1 = null;
if ($stream_index == null)
  $stream_index = 0;
$buf = $llc.blob;
if (1) {
```

Figure C.1: Stages of Compilation Example

```
if (1) {
  $content_str[0] = ”abc”;
  $content_str[1] = ”xyz”;
  $min_distance[0] = 0;
  $min_distance[1] = 0;
  $max_within[0] = (-1);
  $max_within[1] = 23;
  $nocase[0] = 1;
  $nocase[1] = 0;
  $offset[0] = 0;
  $offset[1] = 0;
  $offset[-1] = 0;
  $max_within[-1] = -1;
  $min_distance[-1] = 0;
  $done_content = 0;
  $match = 0;
  $stream_char_done = 0;
  $stream_index_adj = 0;
  $s = $buf;
  while (!$done_content) {
    $i = -1;
    while (($i < $n_pm) && (!$match)) {
      $x = $pm[$i];
      if (($max_within[$x[0]] >= 0)
          && (($x[1] + $max_within[$x[0]]) < $stream_index)) {
        $pm[$i] = $pm[$n_pm - 1];
        $n_pm = $n_pm - 1;
        $pm[$n_pm] = null;
        $i = $i - 1;
      } else {
        if (($stream_index >= $offset[$x[0]])
            && (($x[1] + $min_distance[$x[0]]) >= $stream_index)) {
          $submatch = 0;
          if ($nocase[$i]) {
            if (strcasecomp ($content_str[$x[0]], $s))
              $submatch = 1;
          } else {
            if (strcomp ($content_str[$x[0]], $s))
              $submatch = 1;
```

Figure C.1: Continued

```
            }
            if ($submatch) {
              $next_str_i = ($i) + 1;
              if ($next_str_i >= 2) {
                $match = 1;
              } else {
                $pm[$n_pm] =
                  [$next_str_i, $stream_index + strlen ($content_str[$i])];
                $n_pm = $n_pm + 1;
              }
            } else {
              if (strlen ($s) < strlen ($content_str[$x[0]])) {
                $insuf_length = 1;
              }
            }
          }
          $i = $i + 1;
        }
        if ($match) {
          $done_content = 1;
        }
        if (strlen ($s) <= 1)
          $done_content = 1;
        else
          $s = substring ($s, 1);
        $stream_index = $stream_index + 1;
        if ($insuf_length)
          $stream_char_done = $stream_char_done + 1;
        else
          $stream_index_adj = $stream_index_adj + 1;
      }
    }
    $buf = substring ($buf, $stream_index_adj);
    $stream_index = $stream_index - $stream_index_adj;
    if ($match) {
      echo _alert_msg;
    }
  }
}
```

Figure C.1: Continued

```
(define snort-group-suboptions
  (lambda (parsed-snort-rule)
    (comp-add-stream-assemble-field
     (snort-group-content
      (snort-group-uricontent parsed-snort-rule)))))

(define (comp-add-stream-assemble-field x) ; x is a parsed-snort-rule
  (if (ormap (lambda (y) (assoc y x))
             '("content-group" "uricontent-group" "uricontent" "content"))
      (cons (cons "stream-assemble" '()) x)
      x))

(define (snort-group-content x)
  (snort-group-fields
   x
   "content-group"
   '("content")
   '("nocase" "distance" "within" "offset")))

(define (snort-group-uricontent x)
  (snort-group-fields
   x
   "uricontent-group"
   '("uricontent")
   '("nocase")))
```

Figure C.2: Regrouping Snort Options

an ‘if’ statement with a test for the Snort option field followed by the remainder of the compilation. The exception to this strategy is the handling of the direction field. The handling of the direction field is shown at the end of figure C.6 and in figure C.7. The direction field is handled by a logical ‘or’ with a duplication of the rule with the directions reversed. A simple example of a compiled rule versus the bidirectional rule is shown in figure C.8. In figure C.8 the time to live (ttl) field is extracted by offset from the packet boundary rather than the `ttl` mnemonic used by Snort. This is typical of N-Code field matching where the offset into the packet must be specified rather than a mnemonic.

```
(define snort-group-fields
  (lambda (snort-rule group-name group-triggers group-additional-fields)
    (if (ormap (lambda (t) (assoc t snort-rule)) group-triggers)
       (let outer-loop
           ((r snort-rule)
            (g '()))
         (cond
          [(null? r)
           (list (list group-name g))]
          [(member (caar r) group-triggers)
           (let inner-loop
               ((x (cdr r))
                (l (list (car r))))
             (cond
              [(or (null? x)
                   (not (member (caar x) group-additional-fields)))
               (outer-loop x (append g (list l)))]
              [else
               (inner-loop (cdr x) (append l (list (car x))))]))]
          [else (cons (car r) (outer-loop (cdr r) g))]))
       snort-rule)))
```

Figure C.3: General Regrouping

C.3 Mandatory Options

The handling of network address is shown in figures C.9 and C.10. It is straightforward as NFR has functions for determining if an address is within a subnet. Snort addresses can be lists of addresses, which are easily translatable into a logical ‘or’ statement of tests. It is worth noting that Snort variables can be lists of network addresses, so they must be substituted for before the compilation to insure that they get translated. Thus the substitution of Snort variables must happen before compilation. Because the network address field is straightforward to compile, a separate example of the compilation has not been included.

The handling of port ranges is shown in figures C.11 and C.12. The 2.0 version of Snort on which this system was developed did not support lists of ports [54]. It is

```
(define (comp-stream-assemble proto uses rest)
  (let ((content-inits
         '("$t1[0] = -1; $t1[1] = 0; " ; put an initial $pm[-1]=[-1,0]
           "if ($pm == null) $pm[-1] = $t1; "
           "$t1 = null; "
           "if ($stream_index == null) $stream_index = 0; ")))
    (case proto
      [(ip)
       (list content-inits " $buf = $llc.blob; " rest)]
      [(icmp)
       (list content-inits " $buf = $ip.blob; " rest)]
      [(udp)
       (list content-inits " $buf = $udp.blob; " rest)]
      [(tcp)
       (list
           ; these two are needed for content matches
        "declare $stream_index inside tcp.connSym; "
        "declare $pm inside tcp.connSym; "
        content-inits
      " declare $buf inside tcp.connSym; "
      "declare $segs inside tcp.connSym; "
      "declare $next_seq inside tcp.connSym; "
      "if ((null == $buf) && (tcp.length > 0)) { "
           "$buf = tcp.blob; "
           "$next_seq = ulong(ip.blob,8) + tcp.length; "
       "} else { "
           "segs[ulong(ip.blob, 8)] = tcp.blob; "
           "$new_tcp_stream_data = 0; "
           "while (null != segs[$next_seq]) { "
              "$new_tcp_stream_data = 1; "
              "$buf = cat($buf, segs[$next_seq]); "
              "$last_seq = $next_seq; "
              "$next_seq = $next_seq + strlen(segs[$next_seq]); "
              "segs[$last_seq] = null; } "
           "if ($new_tcp_stream_data) { "
              rest
           "}}")])))
```

Figure C.4: Compiling Stream Assemble

```
(define (flatten-tree l)
  (if (list? l)
      (apply append (map flatten-tree l))
      (list l)))

(define (try-comp-to-string snort-rule)
  (apply string-append (flatten-tree (try-comp snort-rule))))

(define snort->n-code
  (lambda (snort-rule . l)
    (try-comp-to-string
     (snort-group-suboptions
      (cond
       [(list? snort-rule) snort-rule]
       [(null? l) (parse-snort-rule snort-rule)]
       [else (parse-snort-rule ((car l) 'sub snort-rule))])))))
```

Figure C.5: Compiler Output Processing

expected that future versions of Snort will support lists of ports. Under the current version of Snort, to specify both HTTP ports 80 and 8080, it is necessary to specify all ports from 80 to 8080. The options available to the port range Snort field are accurately described in the manual [54].

C.4 Options with Common Language Features

Arguments to Snort options which are listed in table 4.1 as using a `numerical` field are compiled with *comp-numerical-compare* shown in figure C.13. As may be seen from figure C.6, each Snort option using this compilation function passes in the proper N-Code variable reference for comparison. The example in figure C.8 uses *comp-numerical-compare* on the `ttl` field for an exact comparison. *Comp-numerical-compare* is used for the compilation of the following options: `ack`, `dsize`, `icmp_id`, `icmp_seq`, `icode`, `id`, `ip_proto`, `itype`, `seq`, and `ttl`.

Arguments to Snort options which are listed in table 4.1 as using a `string` field are compiled with *snort-content->n-code-content* shown in figure C.14. For example,

```
(define try-comp
  (lambda (snort-rule)
    (let ((proto (string->symbol (cadr (assoc "proto" snort-rule))))
          (snort-dir (cadr (assoc "dir" snort-rule))))
      (letrec
          ((loop
            (lambda (l dir)
              (letrec
                  ((do-pred
                    (lambda (p)
                      (list " if ( "
                          p
                          " ) { "
                          (loop (cdr l) dir)
                          " } ")))
                   (pre-pred
                    (lambda (pre x)
                      (list "if ( " pre " ) { " x " } " )))
                   (do-netaddr-pred
                    (lambda (v)
                      (do-pred (comp-netaddr-pred v (cadar l)))))
                   (do-port
                    (lambda (x)
                      (let ((f (lambda (x)
                                 (do-pred (comp-port-pred x (cadar l))))))
                        (case proto
                          [(tcp)
                           (f (string-append "tcp." x "port"))]
                          [(udp)
                           (f (string-append "udp." x "port"))]
                          [else (loop (cdr l) dir)]))))) ; no icmp ports
                (if (null? l)
                   '(" echo _alert_msg; ")
```

Figure C.6: Main Compilation Loop

```
(case-equal (caar l)
  [("src-net")
   (do-netaddr-pred "ip.src")]
  [("dst-net")
   (do-netaddr-pred "ip.dst")]
  [("src-port")
   (do-port "source")]
  [("dst-port")
   (do-port "dest")]
  [("ttl")
   (do-pred (comp-numerical-compare
             "byte(ip.blob,8) "
             (cadar l)))]
  [("sameip")
   (do-pred (list "ip.src == ip.dst"))]
  [("ip_proto") ; only numerics are used
   (do-pred (comp-numerical-compare
             " ip.protocol "
             (cadar l)))]
  [("itype")
   (do-pred (comp-numerical-compare
             "icmp.type"
             (cadar l)))]
  [("icode")
   (do-pred (comp-numerical-compare
             "icmp.code"
             (cadar l)))]
  [("icmp_seq")
   (do-pred (comp-numerical-compare
             "(long(ip.blob,4) & 0xffff)" ; Stevens p74.
             (cadar l)))]
  [("icmp_id")
   (do-pred (comp-numerical-compare
             "(long(ip.blob,4) >> 16)"
             (cadar l)))]
```

Figure C.6: Continued

```
[("seq")  ; (assert (eq? proto 'tcp))
 (do-pred (comp-numerical-compare
            "(long(ip.blob,4))"
            (cadar l)))]
[("id")
 (do-pred (comp-numerical-compare
            "(long(llc.blob,4) >> 16)"
            (cadar l)))]
[("ack")  ; (assert (eq? proto 'tcp))
 (do-pred (comp-numerical-compare
            "(long(ip.blob,8))"
            (cadar l)))]
[("dsize")
 (do-pred (comp-numerical-compare
            "ip.len"
            (cadar l)))]
[("flags")
 (do-pred (comp-tcp-flags
            "((long(ip.blob,12) >> 16) & 0x000001ff)"
            (cadar l)))]
[("fragbits")
 (do-pred (comp-ip-frag-flags
            "(byte(llc.blob,6) >> 5)"
            (cadar l)))]
[("ipopts")
 (pre-pred
   ; check length
  (list "(((byte(llc.blob,0) & 0x0f) << 2) > 20) ")
  (do-pred (comp-ip-opts (cadar l))))]
[("flow")
 (do-pred (comp-flow (cadar l) dir))]
[("stream-assemble")
 (comp-stream-assemble
  proto
  (car l)
  (loop (cdr l) dir))]
[("content-group")
 (comp-content-group (cadar l) (loop (cdr l) dir))]
```

Figure C.6: Continued

```
                     [("uricontent-group")
                      (comp-uricontent (caadar l) (loop (cdr l) dir))]
                     [else (loop (cdr l) dir)])))))))
      (if (equal? snort-dir "<>")
          (list
           (loop snort-rule 'forward)
           (loop (reverse-dirs snort-rule) 'reverse))
          (loop snort-rule 'forward))))))
```

Figure C.6: Continued

```
(define (reverse-dirs x)
  (let* ((m (map list snort-main-field-list (snort-rule-main x)))
         (f (lambda (y) (cadr (assoc y m)))))
    (make-snort-rule
     (list "alert"
           (f "proto")
           (f "dst-net") (f "dst-port")
           "<-reverse"
           (f "src-net") (f "src-port"))
     (snort-rule-options x))))
```

Figure C.7: Reversing Directions

Snort string "abc|00|efg" compiles to N-Code string "abc\x00efg" , while the invalid Snort string "|0|" generates a compilation error.

Each Snort field using the `bitfield` in table 4.1 defined different mnemonics for the flags, but allow the same masking and test operations with the same syntax. Because the same syntax was used, a general bitfield compilation procedure called *comp-flags* was developed. The mnemonics used for the TCP flag field are shown in figure C.15 along with the instantiation of the general flag matching compilation function. Similarly, figure C.16 details *comp-ip-frag-flags*. The general flag compilation procedure is detailed in figure C.17. Examples of compiling flag fields are shown in figure C.18.

Single Direction Rule: alert udp 1.2.3.4 any -> 5.6.7.8 19 (ttl:0;)

```
if ((ip.src == 1.2.3.4)) {
  if (1) {
    if ((ip.dst == 5.6.7.8)) {
      if ((udp.destport == 19)) {
        if (byte (ip.blob, 8) == 0) {
          echo _alert_msg;
        }
      }
    }
  }
}
```

Bi-directional Rule: alert udp 1.2.3.4 any <> 5.6.7.8 19 (ttl:0;)

```
if ((ip.src == 1.2.3.4)) {
  if (1) {
    if ((ip.dst == 5.6.7.8)) {
      if ((udp.destport == 19)) {
        if (byte (ip.blob, 8) == 0) {
          echo _alert_msg;
        }
      }
    }
  }
}
if ((ip.src == 5.6.7.8)) {
  if ((udp.sourceport == 19)) {
    if ((ip.dst == 1.2.3.4)) {
      if (1) {
        if (byte (ip.blob, 8) == 0) {
          echo _alert_msg;
        }
      }
    }
  }
}
```

Figure C.8: Example of Bidirectional Rule Compilation

```
(define (comp-netaddr-pred v snort-addr)
  (let comp
      ((l (parse-snort-net-addr-range snort-addr)))
    (case (car l)
      [(not) (list "! ( " (comp (cadr l)) " ) ")]
      [(net-addr-list)
       (if (null? (cddr l))
           (list " ( " (comp (cadr l)) " ) ")
           (list " ( " (comp (cadr l))
                 " || " (comp (cons 'net-addr-list (cddr l)))
                 " ) " ))]
      [(any) " 1 "] ; #t
      [(netmask-addr) (list " ( " v " inside " (cadr l) " )")]
      [(ip-addr) (list " ( " v " == " (cadr l) " ) " )])))
```

Figure C.9: Compiling Network Addresses

```
(define parse-snort-net-addr-range
  (lambda (s)
    (let ((x (string-length s))
          (f parse-snort-net-addr-range))
      (cond
       [(zero? x) (error 'snort-net-addr
                         "no address specified or list with empty element")]
       [(equal? #\! (string-ref s 0))
        (list 'not (f (substring s 1 x)))]
       [(equal? #\[ (string-ref s 0))
        (cons 'net-addr-list
              (map f (split #\, (substring s 1 (- x 1)))))]
       [(equal? s "any") (list 'any)]
       [else
        (list (if (member #\/ (string->list s))
                  'netmask-addr
                  'ip-addr)
              s)])))))
```

Figure C.10: Parsing Snort Network Address Ranges

```
(define (comp-port-pred n-code-name snort-port-string)
  (let comp
       ((l (parse-snort-port-range snort-port-string)))
    (case (car l)
      [(any) " 1 "]
      [(not) (list "! ( " (comp (cadr l)) " ) ")]
      [(upto) (list "( " n-code-name " <= " (cadr l) " ) ")]
      [(from) (list "( " n-code-name " >= " (cadr l) " ) ")]
      [(range) (list "( " (comp (list 'from (cadr l))) " && "
                     (comp (list 'upto (caddr l))) " ) ")]
      [(exact) (list "( " n-code-name " == " (cadr l) " ) ")]
      [else (error 'comp-port "unknown parsed snort port type ~a~n" (car l))])))
```

Figure C.11: Compiling Port Ranges

```
(define parse-snort-port-range ; port lists aren't currently supported in Snort.
  (lambda (s)
    (cond
     [(< (string-length s) 1) (error 'snort-port-range "no range specified")]
     [(equal? #\! (string-ref s 0))
      (list 'not (parse-snort-port-range (substring s 1 (string-length s))))]
     [(equal? s "any")
      (list 'any)]
     [else
      (if (member #\: (string->list s))
          (let* ((x (split #\: s))
                 (from-p (car x))
                 (to-p (cadr x)))
            (cond
             [(equal? "" from-p) (list 'upto to-p)]
             [(equal? "" to-p) (list 'from from-p)]
             [else (list 'range from-p to-p)]))
          (list 'exact s))])))
```

Figure C.12: Parsing Snort Port Ranges

```
(define (comp-numerical-compare v s)
  (let*
      ((n (string-length s))
       (c (if (> n 0) (string-ref s 0) #\0))
       (r (substring s 1 n)))
     (case-equal c
      [(#\> #\<)
       (list v " " (string c) " " r)]
      [(#\!)
       (list "!("
             (comp-numerical-compare v r)
             ")")]
      [else (list v " == " s)])))
```

Figure C.13: Compiling Numerical Comparison Operations

C.5 Options with Special Languages

Fields marked special in table 4.1 each have their own compilation function. Usually, this is because the field used special mnemonics to specify the argument. For example, figure C.19 shows that the `ipoption` field simply has special mnemonics. Figure C.20 uses an aspect of some special N-Code variables for the compilation of the `flow` Snort option.

The `uricontent` option outputs as significant amount of code when compiled compared with the other options treated so far, but is still significantly simpler than the `content` option. Simply put, the N-Code code for the `uricontent` option stores the stream and rejects matches until three whitespace characters have been found. Then it extracts and decodes the URI field, which is the third field. It compares this decoded URI with the specified string and decides if a match has occurred. If a match has not occurred, then the rule cannot match on the string and this state is stored. Otherwise, the remainder of the tests are run. Figure C.21 lists the code to compile the `uricontent` field. Figure C.22 shows a compilation of an example rule with the `uricontent` field.

```
(define (snort-content->n-code-content s)
  (list->string
   (letrec
       ((char-state
         (lambda (l)
           (cond
            [(null? l) '()]
            [(equal? (car l) #\|)
             (byte-code-state (cdr l))]
            [(equal? (car l) #\\)
             (escaped-char-state (cdr l))]
            [else (cons (car l) (char-state (cdr l)))])))
        (escaped-char-state
         (lambda (l)
           (cond
            [(null? l)
             (cons #\\ '())]
            [(member (car l) '(#\: #\;))
             (cons (car l) (char-state (cdr l)))]
            [else ;#\\& #\" also covered.
             (cons #\\ (cons (car l) (char-state (cdr l))))])))
        (byte-code-state
         (lambda (l)
           (cond
            [(null? l) (error 'compatablity-snort-content
                              "Mismatched pipes for bytecode delimination")]
            [(equal? (car l) #\|)
             (char-state (cdr l))]
            [(equal? (car l) #\space)
             (byte-code-state (cdr l))]
            [else (append (list #\\ #\x (car l))
                          (second-byte-state (cdr l)))])))
```

Figure C.14: Compiling Strings from Snort to N-Code

```
    (second-byte-state
     (lambda (l)
       (cond
        [(null? l) (error 'compatablity-snort-content
                          "Mismatched pipes for bytecode delimination")]
        [(equal? (car l) #\|)
         (error 'compatablity-snort-content
                "Odd number of hexademical digits in bytecode")]
        [(equal? (car l) #\space)
         (second-byte-state (cdr l))]
        [else (cons (car l) (byte-code-state (cdr l)))]))))
  (char-state (string->list s)))))
```

Figure C.14: Continued

```
(define (comp-tcp-flags v s)
  (comp-flags v s (lambda (c)
                    (case-equal c
                     [(#\F) "0x00000001"]
                     [(#\S) "0x00000002"]
                     [(#\R) "0x00000004"]
                     [(#\P) "0x00000008"]
                     [(#\A) "0x00000010"]
                     [(#\U) "0x00000020"]
                     [(#\2) "0x00000040"]
                     [(#\1) "0x00000080"]
                     [(#\0) "0x00000000"]
                     [else "0x00000000"])))) ; not a flag
```

Figure C.15: Compiling TCP Flags

```
(define (comp-ip-frag-flags v s)
  (comp-flags v s (lambda (c)
                    (case-equal c
                     [(#\R) "0x04"]
                     [(#\M) "0x02"]
                     [(#\D) "0x01"]
                     [else "0x00"]))))
```

Figure C.16: Compiling IP Fragmentation Flags

```
(define (comp-flags v s bit-lookup)
  (cond
   [(and (>= (string-length s) 1)
         (equal? #\! (string-ref s 0)))
    (list "!(" (comp-flags v (substring s 1 (string-length s))) ")")]
   [(and (>= (string-length s) 1)
         (equal? #\space (string-ref s 0)))  ; just in case
    (comp-flags v (substring s 1 (string-length s)))]
   [else
    (let*
        ((x (split #\, s))
         (flags (car x))
         (mask (if (null? (cdr x)) "0" (cadr x))))
      (letrec
          ((bits
            (lambda (l)
              (if (null? l)
                  (list " 0x00000000 ")
                  (let ((b (bit-lookup (car l))))
                    (list "( " b " | " (bits (cdr l)) ")")))))
           (exact-test
            (lambda (v m c)
              (list "( " v " & " m " ) == " c)))
           (any-test
            (lambda (v m c)
              (list "( " v " & " m " ) & " c)))
           (all-test
            (lambda (v m c)
              (list "(( " v " & " m " ) & " c " ) == " c))))
        (let ((l (string->list s)))
          ((cond
            [(member #\+ l) all-test]
            [(member #\* l) any-test]
            [else exact-test])
           v
           (list "( ~ ( " (bits (string->list mask)) " ))")
           (list " ( " (bits (string->list flags)) " ) "))))))]))
```

Figure C.17: Compiling Bit-Flag Fields

IP Fragmentation Flags String: RM

```
(((byte (llc.blob, 6) >> 5) & ((~(((0x00 | (0x00000000))))))) ==
 ((((0x04 | ((0x02 | (0x00000000))))))))
```

IP Fragmentation Flags String: RM+

```
((((byte (llc.blob, 6) >> 5) & ((~(((0x00 | (0x00000000))))))) &
  ((((0x04 | ((0x02 | ((0x00 | (0x00000000)))))))))) ==
 ((((0x04 | ((0x02 | ((0x00 | (0x00000000))))))))))
```

IP Fragmentation Flags String: RM*

```
(((byte (llc.blob, 6) >> 5) & (~((0x00 | 0x00000000)))) &
 ((0x04 | (0x02 | (0x00 | 0x00000000)))))
```

TCP Flags String: SF,12

```
((((long (ip.blob, 12) >> 16) &0x000001ff) &
  ((~(((0x00000080 | ((0x00000040 | (0x00000000))))))))) ==
 ((((0x00000002 | ((0x00000001 | (0x00000000))))))))
```

Figure C.18: Bit-Flag Examples

```
(define (comp-ip-opts s)
  (list "byte(llc.blob,20) == "  ; just check the first option.
        (case-equal s
         [("rr") "7"]  ; src: Stevens, TCP/IP Illustrated
         [("lsrr") "0x83"]
         [("ssrr") "0x89"]
         [else "0"]))) ; or error
```

Figure C.19: Compiling the Snort IP Option Field

```
(define (comp-flow s dir)
  (let ((l (map string-downcase (split #\, s))))
    (list
     "(tcp.conn) & "
     (if (or (and (eq? dir 'forward)
                  (or (member "to_server" l) (member "from_client" l)))
             (and (eq? dir 'reverse)
                  (or (member "from_server" l) (member "to_client" l))))
       "(tcp.Dst == ip.dst)"
       "(tcp.Dst == ip.src)"))))
```

Figure C.20: Compiling the Snort Flow Option

C.6 Content Option

The `content` is the most complicated Snort option to compile. The suboptions `distance`, `nocase`, `offset`, and `within` must be handled while compiling the `content` option. The code to compile the `content` option is shown in figure C.23. The complexity of the code is due to the need to store partial matches for later matching when more of the TCP stream is available while still ensuring that matches which are matchable with the current portion of the stream are matched now. An example compilation of a rule involving the `content` option is shown in figure C.1.

An examination of the desired behavior for the `content` option is necessary before discussing the implementation. Section 3.4.3 mentions the formal languages that can be recognized by Snort `content` option without the `byte_jump` and `byte_test` options. The `content` option attempts to match the specified string against the TCP stream or the UDP or IP packet payload. When the `content` option is specified multiple times, each string in turn must match against a portion of the payload in the order in which the strings were specified. The `distance` option specifies a minimum distance between the match of the specified string and the match on the last specified string. Similarly, the `within` option specifies a maximum distance between the match on the specified string and the match on the last specified string. The `offset` option

```
(define comp-uricontent
  (lambda (opt rest)
    (list ; need to do this after tcp reassembly
     "declare $did_uri inside tcp.connSym; "
     "declare $found_uricontent inside tcp.connSym; "
     "if (null == $did_uri) {"
     "$found_uricontent = 0; "
     "if (($i = index($buf, \" \")) > 0) " ; skip http/version
     "{ $b2 = substring($buf, $i + 1); "
     "if (($i = index($b2, \" \")) > 0) " ; skip command
     "{ $b3 = substring($b2, $i + 1); "
     "$t = index($b3, \"\\n\"); " ; find end of URI
     "$t2 = index($b3, \"\\r\"); "
     "if (($t < 0) || (($t2 < $t) && ($t2 >= 0))) { $t = $t2; } "
     "$t2 = index($b3, \" \"); "
     "if (($t < 0) || (($t2 < $t) && ($t2 >= 0))) { $t = $t2; } "
     "$t2 = index($b3, \"\\t\"); "
     "if (($t < 0) || (($t2 < $t) && ($t2 >= 0))) { $t = $t2; } "
     "if ($t >= 0) " ; URI found
     "{ $uri = dehex(substring($b3, 0, $t)); "
     "$did_uri = 1; "
     "while (strlen($uri) > 0) { "
     "if ("
     (if (assoc "nocase" opt) "strcasecomp" "strcomp")
     "($uri,"
     (snort-content->n-code-content (cadr (assoc "uricontent" opt)))
     ")) {"
     "$found_uricontent = 1; } "
     "$uri = substring($uri, 1); }"
     "}}}} "
     "if ((null != $did_uri) && ($found_uricontent)) { "
     rest
     "} ")))
```

Figure C.21: Compiling the URI-Content Field

Snort Rule
alert tcp any any -> any 80 (uricontent:"/edit.pl";)

Compiled Form

```
declare $stream_index inside tcp.connSym;
declare $pm inside tcp.connSym;
$t1[0] = -1;
$t1[1] = 0;
if ($pm == null)
  $pm[-1] = $t1;
$t1 = null;
if ($stream_index == null)
  $stream_index = 0;
declare $buf inside tcp.connSym;
declare $segs inside tcp.connSym;
declare $next_seq inside tcp.connSym;
if ((null == $buf) && (tcp.length > 0)) {
  $buf = tcp.blob;
  $next_seq = ulong (ip.blob, 8) + tcp.length;
} else {
  segs[ulong (ip.blob, 8)] = tcp.blob;
  $new_tcp_stream_data = 0;
  while (null != segs[$next_seq]) {
    $new_tcp_stream_data = 1;
    $buf = cat ($buf, segs[$next_seq]);
    $last_seq = $next_seq;
    $next_seq = $next_seq + strlen (segs[$next_seq]);
    segs[$last_seq] = null;
  }
  if ($new_tcp_stream_data) {
    if (1) {
      if (1) {
        if (1) {
          if ((tcp.destport == 80)) {
            declare $did_uri inside tcp.connSym;
            declare $found_uricontent inside tcp.connSym;
            if (null == $did_uri) {
              $found_uricontent = 0;
```

Figure C.22: Example Rule Compilation with URI-Content Field

```
              if (($i = index ($buf, ” ”)) > 0) {
                $b2 = substring ($buf, $i + 1);
                if (($i = index ($b2, ” ”)) > 0) {
                  $b3 = substring ($b2, $i + 1);
                  $t = index ($b3, ”\n”);
                  $t2 = index ($b3, ”\r”);
                  if (($t < 0) || (($t2 < $t) && ($t2 >= 0))) {
                    $t = $t2;
                  }
                  $t2 = index ($b3, ” ”);
                  if (($t < 0) || (($t2 < $t) && ($t2 >= 0))) {
                    $t = $t2;
                  }
                  $t2 = index ($b3, ”\t”);
                  if (($t < 0) || (($t2 < $t) && ($t2 >= 0))) {
                    $t = $t2;
                  }
                  if ($t >= 0) {
                    $uri = dehex (substring ($b3, 0, $t));
                    $did_uri = 1;
                    while (strlen ($uri) > 0) {
                      if (strcomp ($uri, ”/edit.pl”)) {
                        $found_uricontent = 1;
                      }
                      $uri = substring ($uri, 1);
                    }
                  }
                }
              }
            }
            if ((null != $did_uri) && ($found_uricontent)) {
              echo _alert_msg;
            }
          }
        }
      }
    }
  }
}
```

Figure C.22: Continued

specifies an offset from the start of the TCP stream, UDP packet or IP packet to begin searching for the first string match. The `nocase` option specifies that the string should be matched in a case insensitive manner.

The *comp-content-group* code is straightforward conceptually. First, it outputs arrays representing the content strings, the `distance` options, the `within` options, and the `nocase` options. Although N-Code supports lists, it does not provide sufficient primitives on these lists so that they can be used in a useful fashion. Thus it was necessary to use associative arrays, which are supported in N-Code. The code to output these lists as associative arrays is listed in figure C.24. Once this data has been output, *comp-content-group* outputs a fixed procedure with the code for the subtests filled in to preform the content matching. This fixed procedure uses an array of partial matches. Each partial match in the array represents the index of the string that it has matched and the place at which to try for the next partial match. By stepping though the available data, all of the partial matches can be further evaluated. A variable is set when the data stream should not be advanced further, otherwise the data stream is advanced by a byte. For more details, consult either the code in figure C.23 or the example in figure C.1. Because the `byte_test` and `byte_jump` Snort options are content based matches which interact with the `content` option, the code for `byte_test` and `byte_jump` would further complicate the `content` option code. The implementation of `byte_test` and `byte_jump` were delayed until after testing the `content` option, which has not happened because of the unavailability of the NFR IDS.

```
(define comp-content-group
  (lambda (l rest)  ; rest are subtests for the compilation
    (let* ((n (length l))
           (content-l
            ; string length here != string length in n-code (\x00)
            (map (lambda (x)
                    (snort-content->n-code-content (cadr (assoc "content" x))))
                  l))
           (strmatch
            (lambda (c) (if (assoc "nocase" c) "strcasecomp" "strcomp")))
           (do-submatch
            (lambda (v)
              (list
               "$next_str_i = (" v ") + 1; "
               "if ($next_str_i >= " (number->string n) ") { " ;full match?
               "$match = 1; } "
               "else { "
               "$pm[$n_pm] = [ $next_str_i, "
                                   "$stream_index + strlen($content_str[$i])]; "
                "$n_pm = $n_pm + 1; } ")))
          )
       (list
 ; declares for stateful variables need to go into TCP reassemble.
         (list->n-code-array "$content_str" content-l)
         (list->n-code-array "$min_distance"
                             (map (lambda (x)
                                     (assoc-default "distance" x "0"))
                                   l))
         (list->n-code-array "$max_within"
                             (map (lambda (x)
                                     (assoc-default "within" x "(-1)"))
                                   l))
         (list->n-code-array "$nocase"
                             (map (lambda (x)
                                     (if (assoc "nocase" x) "1" "0"))
                                   l))
```

Figure C.23: Compiling the Snort Content Option

```
(list->n-code-array "$offset"
                    (map (lambda (x)
                           (assoc-default "offset" x "0"))
                         l))
"$offset[-1] = 0; "
"$max_within[-1] = -1; "
"$min_distance[-1] = 0; "
"$done_content = 0; "
"$match = 0; "
"$stream_char_done = 0; "
"$stream_index_adj = 0; "
"$s = $buf; "

"while (!$done_content) { " ;depth 1
"$i = -1; "

"while (($i < $n_pm) && (!$match)) {" ;depth2

"$x = $pm[$i]; "   ; [string_index, stream_index]
"if (($max_within[$x[0]] >= 0) && "
     "(($x[1] + $max_within[$x[0]]) < $stream_index)) {"
; already went too far ;depth3
"$pm[$i] = $pm[$n_pm - 1]; "
"$n_pm = $n_pm - 1; "
"$pm[$n_pm] = null; " ; delete
"$i = $i - 1; "
"} else { " ; still in range
"if (($stream_index >= $offset[$x[0]]) && "
     "(($x[1] + $min_distance[$x[0]]) >= $stream_index)) { " ;depth4
"$submatch = 0; "
"if ($nocase[$i]) { " ;depth5
"if (strcasecomp($content_str[$x[0]],$s))  $submatch = 1; "
"} else { "
"if (strcomp($content_str[$x[0]],$s))  $submatch = 1; "
"} " ;depth4
"if ($submatch) { " ;depth5
(do-submatch "$i")
"} else { "
"if (strlen($s) < strlen($content_str[$x[0]])) { " ; should really cache these.
```

Figure C.23: Continued

```
          ;depth6
          "$insuf_length = 1;"
          "}}} " ;depth3

          "$i = $i + 1; "
          "}" ;depth2

          "if ($match) { $done_content = 1; } "
          "if (strlen($s) <= 1) $done_content = 1; "
          "else $s = substring($s, 1); "
          "$stream_index = $stream_index + 1; "
          "if ($insuf_length) $stream_char_done = $stream_char_done + 1; "
          "else $stream_index_adj = $stream_index_adj + 1; "
          "}}" ;depth0

          "$buf = substring($buf, $stream_index_adj); "
          "$stream_index = $stream_index - $stream_index_adj; "
          "if ($match) { " rest " } "
))))
```

Figure C.23: Continued

```
(define (list->n-code-array v l)
 ; "[,,,]" will denote a list in N-Code, not an array
 ; to initialize an array, must initialize every element.
   (let ((n (length l))) ; memoize this
     (let loop
          ((i 0)
           (x l))
        (if (>= i n)
           '()
           (list " " v "[" (number->string i) "] = " (car x) "; "
                 (loop (+ i 1) (cdr x)))))))
```

Figure C.24: Helper Function for Content Compilation